

# Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer's Manual

## Documentation Changes

---

June 2021

**Notice:** The Intel<sup>®</sup> 64 and IA-32 architectures may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-067



Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at [intel.com](http://intel.com), or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting <http://www.intel.com/design/literature.htm>.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

\*Other names and brands may be claimed as the property of others.

Copyright © 1997-2021, Intel Corporation. All Rights Reserved.



# Contents

---

<b>Revision History</b> . . . . .	4
<b>Preface</b> . . . . .	7
<b>Summary Tables of Changes</b> . . . . .	8
<b>Documentation Changes</b> . . . . .	9



# Revision History

---

Revision	Description	Date
-001	<ul style="list-style-type: none"><li>Initial release</li></ul>	November 2002
-002	<ul style="list-style-type: none"><li>Added 1-10 Documentation Changes.</li><li>Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual</li></ul>	December 2002
-003	<ul style="list-style-type: none"><li>Added 9 -17 Documentation Changes.</li><li>Removed Documentation Change #6 - References to bits Gen and Len Deleted.</li><li>Removed Documentation Change #4 - VIF Information Added to CLI Discussion</li></ul>	February 2003
-004	<ul style="list-style-type: none"><li>Removed Documentation changes 1-17.</li><li>Added Documentation changes 1-24.</li></ul>	June 2003
-005	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-24.</li><li>Added Documentation Changes 1-15.</li></ul>	September 2003
-006	<ul style="list-style-type: none"><li>Added Documentation Changes 16- 34.</li></ul>	November 2003
-007	<ul style="list-style-type: none"><li>Updated Documentation changes 14, 16, 17, and 28.</li><li>Added Documentation Changes 35-45.</li></ul>	January 2004
-008	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-45.</li><li>Added Documentation Changes 1-5.</li></ul>	March 2004
-009	<ul style="list-style-type: none"><li>Added Documentation Changes 7-27.</li></ul>	May 2004
-010	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-27.</li><li>Added Documentation Changes 1.</li></ul>	August 2004
-011	<ul style="list-style-type: none"><li>Added Documentation Changes 2-28.</li></ul>	November 2004
-012	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-28.</li><li>Added Documentation Changes 1-16.</li></ul>	March 2005
-013	<ul style="list-style-type: none"><li>Updated title.</li><li>There are no Documentation Changes for this revision of the document.</li></ul>	July 2005
-014	<ul style="list-style-type: none"><li>Added Documentation Changes 1-21.</li></ul>	September 2005
-015	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-21.</li><li>Added Documentation Changes 1-20.</li></ul>	March 9, 2006
-016	<ul style="list-style-type: none"><li>Added Documentation changes 21-23.</li></ul>	March 27, 2006
-017	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-23.</li><li>Added Documentation Changes 1-36.</li></ul>	September 2006
-018	<ul style="list-style-type: none"><li>Added Documentation Changes 37-42.</li></ul>	October 2006
-019	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-42.</li><li>Added Documentation Changes 1-19.</li></ul>	March 2007
-020	<ul style="list-style-type: none"><li>Added Documentation Changes 20-27.</li></ul>	May 2007
-021	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-27.</li><li>Added Documentation Changes 1-6</li></ul>	November 2007
-022	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-6</li><li>Added Documentation Changes 1-6</li></ul>	August 2008
-023	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-6</li><li>Added Documentation Changes 1-21</li></ul>	March 2009





Revision	Description	Date
-024	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-21</li> <li>Added Documentation Changes 1-16</li> </ul>	June 2009
-025	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-16</li> <li>Added Documentation Changes 1-18</li> </ul>	September 2009
-026	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-18</li> <li>Added Documentation Changes 1-15</li> </ul>	December 2009
-027	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-15</li> <li>Added Documentation Changes 1-24</li> </ul>	March 2010
-028	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-24</li> <li>Added Documentation Changes 1-29</li> </ul>	June 2010
-029	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-29</li> </ul>	September 2010
-030	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-29</li> </ul>	January 2011
-031	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-29</li> </ul>	April 2011
-032	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-14</li> </ul>	May 2011
-033	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-14</li> <li>Added Documentation Changes 1-38</li> </ul>	October 2011
-034	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-38</li> <li>Added Documentation Changes 1-16</li> </ul>	December 2011
-035	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-16</li> <li>Added Documentation Changes 1-18</li> </ul>	March 2012
-036	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-18</li> <li>Added Documentation Changes 1-17</li> </ul>	May 2012
-037	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-17</li> <li>Added Documentation Changes 1-28</li> </ul>	August 2012
-038	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-28</li> <li>Add Documentation Changes 1-22</li> </ul>	January 2013
-039	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-22</li> <li>Add Documentation Changes 1-17</li> </ul>	June 2013
-040	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-17</li> <li>Add Documentation Changes 1-24</li> </ul>	September 2013
-041	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-24</li> <li>Add Documentation Changes 1-20</li> </ul>	February 2014
-042	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-20</li> <li>Add Documentation Changes 1-8</li> </ul>	February 2014
-043	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-8</li> <li>Add Documentation Changes 1-43</li> </ul>	June 2014
-044	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-43</li> <li>Add Documentation Changes 1-12</li> </ul>	September 2014
-045	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-12</li> <li>Add Documentation Changes 1-22</li> </ul>	January 2015
-046	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-22</li> <li>Add Documentation Changes 1-25</li> </ul>	April 2015
-047	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-25</li> <li>Add Documentation Changes 1-19</li> </ul>	June 2015



Revision	Description	Date
-048	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-19</li><li>Add Documentation Changes 1-33</li></ul>	September 2015
-049	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-33</li><li>Add Documentation Changes 1-33</li></ul>	December 2015
-050	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-33</li><li>Add Documentation Changes 1-9</li></ul>	April 2016
-051	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-9</li><li>Add Documentation Changes 1-20</li></ul>	June 2016
-052	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-20</li><li>Add Documentation Changes 1-22</li></ul>	September 2016
-053	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-22</li><li>Add Documentation Changes 1-26</li></ul>	December 2016
-054	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-26</li><li>Add Documentation Changes 1-20</li></ul>	March 2017
-055	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-20</li><li>Add Documentation Changes 1-28</li></ul>	July 2017
-056	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-28</li><li>Add Documentation Changes 1-18</li></ul>	October 2017
-057	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-18</li><li>Add Documentation Changes 1-29</li></ul>	December 2017
-058	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-29</li><li>Add Documentation Changes 1-17</li></ul>	March 2018
-059	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-17</li><li>Add Documentation Changes 1-24</li></ul>	May 2018
-060	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-24</li><li>Add Documentation Changes 1-23</li></ul>	November 2018
-061	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-23</li><li>Add Documentation Changes 1-21</li></ul>	January 2019
-062	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-21</li><li>Add Documentation Changes 1-28</li></ul>	May 2019
-063	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-28</li><li>Add Documentation Changes 1-34</li></ul>	October 2019
-064	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-34</li><li>Add Documentation Changes 1-36</li></ul>	May 2020
-065	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-36</li><li>Add Documentation Changes 1-31</li></ul>	November 2020
-066	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-31</li><li>Add Documentation Changes 1-24</li></ul>	April 2021
-067	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-24</li><li>Add Documentation Changes 1-30</li></ul>	June 2021

§

# Preface

---

This document is an update to the specifications contained in the [Affected Documents](#) table below. This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

## Affected Documents

Document Title	Document Number/ Location
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i>	253665
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L</i>	253666
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U</i>	253667
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V-Z</i>	326018
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference</i>	334569
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1</i>	253668
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2</i>	253669
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3</i>	326019
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4</i>	332831
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model Specific Registers</i>	335592

## Nomenclature

**Documentation Changes** include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.

# Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 architectures. This table uses the following notations:

## Codes Used in Summary Tables

Change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

## Documentation Changes(Sheet 1 of 2)

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 1, Volume 1
2	Updates to Chapter 5, Volume 1
3	Updates to Chapter 8, Volume 1
4	Updates to Chapter 13, Volume 1
5	Updates to Chapter 18, Volume 1
6	Updates to Appendix D (previously Appendix E), Volume 1
7	Updates to Chapter 1, Volume 2A
8	Updates to Chapter 2, Volume 2A
9	Updates to Chapter 3, Volume 2A
10	Updates to Chapter 4, Volume 2B
11	Updates to Chapter 5, Volume 2C
12	Updates to Chapter 6, Volume 2D
13	Updates to Chapter 7, Volume 2D
14	Updates to Chapter 1, Volume 3A
15	Updates to Chapter 2, Volume 3A
16	Updates to Chapter 6, Volume 3A
17	Updates to Chapter 9, Volume 3A
18	Updates to Chapter 15, Volume 3B
19	Updates to Chapter 16, Volume 3B
20	Updates to Chapter 17, Volume 3B
21	Updates to Chapter 18, Volume 3B
22	Updates to Chapter 22, Volume 3C
23	Updates to Chapter 27, Volume 3C
24	Updates to Chapter 30, Volume 3C
25	Updates to Chapter 32, Volume 3D
26	Updates to Chapter 35, Volume 3D

## Documentation Changes(Sheet 2 of 2)

No.	DOCUMENTATION CHANGES
27	Updates to Chapter 36, Volume 3D
28	Updates to Appendix A, Volume 3D
29	Updates to Chapter 1, Volume 4
30	Updates to Chapter 2, Volume 4

# ***Documentation Changes***

---

Changes to the Intel® 64 and IA-32 Architectures Software Developer's Manual volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

## 1. Updates to Chapter 1, Volume 1

Change bars and green text show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----  
Changes to this chapter: Updated section 1.1 "Intel® 64 and IA-32 Processors Covered in this Manual". Updated Section 1.2 "Overview of Volume 1: Basic Architecture" to delete the appendix that has been removed and relabeled the existing appendix that followed. The appendix removed is Appendix D, "Guidelines for Writing x87 FPU Exception Handlers". Removed link to outdated reference document.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (order number 253665) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

## 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series



## ABOUT THIS MANUAL

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family
- 8th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
- Intel® Xeon® E processors
- 9th generation Intel® Core™ processors
- 2nd generation Intel® Xeon® Processor Scalable Family

- 10th generation Intel® Core™ processors
- 11th generation Intel® Core™ processors
- 3rd generation Intel® Xeon® Processor Scalable Family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Airmont microarchitecture.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor C series, the Intel® Atom™ processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Processor Scalable Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Processor Scalable Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE

A description of this manual's content follows:

**Chapter 1 — About This Manual.** Gives an overview of all five volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — Intel® 64 and IA-32 Architectures.** Introduces the Intel 64 and IA-32 architectures along with the families of Intel processors that are based on these architectures. It also gives an overview of the common features found in these processors and brief history of the Intel 64 and IA-32 architectures.

**Chapter 3 — Basic Execution Environment.** Introduces the models of memory organization and describes the register set used by applications.

**Chapter 4 — Data Types.** Describes the data types and addressing modes recognized by the processor; provides an overview of real numbers and floating-point formats and of floating-point exceptions.

**Chapter 5 — Instruction Set Summary.** Lists all Intel 64 and IA-32 instructions, divided into technology groups.

**Chapter 6 — Procedure Calls, Interrupts, and Exceptions.** Describes the procedure stack and mechanisms provided for making procedure calls and for servicing interrupts and exceptions.

**Chapter 7 — Programming with General-Purpose Instructions.** Describes basic load and store, program control, arithmetic, and string instructions that operate on basic data types, general-purpose and segment registers; also describes system instructions that are executed in protected mode.

**Chapter 8 — Programming with the x87 FPU.** Describes the x87 floating-point unit (FPU), including floating-point registers and data types; gives an overview of the floating-point instruction set and describes the processor's floating-point exception conditions.

**Chapter 9 — Programming with Intel® MMX™ Technology.** Describes Intel MMX technology, including MMX registers and data types; also provides an overview of the MMX instruction set.

**Chapter 10 — Programming with Intel® Streaming SIMD Extensions (Intel® SSE).** Describes SSE extensions, including XMM registers, the MXCSR register, and packed single-precision floating-point data types; provides an overview of the SSE instruction set and gives guidelines for writing code that accesses the SSE extensions.

**Chapter 11 — Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2).** Describes SSE2 extensions, including XMM registers and packed double-precision floating-point data types; provides an overview of the SSE2 instruction set and gives guidelines for writing code that accesses SSE2 extensions. This chapter also describes SIMD floating-point exceptions that can be generated with SSE and SSE2 instructions. It also provides general guidelines for incorporating support for SSE and SSE2 extensions into operating system and applications code.

**Chapter 12 — Programming with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® Streaming SIMD Extensions 4 (Intel® SSE4) and Intel® AES New Instructions (Intel® AES-NI).** Provides an overview of the SSE3 instruction set, Supplemental SSE3, SSE4, AESNI instructions, and guidelines for writing code that access these extensions.

**Chapter 13 — Managing State Using the XSAVE Feature Set.** Describes the XSAVE feature set instructions and explains how software can enable the XSAVE feature set and XSAVE-enabled features.

**Chapter 14 — Programming with AVX, FMA and AVX2.** Provides an overview of the Intel® AVX instruction set, FMA and Intel AVX2 extensions and gives guidelines for writing code that access these extensions.

**Chapter 15 — Programming with Intel® AVX-512.** Provides an overview of the Intel® AVX-512 instruction set extensions and gives guidelines for writing code that access these extensions.

**Chapter 16 — Programming with Intel Transactional Synchronization Extensions.** Describes the instruction extensions that support lock elision techniques to improve the performance of multi-threaded software with contended locks.

**Chapter 17 — Intel® Memory Protection Extensions.** Provides an overview of the Intel® Memory Protection Extensions and gives guidelines for writing code that access these extensions.

**Chapter 18 — Control-flow Enforcement Technology.** Provides an overview of the Control-flow Enforcement Technology (CET) and gives guidelines for writing code that access these extensions.

**Chapter 19 — Input/Output.** Describes the processor's I/O mechanism, including I/O port addressing, I/O instructions, and I/O protection mechanisms.

**Chapter 20 — Processor Identification and Feature Determination.** Describes how to determine the CPU type and features available in the processor.

**Appendix A — EFLAGS Cross-Reference.** Summarizes how the IA-32 instructions affect the flags in the EFLAGS register.

**Appendix B — EFLAGS Condition Codes.** Summarizes how conditional jump, move, and 'byte set on condition code' instructions use condition code flags (OF, CF, ZF, SF, and PF) in the EFLAGS register.

**Appendix C — Floating-Point Exceptions Summary.** Summarizes exceptions raised by the x87 FPU floating-point and SSE/SSE2/SSE3 floating-point instructions.

**Appendix D — Guidelines for Writing SIMD Floating-Point Exception Handlers.** Gives guidelines for writing exception handlers for exceptions generated by SSE/SSE2/SSE3 floating-point instructions.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. This notation is described below.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. See Figure 1-1.



For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.3 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, 0F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The “B” designation is only used in situations where confusion as to the type of number might arise.

### 1.3.4 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

```
CS:EIP
```

### 1.3.5 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

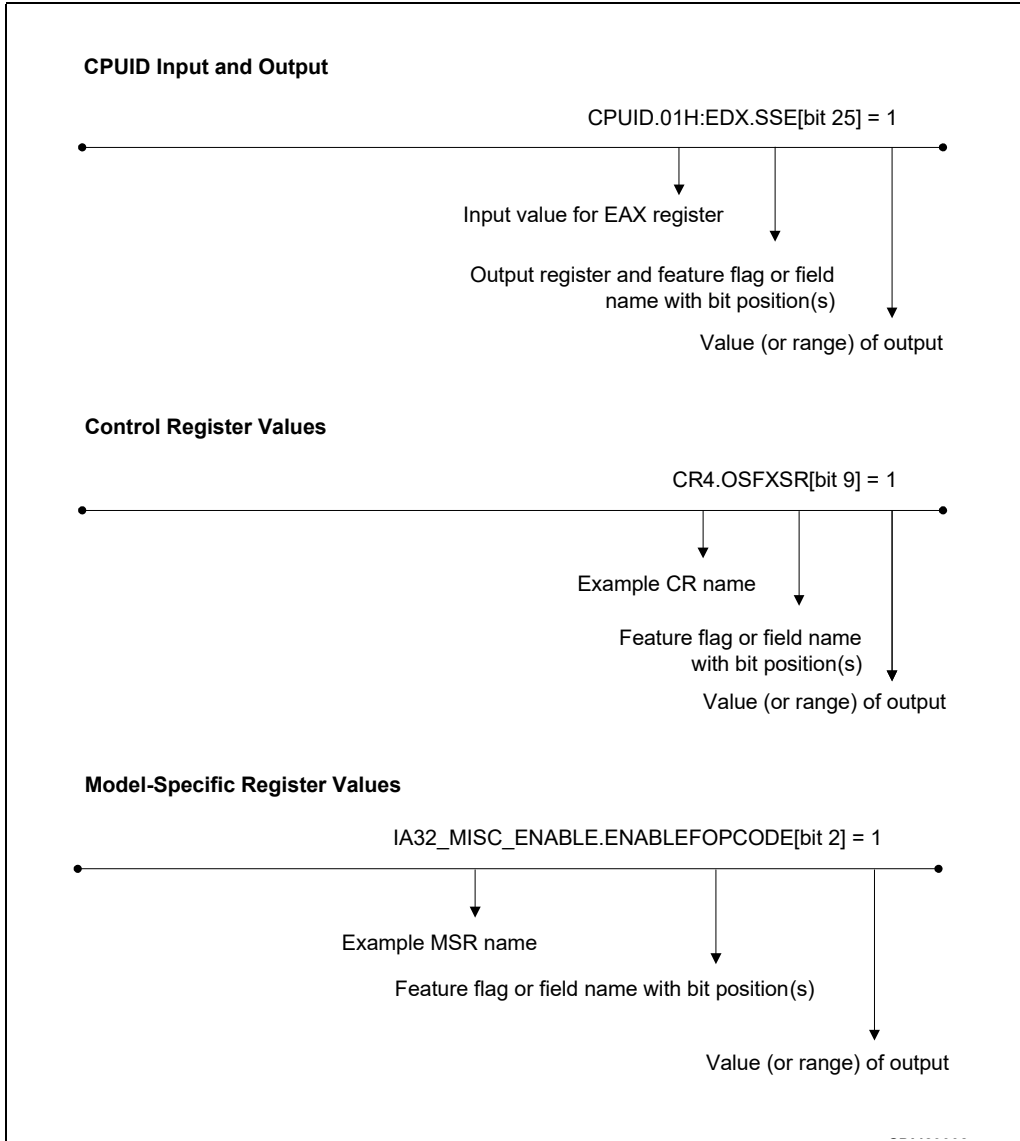


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions that produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

<https://software.intel.com/en-us/articles/intel-sdm>

See also:

- The latest security information on Intel® products:  
<https://www.intel.com/content/www/us/en/security-center/default.html>
- Software developer resources, guidance and insights for security advisories:  
<https://software.intel.com/security-software-guidance/>
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Software Development Tools:  
<https://software.intel.com/en-us/intel-sdp-home>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):  
<https://software.intel.com/en-us/articles/intel-sdm>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<https://software.intel.com/en-us/articles/intel-sdm#optimization>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Intel® Software Guard Extensions (Intel® SGX) Information  
<https://software.intel.com/en-us/isa-extensions/intel-sgx>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
<https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf>
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<https://software.intel.com/sites/default/files/22/30/25602>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference  
<https://software.intel.com/en-us/isa-extensions>

More relevant links are:

- Intel® Developer Zone:  
<https://software.intel.com/en-us>
- Developer centers:  
<http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>





## 2. Updates to Chapter 5, Volume 1

Change bars and green text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----  
Changes to this chapter: Update to Table 5-2, "Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors" to combine the Intel SHA extensions and SHA-NI lines into one. Added protection keys.

# CHAPTER 5

## INSTRUCTION SET SUMMARY

---

This chapter provides an abridged overview of Intel 64 and IA-32 instructions. Instructions are divided into the following groups:

- Section 5.1, “General-Purpose Instructions”.
- Section 5.2, “x87 FPU Instructions”.
- Section 5.3, “x87 FPU AND SIMD State Management Instructions”.
- Section 5.4, “MMX™ Instructions”.
- Section 5.5, “SSE Instructions”.
- Section 5.6, “SSE2 Instructions”.
- Section 5.7, “SSE3 Instructions”.
- Section 5.8, “Supplemental Streaming SIMD Extensions 3 (SSSE3) Instructions”.
- Section 5.9, “SSE4 Instructions”.
- Section 5.10, “SSE4.1 Instructions”.
- Section 5.11, “SSE4.2 Instruction Set”.
- Section 5.12, “Intel® AES-NI and PCLMULQDQ”.
- Section 5.13, “Intel® Advanced Vector Extensions (Intel® AVX)”.
- Section 5.14, “16-bit Floating-Point Conversion”.
- Section 5.15, “Fused-Multiply-ADD (FMA)”.
- Section 5.16, “Intel® Advanced Vector Extensions 2 (Intel® AVX2)”.
- Section 5.17, “Intel® Transactional Synchronization Extensions (Intel® TSX)”.
- Section 5.18, “Intel® SHA Extensions”.
- Section 5.19, “Intel® Advanced Vector Extensions 512 (Intel® AVX-512)”.
- Section 5.20, “System Instructions”.
- Section 5.21, “64-Bit Mode Instructions”.
- Section 5.22, “Virtual-Machine Extensions”.
- Section 5.23, “Safer Mode Extensions”.
- Section 5.24, “Intel® Memory Protection Extensions”.
- Section 5.25, “Intel® Software Guard Extensions”.
- Section 5.26, “Shadow Stack Management Instructions”.
- Section 5.27, “Control Transfer Terminating Instructions”.

Table 5-1 lists the groups and IA-32 processors that support each group. More recent instruction set extensions are listed in Table 5-2. Within these groups, most instructions are collected into functional subgroups.

**Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors**

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors.
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.

**Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors (Contd.)**

<b>Instruction Set Architecture</b>	<b>Intel 64 and IA-32 Processor Support</b>
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxx, 5xxx, 7xxx Series, Intel Atom processors.
SSSE3 Extensions	Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors.
IA-32e mode: 64-bit mode instructions	Intel 64 processors.
System Instructions	Intel 64 and IA-32 processors.
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology.
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx.

**Table 5-2. Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors**

<b>Instruction Set Architecture</b>	<b>Processor Generation Introduction</b>
SSE4.1 Extensions	Intel® Xeon® processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel® Core™ 2 Extreme processors QX9000 series, Intel® Core™ 2 Quad processor Q9000 series, Intel® Core™ 2 Duo processors 8000 series and T9000 series, Intel Atom® processor based on Silvermont microarchitecture.
SSE4.2 Extensions, CRC32, POPCNT	Intel® Core™ i7 965 processor, Intel® Xeon® processors X3400, X3500, X5500, X6500, X7500 series, Intel Atom processor based on Silvermont microarchitecture.
Intel® AES-NI, PCLMULQDQ	Intel® Xeon® processor E7 series, Intel® Xeon® processors X3600 and X5600, Intel® Core™ i7 980X processor, Intel Atom processor based on Silvermont microarchitecture. Use CPUID to verify presence of Intel AES-NI and PCLMULQDQ across Intel® Core™ processor families.
Intel® AVX	Intel® Xeon® processor E3 and E5 families, 2nd Generation Intel® Core™ i7, i5, i3 processor 2xxx families.
F16C	3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Intel® Xeon® processor E5 v2 and E7 v2 families.
RDRAND	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom processor based on Silvermont microarchitecture.
FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom® processor based on Goldmont microarchitecture.
FMA, AVX2, BMI1, BMI2, INVPCID, LZCNT, Intel® TSX	Intel® Xeon® processor E3/E5/E7 v3 product families, 4th Generation Intel® Core™ processor family.
MOVBE	Intel Xeon processor E3/E5/E7 v3 product families, 4th Generation Intel Core processor family, Intel Atom processors.
PREFETCHW	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family, Intel Atom processor based on Silvermont microarchitecture.

**Table 5-2. Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors (Contd.)**

<b>Instruction Set Architecture</b>	<b>Processor Generation Introduction</b>
ADX	Intel Core M processor family, 5th Generation Intel Core processor family.
RDSEED, CLAC, STAC	Intel Core M processor family, 5th Generation Intel Core processor family, Intel Atom processor based on Goldmont microarchitecture.
AVX512ER, AVX512PF, PREFETCHWT1	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series.
AVX512F, AVX512CD	Intel Xeon Phi Processor 3200, 5200, 7200 Series, Intel® Xeon® Processor Scalable Family, Intel® Core™ i3-8121U processor.
CLFLUSHOPT, XSAVEC, XSAVES, Intel® MPX	Intel Xeon Processor Scalable Family, 6th Generation Intel® Core™ processor family, Intel Atom processor based on Goldmont microarchitecture.
SGX1	6th Generation Intel Core processor family, Intel Atom® processor based on Goldmont Plus microarchitecture.
AVX512DQ, AVX512BW, AVX512VL	Intel Xeon Processor Scalable Family, Intel Core i3-8121U processor based on Cannon Lake microarchitecture.
CLWB	Intel Xeon Processor Scalable Family, Intel Atom® processor based on Tremont microarchitecture, 11th Generation Intel Core processor family based on Tiger Lake microarchitecture.
PKU	Intel Xeon Processor Scalable Family, 10th generation Intel® Core™ processors based on Comet Lake microarchitecture.
AVX512_IFMA, AVX512_VBMI	Intel Core i3-8121U processor based on Cannon Lake microarchitecture.
Intel® SHA Extensions	Intel Core i3-8121U processor based on Cannon Lake microarchitecture, Intel Atom processor based on Goldmont microarchitecture, 3rd Generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture.
UMIP	Intel Core i3-8121U processor based on Cannon Lake microarchitecture, Intel Atom processor based on Goldmont Plus microarchitecture.
PTWRITE	Intel Atom processor based on Goldmont Plus microarchitecture.
RDPID	10th Generation Intel® Core™ processor family based on Ice Lake microarchitecture, Intel Atom processor based on Goldmont Plus microarchitecture.
AVX512_4FMAPS, AVX512_4VNNIW	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series.
AVX512_VNNI	2nd Generation Intel® Xeon® Processor Scalable Family, 10th Generation Intel Core processor family based on Ice Lake microarchitecture.
AVX512_VPOPCNTDQ	Intel Xeon Phi Processor 7215, 7285, 7295 Series, 10th Generation Intel Core processor family based on Ice Lake microarchitecture.
Fast Short REP MOV	10th Generation Intel Core processor family based on Ice Lake microarchitecture.
GFNI (SSE)	10th Generation Intel Core processor family based on Ice Lake microarchitecture, Intel Atom processor based on Tremont microarchitecture.
VAES, GFNI (AVX/AVX512), AVX512_VBMI2, VPCLMULQDQ, AVX512_BITALG	10th Generation Intel Core processor family based on Ice Lake microarchitecture.
ENCLV	Intel Atom processor based on Tremont microarchitecture, 3rd Generation Intel® Xeon® Processor Scalable Processors based on Ice Lake microarchitecture.
Split Lock Detection	10th Generation Intel Core processor family based on Ice Lake microarchitecture, Intel Atom processor based on Tremont microarchitecture.

Table 5-2. Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors (Contd.)

Instruction Set Architecture	Processor Generation Introduction
CLDEMOTÉ	Intel Atom processor based on Tremont microarchitecture.
Direct stores: MOVDIRI, MOVDIR64B	Intel Atom processor based on Tremont microarchitecture, 11th Generation Intel Core processor family based on Tiger Lake microarchitecture.
User wait: TPAUSE, UMONITOR, UMWAIT	Intel Atom processor based on Tremont microarchitecture.
AVX512_BF16	3rd Generation Intel® Xeon® Processor Scalable Processors based on Cooper Lake product.
AVX512_VP2INTERSECT	11th Generation Intel Core processor family based on Tiger Lake microarchitecture.
Key Locker <sup>1</sup>	11th Generation Intel Core processor family based on Tiger Lake microarchitecture.
Control-flow Enforcement Technology (CET)	11th Generation Intel Core processor family based on Tiger Lake microarchitecture.
MKTME <sup>2</sup> , PCONFIG	3rd Generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture.
WBNOINVD	3rd Generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture.
Supervisor Memory Protection Keys (PKS)	Future Intel processors

**NOTES:**

1. Details on Key Locker can be found in the Intel Key Locker Specification here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

2. Further details on MKTME usage can be found here:

<https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf>.

The following sections list instructions in each major group and subgroup. Given for each instruction is its mnemonic and descriptive names. When two or more mnemonics are given (for example, CMOVA/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move if not below or equal) represent the same condition. For detailed information about specific instructions, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

## 5.1 GENERAL-PURPOSE INSTRUCTIONS

The general-purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run on Intel 64 and IA-32 processors. They operate on data contained in memory, in the general-purpose registers (EAX, EBX, ECX, EDX, EDI, ESI, EBX, and ESP) and in the EFLAGS register. They also operate on address information contained in memory, the general-purpose registers, and the segment registers (CS, DS, SS, ES, FS, and GS).

This group of instructions includes the data transfer, binary integer arithmetic, decimal arithmetic, logic operations, shift and rotate, bit and byte operations, program control, string, flag control, segment register operations, and miscellaneous subgroups. The sections that follow introduce each subgroup.

For more detailed information on general purpose-instructions, see Chapter 7, "Programming With General-Purpose Instructions."

### 5.1.1 Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero.
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal.
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below.
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater.
CMOVC	Conditional move if carry.
CMOVNC	Conditional move if not carry.
CMOVO	Conditional move if overflow.
CMOVNO	Conditional move if not overflow.
CMOVS	Conditional move if sign (negative).
CMOVNS	Conditional move if not sign (non-negative).
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd.
XCHG	Exchange.
BSWAP	Byte swap.
XADD	Exchange and add.
CMPXCHG	Compare and exchange.
CMPXCHG8B	Compare and exchange 8 bytes.
PUSH	Push onto stack.
POP	Pop off of stack.
PUSHA/PUSHAD	Push general-purpose registers onto stack.
POPA/POPAD	Pop general-purpose registers from stack.
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register.
MOVSX	Move and sign extend.
MOVZX	Move and zero extend.

## 5.1.2 Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADCX	Unsigned integer add with carry.
ADOX	Unsigned integer add with overflow.
ADD	Integer add.
ADC	Add with carry.
SUB	Subtract.
SBB	Subtract with borrow.
IMUL	Signed multiply.
MUL	Unsigned multiply.
IDIV	Signed divide.

DIV	Unsigned divide.
INC	Increment.
DEC	Decrement.
NEG	Negate.
CMP	Compare.

### 5.1.3 Decimal Arithmetic Instructions

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

DAA	Decimal adjust after addition.
DAS	Decimal adjust after subtraction.
AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.
AAM	ASCII adjust after multiplication.
AAD	ASCII adjust before division.

### 5.1.4 Logical Instructions

The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.

AND	Perform bitwise logical AND.
OR	Perform bitwise logical OR.
XOR	Perform bitwise logical exclusive OR.
NOT	Perform bitwise logical NOT.

### 5.1.5 Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in word and doubleword operands.

SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.
SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

### 5.1.6 Bit and Byte Instructions

Bit instructions test and modify individual bits in word and doubleword operands. Byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.

BT	Bit test.
BTS	Bit test and set.
BTR	Bit test and reset.
BTC	Bit test and complement.
BSF	Bit scan forward.



BSR	Bit scan reverse.
SETE/SETZ	Set byte if equal/Set byte if zero.
SETNE/SETNZ	Set byte if not equal/Set byte if not zero.
SETA/SETNBE	Set byte if above/Set byte if not below or equal.
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry.
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry.
SETBE/SETNA	Set byte if below or equal/Set byte if not above.
SETG/SETNLE	Set byte if greater/Set byte if not less or equal.
SETGE/SETNL	Set byte if greater or equal/Set byte if not less.
SETL/SETNGE	Set byte if less/Set byte if not greater or equal.
SETLE/SETNG	Set byte if less or equal/Set byte if not greater.
SETS	Set byte if sign (negative).
SETNS	Set byte if not sign (non-negative).
SETO	Set byte if overflow.
SETNO	Set byte if not overflow.
SETPE/SETP	Set byte if parity even/Set byte if parity.
SETPO/SETNP	Set byte if parity odd/Set byte if not parity.
TEST	Logical compare.
CRC32 <sup>1</sup>	Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
POPCNT <sup>2</sup>	This instruction calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

### 5.1.7 Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

JMP	Jump.
JE/JZ	Jump if equal/Jump if zero.
JNE/JNZ	Jump if not equal/Jump if not zero.
JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal/Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/Jump if not above.
JG/JNLE	Jump if greater/Jump if not less or equal.
JGE/JNL	Jump if greater or equal/Jump if not less.
JL/JNGE	Jump if less/Jump if not greater or equal.
JLE/JNG	Jump if less or equal/Jump if not greater.
JC	Jump if carry.
JNC	Jump if not carry.
JO	Jump if overflow.
JNO	Jump if not overflow.
JS	Jump if sign (negative).
JNS	Jump if not sign (non-negative).

1. Processor support of CRC32 is enumerated by CPUID.01:ECX[SSE4.2] = 1

2. Processor support of POPCNT is enumerated by CPUID.01:ECX[POPCNT] = 1

JPO/JNP	Jump if parity odd/Jump if not parity.
JPE/JP	Jump if parity even/Jump if parity.
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero.
LOOP	Loop with ECX counter.
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal.
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal.
CALL	Call procedure.
RET	Return.
IRET	Return from interrupt.
INT	Software interrupt.
INTO	Interrupt on overflow.
BOUND	Detect value out of range.
ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

### 5.1.8 String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

MOVS/MOVS	Move string/Move byte string.
MOVS/MOVSW	Move string/Move word string.
MOVS/MOVSD	Move string/Move doubleword string.
CMPS/CMPSB	Compare string/Compare byte string.
CMPS/CMPSW	Compare string/Compare word string.
CMPS/CMPSD	Compare string/Compare doubleword string.
SCAS/SCASB	Scan string/Scan byte string.
SCAS/SCASW	Scan string/Scan word string.
SCAS/SCASD	Scan string/Scan doubleword string.
LODS/LODSB	Load string/Load byte string.
LODS/LODSW	Load string/Load word string.
LODS/LODSD	Load string/Load doubleword string.
STOS/STOSB	Store string/Store byte string.
STOS/STOSW	Store string/Store word string.
STOS/STOSD	Store string/Store doubleword string.
REP	Repeat while ECX not zero.
REPE/REPZ	Repeat while equal/Repeat while zero.
REPNE/REPZ	Repeat while not equal/Repeat while not zero.

### 5.1.9 I/O Instructions

These instructions move data between the processor's I/O ports and a register or memory.

IN	Read from a port.
OUT	Write to a port.
INS/INSB	Input string from port/Input byte string from port.
INS/INSW	Input string from port/Input word string from port.
INS/INSD	Input string from port/Input doubleword string from port.
OUTS/OUTSB	Output string to port/Output byte string to port.
OUTS/OUTSW	Output string to port/Output word string to port.

OUTS/OUTSD            Output string to port/Output doubleword string to port.

### 5.1.10 Enter and Leave Instructions

These instructions provide machine-language support for procedure calls in block-structured languages.

ENTER                    High-level procedure entry.  
LEAVE                    High-level procedure exit.

### 5.1.11 Flag Control (EFLAG) Instructions

The flag control instructions operate on the flags in the EFLAGS register.

STC                      Set carry flag.  
CLC                      Clear the carry flag.  
CMC                      Complement the carry flag.  
CLD                      Clear the direction flag.  
STD                      Set direction flag.  
LAHF                     Load flags into AH register.  
SAHF                     Store AH register into flags.  
PUSHF/PUSHFD         Push EFLAGS onto stack.  
POPF/POPF             Pop EFLAGS from stack.  
STI                      Set interrupt flag.  
CLI                      Clear the interrupt flag.

### 5.1.12 Segment Register Instructions

The segment register instructions allow far pointers (segment addresses) to be loaded into the segment registers.

LDS                      Load far pointer using DS.  
LES                      Load far pointer using ES.  
LFS                      Load far pointer using FS.  
LGS                      Load far pointer using GS.  
LSS                      Load far pointer using SS.

### 5.1.13 Miscellaneous Instructions

The miscellaneous instructions provide such functions as loading an effective address, executing a “no-operation,” and retrieving processor identification information.

LEA                      Load effective address.  
NOP                      No operation.  
UD                      Undefined instruction.  
XLAT/XLATB            Table lookup translation.  
CPUID                    Processor identification.  
MOVBE<sup>1</sup>                Move data after swapping data bytes.  
PREFETCHW             Prefetch data into cache in anticipation of write.  
PREFETCHWT1          Prefetch hint T1 with intent to write.

---

1. Processor support of MOVBE is enumerated by CPUID.01:ECX.MOVBE[bit 22] = 1.

CLFLUSH	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy.
CLFLUSHOPT	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy with optimized memory system throughput.

### 5.1.14 User Mode Extended State Save/Restore Instructions

XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XRSTOR	Restore processor extended states from memory.
XGETBV	Reads the state of an extended control register.

### 5.1.15 Random Number Generator Instructions

RDRAND	Retrieves a random number generated from hardware.
RDSEED	Retrieves a random number generated from hardware.

### 5.1.16 BMI1, BMI2

ANDN	Bitwise AND of first source with inverted 2nd source operands.
BEXTR	Contiguous bitwise extract.
BLSI	Extract lowest set bit.
BLSMSK	Set all lower bits below first set bit to 1.
BLSR	Reset lowest set bit.
BZHI	Zero high bits starting from specified bit position.
LZCNT	Count the number leading zero bits.
MULX	Unsigned multiply without affecting arithmetic flags.
PDEP	Parallel deposit of bits using a mask.
PEXT	Parallel extraction of bits using a mask.
RORX	Rotate right without affecting arithmetic flags.
SARX	Shift arithmetic right.
SHLX	Shift logic left.
SHRX	Shift logic right.
TZCNT	Count the number trailing zero bits.

#### 5.1.16.1 Detection of VEX-encoded GPR Instructions, LZCNT and TZCNT, PREFETCHW

VEX-encoded general-purpose instructions do not operate on any vector registers.

There are separate feature flags for the following subsets of instructions that operate on general purpose registers, and the detection requirements for hardware support are:

CPUID.(EAX=07H, ECX=0H):EBX.BMI1[bit 3]: if 1 indicates the processor supports the first group of advanced bit manipulation extensions (ANDN, BEXTR, BLSI, BLSMSK, BLSR, TZCNT);

CPUID.(EAX=07H, ECX=0H):EBX.BMI2[bit 8]: if 1 indicates the processor supports the second group of advanced bit manipulation extensions (BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX);

CPUID.EAX=80000001H:ECX.LZCNT[bit 5]: if 1 indicates the processor supports the LZCNT instruction.

CPUID.EAX=80000001H:ECX.PREFETCHW[bit 8]: if 1 indicates the processor supports the PREFETCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFETCHWT1[bit 0]: if 1 indicates the processor supports the PREFETCHWT1 instruction.

## 5.2 X87 FPU INSTRUCTIONS

The x87 FPU instructions are executed by the processor's x87 FPU. These instructions operate on floating-point, integer, and binary-coded decimal (BCD) operands. For more detail on x87 FPU instructions, see Chapter 8, "Programming with the x87 FPU."

These instructions are divided into the following subgroups: data transfer, load constants, and FPU control instructions. The sections that follow introduce each subgroup.

### 5.2.1 x87 FPU Data Transfer Instructions

The data transfer instructions move floating-point, integer, and BCD values between memory and the x87 FPU registers. They also perform conditional move operations on floating-point operands.

FLD	Load floating-point value.
FST	Store floating-point value.
FSTP	Store floating-point value and pop.
FILD	Load integer.
FIST	Store integer.
FISTP <sup>1</sup>	Store integer and pop.
FBLD	Load BCD.
FBSTP	Store BCD and pop.
FXCH	Exchange registers.
FCMOVE	Floating-point conditional move if equal.
FCMOVNE	Floating-point conditional move if not equal.
FCMOVB	Floating-point conditional move if below.
FCMOVBE	Floating-point conditional move if below or equal.
FCMOVNB	Floating-point conditional move if not below.
FCMOVNBE	Floating-point conditional move if not below or equal.
FCMOVU	Floating-point conditional move if unordered.
FCMOVNU	Floating-point conditional move if not unordered.

### 5.2.2 x87 FPU Basic Arithmetic Instructions

The basic arithmetic instructions perform basic arithmetic operations on floating-point and integer operands.

FADD	Add floating-point
FADDP	Add floating-point and pop
FIADD	Add integer
FSUB	Subtract floating-point
FSUBP	Subtract floating-point and pop
FISUB	Subtract integer
FSUBR	Subtract floating-point reverse
FSUBRP	Subtract floating-point reverse and pop
FISUBR	Subtract integer reverse
FMUL	Multiply floating-point
FMULP	Multiply floating-point and pop
FIMUL	Multiply integer
FDIV	Divide floating-point

---

1. SSE3 provides an instruction FISTTP for integer conversion.

FDIVP	Divide floating-point and pop
FIDIV	Divide integer
FDIVR	Divide floating-point reverse
FDIVRP	Divide floating-point reverse and pop
FIDIVR	Divide integer reverse
FPREM	Partial remainder
FPREM1	IEEE Partial remainder
FABS	Absolute value
FCHS	Change sign
FRNDINT	Round to integer
FSCALE	Scale by power of two
FSQRT	Square root
FXTRACT	Extract exponent and significand

### 5.2.3 x87 FPU Comparison Instructions

The compare instructions examine or compare floating-point or integer operands.

FCOM	Compare floating-point.
FCOMP	Compare floating-point and pop.
FCOMPP	Compare floating-point and pop twice.
FUCOM	Unordered compare floating-point.
FUCOMP	Unordered compare floating-point and pop.
FUCOMPP	Unordered compare floating-point and pop twice.
FICOM	Compare integer.
FICOMP	Compare integer and pop.
FCOMI	Compare floating-point and set EFLAGS.
FUCOMI	Unordered compare floating-point and set EFLAGS.
FCOMIP	Compare floating-point, set EFLAGS, and pop.
FUCOMIP	Unordered compare floating-point, set EFLAGS, and pop.
FTST	Test floating-point (compare with 0.0).
FXAM	Examine floating-point.

### 5.2.4 x87 FPU Transcendental Instructions

The transcendental instructions perform basic trigonometric and logarithmic operations on floating-point operands.

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$y \cdot \log_2 x$
FYL2XP1	$y \cdot \log_2(x+1)$

### 5.2.5 x87 FPU Load Constants Instructions

The load constants instructions load common constants, such as  $\pi$ , into the x87 floating-point registers.

FLD1	Load +1.0
FLDZ	Load +0.0
FLDPI	Load $\pi$
FLDL2E	Load $\log_2 e$
FLDLN2	Load $\log_e 2$
FLDL2T	Load $\log_2 10$
FLDLG2	Load $\log_{10} 2$

## 5.2.6 x87 FPU Control Instructions

The x87 FPU control instructions operate on the x87 FPU register stack and save and restore the x87 FPU state.

FINCSTP	Increment FPU register stack pointer.
FDECSTP	Decrement FPU register stack pointer.
FFREE	Free floating-point register.
FINIT	Initialize FPU after checking error conditions.
FNINIT	Initialize FPU without checking error conditions.
FCLEX	Clear floating-point exception flags after checking for error conditions.
FNCLEX	Clear floating-point exception flags without checking for error conditions.
FSTCW	Store FPU control word after checking error conditions.
FNSTCW	Store FPU control word without checking error conditions.
FLDCW	Load FPU control word.
FSTENV	Store FPU environment after checking error conditions.
FNSTENV	Store FPU environment without checking error conditions.
FLDENV	Load FPU environment.
FSAVE	Save FPU state after checking error conditions.
FNSAVE	Save FPU state without checking error conditions.
FRSTOR	Restore FPU state.
FSTSW	Store FPU status word after checking error conditions.
FNSTSW	Store FPU status word without checking error conditions.
WAIT/FWAIT	Wait for FPU.
FNOP	FPU no operation.

## 5.3 X87 FPU AND SIMD STATE MANAGEMENT INSTRUCTIONS

Two state management instructions were introduced into the IA-32 architecture with the Pentium II processor family:

FXSAVE	Save x87 FPU and SIMD state.
FXRSTOR	Restore x87 FPU and SIMD state.

Initially, these instructions operated only on the x87 FPU (and MMX) registers to perform a fast save and restore, respectively, of the x87 FPU and MMX state. With the introduction of SSE extensions in the Pentium III processor family, these instructions were expanded to also save and restore the state of the XMM and MXCSR registers. Intel 64 architecture also supports these instructions.

See Section 10.5, "FXSAVE and FXRSTOR Instructions," for more detail.

## 5.4 MMX™ INSTRUCTIONS

Four extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2

extensions, and SSE3 extensions. For a discussion that puts SIMD instructions in their historical context, see Section 2.2.7, “SIMD Instructions.”

MMX instructions operate on packed byte, word, doubleword, or quadword integer operands contained in memory, in MMX registers, and/or in general-purpose registers. For more detail on these instructions, see Chapter 9, “Programming with Intel® MMX™ Technology.”

MMX instructions can only be executed on Intel 64 and IA-32 processors that support the MMX technology. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

MMX instructions are divided into the following subgroups: data transfer, conversion, packed arithmetic, comparison, logical, shift and rotate, and state management instructions. The sections that follow introduce each subgroup.

### 5.4.1 MMX Data Transfer Instructions

The data transfer instructions move doubleword and quadword operands between MMX registers and between MMX registers and memory.

MOVD	Move doubleword.
MOVQ	Move quadword.

### 5.4.2 MMX Conversion Instructions

The conversion instructions pack and unpack bytes, words, and doublewords

PACKSSWB	Pack words into bytes with signed saturation.
PACKSSDW	Pack doublewords into words with signed saturation.
PACKUSWB	Pack words into bytes with unsigned saturation.
PUNPCKHBW	Unpack high-order bytes.
PUNPCKHWD	Unpack high-order words.
PUNPCKHDQ	Unpack high-order doublewords.
PUNPCKLBW	Unpack low-order bytes.
PUNPCKLWD	Unpack low-order words.
PUNPCKLDQ	Unpack low-order doublewords.

### 5.4.3 MMX Packed Arithmetic Instructions

The packed arithmetic instructions perform packed integer arithmetic on packed byte, word, and doubleword integers.

PADDB	Add packed byte integers.
PADDW	Add packed word integers.
PADDQ	Add packed doubleword integers.
PADDSB	Add packed signed byte integers with signed saturation.
PADDSD	Add packed signed word integers with signed saturation.
PADDUSB	Add packed unsigned byte integers with unsigned saturation.
PADDUSW	Add packed unsigned word integers with unsigned saturation.
PSUBB	Subtract packed byte integers.
PSUBW	Subtract packed word integers.
PSUBQ	Subtract packed doubleword integers.
PSUBSB	Subtract packed signed byte integers with signed saturation.
PSUBSD	Subtract packed signed word integers with signed saturation.



PSUBUSB	Subtract packed unsigned byte integers with unsigned saturation.
PSUBUSW	Subtract packed unsigned word integers with unsigned saturation.
PMULHW	Multiply packed signed word integers and store high result.
PMULLW	Multiply packed signed word integers and store low result.
PMADDWD	Multiply and add packed word integers.

#### 5.4.4 MMX Comparison Instructions

The compare instructions compare packed bytes, words, or doublewords.

PCMPEQB	Compare packed bytes for equal.
PCMPEQW	Compare packed words for equal.
PCMPEQD	Compare packed doublewords for equal.
PCMPGTB	Compare packed signed byte integers for greater than.
PCMPGTW	Compare packed signed word integers for greater than.
PCMPGTD	Compare packed signed doubleword integers for greater than.

#### 5.4.5 MMX Logical Instructions

The logical instructions perform AND, AND NOT, OR, and XOR operations on quadword operands.

PAND	Bitwise logical AND.
PANDN	Bitwise logical AND NOT.
POR	Bitwise logical OR.
PXOR	Bitwise logical exclusive OR.

#### 5.4.6 MMX Shift and Rotate Instructions

The shift and rotate instructions shift and rotate packed bytes, words, or doublewords, or quadwords in 64-bit operands.

PSLLW	Shift packed words left logical.
PSLLD	Shift packed doublewords left logical.
PSLLQ	Shift packed quadword left logical.
PSRLW	Shift packed words right logical.
PSRLD	Shift packed doublewords right logical.
PSRLQ	Shift packed quadword right logical.
PSRAW	Shift packed words right arithmetic.
PSRAD	Shift packed doublewords right arithmetic.

#### 5.4.7 MMX State Management Instructions

The EMMS instruction clears the MMX state from the MMX registers.

EMMS	Empty MMX state.
------	------------------

### 5.5 SSE INSTRUCTIONS

SSE instructions represent an extension of the SIMD execution model introduced with the MMX technology. For more detail on these instructions, see Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."

SSE instructions can only be executed on Intel 64 and IA-32 processors that support SSE extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

SSE instructions are divided into four subgroups (note that the first subgroup has subordinate subgroups of its own):

- SIMD single-precision floating-point instructions that operate on the XMM registers.
- MXCSR state management instructions.
- 64-bit SIMD integer instructions that operate on the MMX registers.
- Cacheability control, prefetch, and instruction ordering instructions.

The following sections provide an overview of these groups.

## 5.5.1 SSE SIMD Single-Precision Floating-Point Instructions

These instructions operate on packed and scalar single-precision floating-point values located in XMM registers and/or memory. This subgroup is further divided into the following subordinate subgroups: data transfer, packed arithmetic, comparison, logical, shuffle and unpack, and conversion instructions.

### 5.5.1.1 SSE Data Transfer Instructions

SSE data transfer instructions move packed and scalar single-precision floating-point operands between XMM registers and between XMM registers and memory.

MOVAPS	Move four aligned packed single-precision floating-point values between XMM registers or between and XMM register and memory.
MOVUPS	Move four unaligned packed single-precision floating-point values between XMM registers or between and XMM register and memory.
MOVHPS	Move two packed single-precision floating-point values to an from the high quadword of an XMM register and memory.
MOVHLPS	Move two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register.
MOVLPS	Move two packed single-precision floating-point values to an from the low quadword of an XMM register and memory.
MOVLHPS	Move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register.
MOVMSKPS	Extract sign mask from four packed single-precision floating-point values.
MOVSS	Move scalar single-precision floating-point value between XMM registers or between an XMM register and memory.

### 5.5.1.2 SSE Packed Arithmetic Instructions

SSE packed arithmetic instructions perform packed and scalar arithmetic operations on packed and scalar single-precision floating-point operands.

ADDPS	Add packed single-precision floating-point values.
ADDSS	Add scalar single-precision floating-point values.
SUBPS	Subtract packed single-precision floating-point values.
SUBSS	Subtract scalar single-precision floating-point values.
MULPS	Multiply packed single-precision floating-point values.
MULSS	Multiply scalar single-precision floating-point values.
DIVPS	Divide packed single-precision floating-point values.
DIVSS	Divide scalar single-precision floating-point values.

RCPSS	Compute reciprocals of packed single-precision floating-point values.
RCPSS	Compute reciprocal of scalar single-precision floating-point values.
SQRTPS	Compute square roots of packed single-precision floating-point values.
SQRTSS	Compute square root of scalar single-precision floating-point values.
RSQRTPS	Compute reciprocals of square roots of packed single-precision floating-point values.
RSQRTSS	Compute reciprocal of square root of scalar single-precision floating-point values.
MAXPS	Return maximum packed single-precision floating-point values.
MAXSS	Return maximum scalar single-precision floating-point values.
MINPS	Return minimum packed single-precision floating-point values.
MINSS	Return minimum scalar single-precision floating-point values.

### 5.5.1.3 SSE Comparison Instructions

SSE compare instructions compare packed and scalar single-precision floating-point operands.

CMPPS	Compare packed single-precision floating-point values.
CMPSS	Compare scalar single-precision floating-point values.
COMISS	Perform ordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register.
UCOMISS	Perform unordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register.

### 5.5.1.4 SSE Logical Instructions

SSE logical instructions perform bitwise AND, AND NOT, OR, and XOR operations on packed single-precision floating-point operands.

ANDPS	Perform bitwise logical AND of packed single-precision floating-point values.
ANDNPS	Perform bitwise logical AND NOT of packed single-precision floating-point values.
ORPS	Perform bitwise logical OR of packed single-precision floating-point values.
XORPS	Perform bitwise logical XOR of packed single-precision floating-point values.

### 5.5.1.5 SSE Shuffle and Unpack Instructions

SSE shuffle and unpack instructions shuffle or interleave single-precision floating-point values in packed single-precision floating-point operands.

SHUFPS	Shuffles values in packed single-precision floating-point operands.
UNPCKHPS	Unpacks and interleaves the two high-order values from two single-precision floating-point operands.
UNPCKLPS	Unpacks and interleaves the two low-order values from two single-precision floating-point operands.

### 5.5.1.6 SSE Conversion Instructions

SSE conversion instructions convert packed and individual doubleword integers into packed and scalar single-precision floating-point values and vice versa.

CVTPI2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTSS2PS	Convert doubleword integer to scalar single-precision floating-point value.
CVTSS2PI	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTSS2PI	Convert with truncation packed single-precision floating-point values to packed doubleword integers.
CVTSS2SI	Convert a scalar single-precision floating-point value to a doubleword integer.

**CVTTSS2SI** Convert with truncation a scalar single-precision floating-point value to a scalar double-word integer.

### 5.5.2 SSE MXCSR State Management Instructions

MXCSR state management instructions allow saving and restoring the state of the MXCSR control and status register.

**LDMXCSR** Load MXCSR register.  
**STMXCSR** Save MXCSR register state.

### 5.5.3 SSE 64-Bit SIMD Integer Instructions

These SSE 64-bit SIMD integer instructions perform additional operations on packed bytes, words, or doublewords contained in MMX registers. They represent enhancements to the MMX instruction set described in Section 5.4, “MMX™ Instructions.”

**PAVGB** Compute average of packed unsigned byte integers.  
**PAVGW** Compute average of packed unsigned word integers.  
**PEXTRW** Extract word.  
**PINSRW** Insert word.  
**PMAXUB** Maximum of packed unsigned byte integers.  
**PMAXSW** Maximum of packed signed word integers.  
**PMINUB** Minimum of packed unsigned byte integers.  
**PMINSW** Minimum of packed signed word integers.  
**PMOVBMSKB** Move byte mask.  
**PMULHUW** Multiply packed unsigned integers and store high result.  
**PSADBW** Compute sum of absolute differences.  
**PSHUFW** Shuffle packed integer word in MMX register.

### 5.5.4 SSE Cacheability Control, Prefetch, and Instruction Ordering Instructions

The cacheability control instructions provide control over the caching of non-temporal data when storing data from the MMX and XMM registers to memory. The **PREFETCHh** allows data to be prefetched to a selected cache level. The **SFENCE** instruction controls instruction ordering on store operations.

**MASKMOVQ** Non-temporal store of selected bytes from an MMX register into memory.  
**MOVNTQ** Non-temporal store of quadword from an MMX register into memory.  
**MOVNTPS** Non-temporal store of four packed single-precision floating-point values from an XMM register into memory.  
**PREFETCHh** Load 32 or more of bytes from memory to a selected level of the processor’s cache hierarchy  
**SFENCE** Serializes store operations.

## 5.6 SSE2 INSTRUCTIONS

SSE2 extensions represent an extension of the SIMD execution model introduced with MMX technology and the SSE extensions. SSE2 instructions operate on packed double-precision floating-point operands and on packed byte, word, doubleword, and quadword operands located in the XMM registers. For more detail on these instructions, see Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2).”

SSE2 instructions can only be executed on Intel 64 and IA-32 processors that support the SSE2 extensions. Support for these instructions can be detected with the **CPUID** instruction. See the description of the **CPUID** instruc-

tion in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

These instructions are divided into four subgroups (note that the first subgroup is further divided into subordinate subgroups):

- Packed and scalar double-precision floating-point instructions.
- Packed single-precision floating-point conversion instructions.
- 128-bit SIMD integer instructions.
- Cacheability-control and instruction ordering instructions.

The following sections give an overview of each subgroup.

## 5.6.1 SSE2 Packed and Scalar Double-Precision Floating-Point Instructions

SSE2 packed and scalar double-precision floating-point instructions are divided into the following subordinate subgroups: data movement, arithmetic, comparison, conversion, logical, and shuffle operations on double-precision floating-point operands. These are introduced in the sections that follow.

### 5.6.1.1 SSE2 Data Movement Instructions

SSE2 data movement instructions move double-precision floating-point data between XMM registers and between XMM registers and memory.

MOVAPD	Move two aligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
MOVUPD	Move two unaligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
MOVHPD	Move high packed double-precision floating-point value to an from the high quadword of an XMM register and memory.
MOVLPD	Move low packed single-precision floating-point value to an from the low quadword of an XMM register and memory.
MOVMSKPD	Extract sign mask from two packed double-precision floating-point values.
MOVSD	Move scalar double-precision floating-point value between XMM registers or between an XMM register and memory.

### 5.6.1.2 SSE2 Packed Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double-precision floating-point operands.

ADDPD	Add packed double-precision floating-point values.
ADDSD	Add scalar double precision floating-point values.
SUBPD	Subtract packed double-precision floating-point values.
SUBSD	Subtract scalar double-precision floating-point values.
MULPD	Multiply packed double-precision floating-point values.
MULSD	Multiply scalar double-precision floating-point values.
DIVPD	Divide packed double-precision floating-point values.
DIVSD	Divide scalar double-precision floating-point values.
SQRTPD	Compute packed square roots of packed double-precision floating-point values.
SQRTSD	Compute scalar square root of scalar double-precision floating-point values.
MAXPD	Return maximum packed double-precision floating-point values.
MAXSD	Return maximum scalar double-precision floating-point values.
MINPD	Return minimum packed double-precision floating-point values.

**MINSD** Return minimum scalar double-precision floating-point values.

### 5.6.1.3 SSE2 Logical Instructions

SSE2 logical instructions perform AND, AND NOT, OR, and XOR operations on packed double-precision floating-point values.

<b>ANDPD</b>	Perform bitwise logical AND of packed double-precision floating-point values.
<b>ANDNPD</b>	Perform bitwise logical AND NOT of packed double-precision floating-point values.
<b>ORPD</b>	Perform bitwise logical OR of packed double-precision floating-point values.
<b>XORPD</b>	Perform bitwise logical XOR of packed double-precision floating-point values.

### 5.6.1.4 SSE2 Compare Instructions

SSE2 compare instructions compare packed and scalar double-precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

<b>CMPPD</b>	Compare packed double-precision floating-point values.
<b>CMPSD</b>	Compare scalar double-precision floating-point values.
<b>COMISD</b>	Perform ordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.
<b>UCOMISD</b>	Perform unordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.

### 5.6.1.5 SSE2 Shuffle and Unpack Instructions

SSE2 shuffle and unpack instructions shuffle or interleave double-precision floating-point values in packed double-precision floating-point operands.

<b>SHUFPS</b>	Shuffles values in packed double-precision floating-point operands.
<b>UNPCKHPD</b>	Unpacks and interleaves the high values from two packed double-precision floating-point operands.
<b>UNPCKLPD</b>	Unpacks and interleaves the low values from two packed double-precision floating-point operands.

### 5.6.1.6 SSE2 Conversion Instructions

SSE2 conversion instructions convert packed and individual doubleword integers into packed and scalar double-precision floating-point values and vice versa. They also convert between packed and scalar single-precision and double-precision floating-point values.

<b>CVTPD2PI</b>	Convert packed double-precision floating-point values to packed doubleword integers.
<b>CVTTPD2PI</b>	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
<b>CVTPI2PD</b>	Convert packed doubleword integers to packed double-precision floating-point values.
<b>CVTPD2DQ</b>	Convert packed double-precision floating-point values to packed doubleword integers.
<b>CVTTPD2DQ</b>	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
<b>CVTDQ2PD</b>	Convert packed doubleword integers to packed double-precision floating-point values.
<b>CVTSP2PD</b>	Convert packed single-precision floating-point values to packed double-precision floating-point values.
<b>CVTPD2PS</b>	Convert packed double-precision floating-point values to packed single-precision floating-point values.
<b>CVTSS2SD</b>	Convert scalar single-precision floating-point values to scalar double-precision floating-point values.

CVTSD2SS	Convert scalar double-precision floating-point values to scalar single-precision floating-point values.
CVTSD2SI	Convert scalar double-precision floating-point values to a doubleword integer.
CVTTSD2SI	Convert with truncation scalar double-precision floating-point values to scalar doubleword integers.
CVTSI2SD	Convert doubleword integer to scalar double-precision floating-point value.

### 5.6.2 SSE2 Packed Single-Precision Floating-Point Instructions

SSE2 packed single-precision floating-point instructions perform conversion operations on single-precision floating-point and integer operands. These instructions represent enhancements to the SSE single-precision floating-point instructions.

CVTDQ2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTPS2DQ	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTPS2DQ	Convert with truncation packed single-precision floating-point values to packed doubleword integers.

### 5.6.3 SSE2 128-Bit SIMD Integer Instructions

SSE2 SIMD integer instructions perform additional operations on packed words, doublewords, and quadwords contained in XMM and MMX registers.

MOVDQA	Move aligned double quadword.
MOVDQU	Move unaligned double quadword.
MOVQ2DQ	Move quadword integer from MMX to XMM registers.
MOVDQ2Q	Move quadword integer from XMM to MMX registers.
PMULUDQ	Multiply packed unsigned doubleword integers.
PADDQ	Add packed quadword integers.
PSUBQ	Subtract packed quadword integers.
PSHUFLW	Shuffle packed low words.
PSHUFHW	Shuffle packed high words.
PSHUFDB	Shuffle packed doublewords.
PSLLDQ	Shift double quadword left logical.
PSRLDQ	Shift double quadword right logical.
PUNPCKHQDQ	Unpack high quadwords.
PUNPCKLQDQ	Unpack low quadwords.

### 5.6.4 SSE2 Cacheability Control and Ordering Instructions

SSE2 cacheability control instructions provide additional operations for caching of non-temporal data when storing data from XMM registers to memory. LFENCE and MFENCE provide additional control of instruction ordering on store operations.

CLFLUSH	See Section 5.1.13.
LFENCE	Serializes load operations.
MFENCE	Serializes load and store operations.
PAUSE	Improves the performance of “spin-wait loops”.
MASKMOVDQU	Non-temporal store of selected bytes from an XMM register into memory.
MOVNTPD	Non-temporal store of two packed double-precision floating-point values from an XMM register into memory.
MOVNTDQ	Non-temporal store of double quadword from an XMM register into memory.



**MOVNTI** Non-temporal store of a doubleword from a general-purpose register into memory.

## 5.7 SSE3 INSTRUCTIONS

The SSE3 extensions offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities. These instructions can be grouped into the following categories:

- One x87FPU instruction used in integer conversion.
- One SIMD integer instruction that addresses unaligned data loads.
- Two SIMD floating-point packed ADD/SUB instructions.
- Four SIMD floating-point horizontal ADD/SUB instructions.
- Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions.
- Two thread synchronization instructions.

SSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

The sections that follow describe each subgroup.

### 5.7.1 SSE3 x87-FP Integer Conversion Instruction

**FISTTP** Behaves like the FISTP instruction but uses truncation, irrespective of the rounding mode specified in the floating-point control word (FCW).

### 5.7.2 SSE3 Specialized 128-bit Unaligned Data Load Instruction

**LDDQU** Special 128-bit unaligned load designed to avoid cache line splits.

### 5.7.3 SSE3 SIMD Floating-Point Packed ADD/SUB Instructions

**ADDSUBPS** Performs single-precision addition on the second and fourth pairs of 32-bit data elements within the operands; single-precision subtraction on the first and third pairs.

**ADDSUBPD** Performs double-precision addition on the second pair of quadwords, and double-precision subtraction on the first pair.

### 5.7.4 SSE3 SIMD Floating-Point Horizontal ADD/SUB Instructions

**HADDPS** Performs a single-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.

**HSUBPS** Performs a single-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.



HADDPD	Performs a double-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.
HSUBPD	Performs a double-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

### 5.7.5 SSE3 SIMD Floating-Point LOAD/MOVE/DUPLICATE Instructions

MOVSHDUP	Loads/moves 128 bits; duplicating the second and fourth 32-bit data elements.
MOVSLDUP	Loads/moves 128 bits; duplicating the first and third 32-bit data elements.
MOVDDUP	Loads/moves 64 bits (bits[63:0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register; duplicates the 64 bits from the source.

### 5.7.6 SSE3 Agent Synchronization Instructions

MONITOR	Sets up an address range used to monitor write-back stores.
MWAIT	Enables a logical processor to enter into an optimized state while waiting for a write-back store to the address range set up by the MONITOR instruction.

## 5.8 SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (SSSE3) INSTRUCTIONS

SSSE3 provide 32 instructions (represented by 14 mnemonics) to accelerate computations on packed integers. These include:

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

SSSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

The sections that follow describe each subgroup.

### 5.8.1 Horizontal Addition/Subtraction

PHADDW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.
PHADDSW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
PHADDD	Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.
PHSUBW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the

	source and destination operands. The signed 16-bit results are packed and written to the destination operand.
PHSUBSW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.
PHSUBD	Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

## 5.8.2 Packed Absolute Values

PABSB	Computes the absolute value of each signed byte data element.
PABSW	Computes the absolute value of each signed 16-bit data element.
PABSD	Computes the absolute value of each signed 32-bit data element.

## 5.8.3 Multiply and Add Packed Signed and Unsigned Bytes

PMADDUBSW	Multiplies each unsigned byte value with the corresponding signed byte value to produce an intermediate, 16-bit signed integer. Each adjacent pair of 16-bit signed values are added horizontally. The signed, saturated 16-bit results are packed to the destination operand.
-----------	--

## 5.8.4 Packed Multiply High with Round and Scale

PMULHRWSW	Multiplies vertically each signed 16-bit integer from the destination operand with the corresponding signed 16-bit integer of the source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.
-----------	--

## 5.8.5 Packed Shuffle Bytes

PSHUFB	Permutates each byte in place, according to a shuffle control mask. The least significant three or four bits of each shuffle control byte of the control mask form the shuffle index. The shuffle mask is unaffected. If the most significant bit (bit 7) of a shuffle control byte is set, the constant zero is written in the result byte.
--------	--

## 5.8.6 Packed Sign

PSIGNB/W/D	Negates each signed integer element of the destination operand if the sign of the corresponding data element in the source operand is less than zero.
------------	---

## 5.8.7 Packed Align Right

PALIGNR	Source operand is appended after the destination operand forming an intermediate value of twice the width of an operand. The result is extracted from the intermediate value into the destination operand by selecting the 128 bit or 64 bit value that are right-aligned to the byte offset specified by the immediate value.
---------	--

## 5.9 SSE4 INSTRUCTIONS

Intel® Streaming SIMD Extensions 4 (SSE4) introduces 54 new instructions. 47 of the SSE4 instructions are referred to as SSE4.1 in this document, 7 new SSE4 instructions are referred to as SSE4.2.

SSE4.1 is targeted to improve the performance of media, imaging, and 3D workloads. SSE4.1 adds instructions that improve compiler vectorization and significantly increase support for packed dword computation. The technology also provides a hint that can improve memory throughput when reading from uncacheable WC memory type.

The 47 SSE4.1 instructions include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction performs a load with a streaming hint.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

The SSE4.2 instructions operating on XMM registers include:

- String and text processing that can take advantage of single-instruction multiple-data programming techniques.
- A SIMD integer instruction that enhances the capability of the 128-bit integer SIMD capability in SSE4.1.

## 5.10 SSE4.1 INSTRUCTIONS

SSE4.1 instructions can use an XMM register as a source or destination. Programming SSE4.1 is similar to programming 128-bit Integer SIMD and floating-point SIMD instructions in SSE/SSE2/SSE3/SSSE3. SSE4.1 does not provide any 64-bit integer SIMD instructions operating on MMX registers. The sections that follow describe each subgroup.

### 5.10.1 Dword Multiply Instructions

PMULLD	Returns four lower 32-bits of the 64-bit results of signed 32-bit integer multiplies.
PMULDQ	Returns two 64-bit signed result of signed 32-bit integer multiplies.

### 5.10.2 Floating-Point Dot Product Instructions

DPPD	Perform double-precision dot product for up to 2 elements and broadcast.
DPPS	Perform single-precision dot products for up to 4 elements and broadcast.

### 5.10.3 Streaming Load Hint Instruction

MOVNTDQA	Provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region (a streaming line) to be fetched and held in a small set of temporary buffers
----------	---

(“streaming load buffers”). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be supplied from the streaming load buffer and can improve throughput.

#### 5.10.4 Packed Blending Instructions

BLENDPD	Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDPS	Conditionally copies specified single-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDVDP	Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
BLENDVPS	Conditionally copies specified single-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
PBLENDVB	Conditionally copies specified byte elements in the source operand to the corresponding elements in the destination, using an implied mask.
PBLENDW	Conditionally copies specified word elements in the source operand to the corresponding elements in the destination, using an immediate byte control.

#### 5.10.5 Packed Integer MIN/MAX Instructions

PMINUW	Compare packed unsigned word integers.
PMINUD	Compare packed unsigned dword integers.
PMINSB	Compare packed signed byte integers.
PMINSD	Compare packed signed dword integers.
PMAXUW	Compare packed unsigned word integers.
PMAXUD	Compare packed unsigned dword integers.
PMASB	Compare packed signed byte integers.
PMASD	Compare packed signed dword integers.

#### 5.10.6 Floating-Point Round Instructions with Selectable Rounding Mode

ROUNDPS	Round packed single precision floating-point values into integer values and return rounded floating-point values.
ROUNDPD	Round packed double precision floating-point values into integer values and return rounded floating-point values.
ROUNDSS	Round the low packed single precision floating-point value into an integer value and return a rounded floating-point value.
ROUNDSD	Round the low packed double precision floating-point value into an integer value and return a rounded floating-point value.

#### 5.10.7 Insertion and Extractions from XMM Registers

EXTRACTPS	Extracts a single-precision floating-point value from a specified offset in an XMM register and stores the result to memory or a general-purpose register.
INSERTPS	Inserts a single-precision floating-point value from either a 32-bit memory location or selected from a specified offset in an XMM register to a specified offset in the destination XMM register. In addition, INSERTPS allows zeroing out selected data elements in the destination, using a mask.

PINSRB	Insert a byte value from a register or memory into an XMM register.
PINSRD	Insert a dword value from 32-bit register or memory into an XMM register.
PINSRQ	Insert a qword value from 64-bit register or memory into an XMM register.
PEXTRB	Extract a byte from an XMM register and insert the value into a general-purpose register or memory.
PEXTRW	Extract a word from an XMM register and insert the value into a general-purpose register or memory.
PEXTRD	Extract a dword from an XMM register and insert the value into a general-purpose register or memory.
PEXTRQ	Extract a qword from an XMM register and insert the value into a general-purpose register or memory.

### 5.10.8 Packed Integer Format Conversions

PMOVSXBW	Sign extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVZXBW	Zero extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVSXBD	Sign extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVZXBBD	Zero extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVSXWD	Sign extend the lower 16-bit integer of each packed dword element into packed signed dword integers.
PMOVZXWD	Zero extend the lower 16-bit integer of each packed dword element into packed signed dword integers.
PMOVSXBQ	Sign extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVZXBQ	Zero extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVSXWQ	Sign extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVZXWQ	Zero extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVSXDQ	Sign extend the lower 32-bit integer of each packed qword element into packed signed qword integers.
PMOVZXDQ	Zero extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

### 5.10.9 Improved Sums of Absolute Differences (SAD) for 4-Byte Blocks

MPSADBW	Performs eight 4-byte wide Sum of Absolute Differences operations to produce eight word integers.
---------	---

### 5.10.10 Horizontal Search

PHMINPOSUW	Finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words. The resulting value and location (offset within the source) are packed into the low dword of the destination XMM register.
------------	--

### 5.10.11 Packed Test

**PTEST** Performs a logical AND between the destination with this mask and sets the ZF flag if the result is zero. The CF flag (zero for TEST) is set if the inverted mask AND'd with the destination is all zeroes.

### 5.10.12 Packed Qword Equality Comparisons

**PCMPEQQ** 128-bit packed qword equality test.

### 5.10.13 Dword Packing With Unsigned Saturation

**PACKUSDW** PACKUSDW packs dword to word with unsigned saturation.

## 5.11 SSE4.2 INSTRUCTION SET

Five of the SSE4.2 instructions operate on XMM register as a source or destination. These include four text/string processing instructions and one packed quadword compare SIMD instruction. Programming these five SSE4.2 instructions is similar to programming 128-bit Integer SIMD in SSE2/SSSE3. SSE4.2 does not provide any 64-bit integer SIMD instructions.

CRC32 operates on general-purpose registers and is summarized in Section 5.1.6. The sections that follow summarize each subgroup.

### 5.11.1 String and Text Processing Instructions

**PCMPESTRI** Packed compare explicit-length strings, return index in ECX/RCX.  
**PCMPESTRM** Packed compare explicit-length strings, return mask in XMM0.  
**PCMPISTRI** Packed compare implicit-length strings, return index in ECX/RCX.  
**PCMPISTRM** Packed compare implicit-length strings, return mask in XMM0.

### 5.11.2 Packed Comparison SIMD integer Instruction

**PCMPGTQ** Performs logical compare of greater-than on packed integer quadwords.

## 5.12 INTEL® AES-NI AND PCLMULQDQ

Six Intel® AES-NI instructions operate on XMM registers to provide accelerated primitives for block encryption/decryption using Advanced Encryption Standard (FIPS-197). The PCLMULQDQ instruction performs carry-less multiplication for two binary numbers up to 64-bit wide.

**AESDEC** Perform an AES decryption round using an 128-bit state and a round key.  
**AESDECLAST** Perform the last AES decryption round using an 128-bit state and a round key.  
**AESENC** Perform an AES encryption round using an 128-bit state and a round key.  
**AESENCLAST** Perform the last AES encryption round using an 128-bit state and a round key.  
**AESIMC** Perform an inverse mix column transformation primitive.  
**AESKEYGENASSIST** Assist the creation of round keys with a key expansion schedule.  
**PCLMULQDQ** Perform carryless multiplication of two 64-bit numbers.

## 5.13 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel® Advanced Vector Extensions (AVX) promotes legacy 128-bit SIMD instruction sets that operate on XMM register set to use a “vector extension” (VEX) prefix and operates on 256-bit vector registers (YMM). Almost all prior generations of 128-bit SIMD instructions that operates on XMM (but not on MMX registers) are promoted to support three-operand syntax with VEX-128 encoding.

VEX-prefix encoded AVX instructions support 256-bit and 128-bit floating-point operations by extending the legacy 128-bit SIMD floating-point instructions to support three-operand syntax.

Additional functional enhancements are also provided with VEX-encoded AVX instructions.

The list of AVX instructions are listed in the following tables:

- Table 14-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-3 lists 256-bit and 128-bit data movement and processing instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-4 lists functional enhancements of 256-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-5 lists 128-bit integer and floating-point instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-6 lists functional enhancements of 128-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-7 lists 128-bit data movement and processing instructions promoted from legacy instruction sets.

## 5.14 16-BIT FLOATING-POINT CONVERSION

Conversion between single-precision floating-point (32-bit) and half-precision FP (16-bit) data are provided by VCVTSP2PH, VCVTPH2PS:

VCVTPH2PS	Convert eight/four data element containing 16-bit floating-point data into eight/four single-precision floating-point data.
VCVTSP2PH	Convert eight/four data element containing single-precision floating-point data into eight/four 16-bit floating-point data.

## 5.15 FUSED-MULTIPLY-ADD (FMA)

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

- Table 14-15 lists FMA instruction sets.

## 5.16 INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

- Table 14-18 lists promoted vector integer instructions in AVX2.
- Table 14-19 lists new instructions in AVX2 that complements AVX.



## 5.17 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

XABORT	Abort an RTM transaction execution.
XACQUIRE	Prefix hint to the beginning of an HLE transaction region.
XRELEASE	Prefix hint to the end of an HLE transaction region.
XBEGIN	Transaction begin of an RTM transaction region.
XEND	Transaction end of an RTM transaction region.
XTEST	Test if executing in a transactional region.

## 5.18 INTEL® SHA EXTENSIONS

Intel® SHA extensions provide a set of instructions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants.

SHA1MSG1	Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
SHA1MSG2	Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
SHA1NEXTE	Calculate SHA1 state E after four rounds.
SHA1RND54	Perform four rounds of SHA1 operations.
SHA256MSG1	Perform an intermediate calculation for the next four SHA256 message dwords.
SHA256MSG2	Perform the final calculation for the next four SHA256 message dwords.
SHA256RND52	Perform two rounds of SHA256 operations.

## 5.19 INTEL® ADVANCED VECTOR EXTENSIONS 512 (INTEL® AVX-512)

The Intel® AVX-512 family comprises a collection of 512-bit SIMD instruction sets to accelerate a diverse range of applications. Intel AVX-512 instructions provide a wide range of functionality that support programming in 512-bit, 256 and 128-bit vector register, plus support for opmask registers and instructions operating on opmask registers.

The collection of 512-bit SIMD instruction sets in Intel AVX-512 include new functionality not available in Intel AVX and Intel AVX2, and promoted instructions similar to equivalent ones in Intel AVX / Intel AVX2 but with enhancement provided by opmask registers not available to VEX-encoded Intel AVX / Intel AVX2. Some instruction mnemonics in AVX / AVX2 that are promoted into AVX-512 can be replaced by new instruction mnemonics that are available only with EVEX encoding, e.g., VBROADCASTF128 into VBROADCASTF32X4. Details of EVEX instruction encoding are discussed in Section 2.6, “Intel® AVX-512 Encoding” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

512-bit instruction mnemonics in AVX-512F that are not AVX/AVX2 promotions include:

VALIGND/Q	Perform dword/qword alignment of two concatenated source vectors.
VBLENDMPD/PS	Replace the VBLENDVPD/PS instructions (using opmask as select control).
VCOMPRESSPD/PS	Compress packed DP or SP elements of a vector.
VCVT(T)PD2UDQ	Convert packed DP FP elements of a vector to packed unsigned 32-bit integers.
VCVT(T)PS2UDQ	Convert packed SP FP elements of a vector to packed unsigned 32-bit integers.
VCVTQQ2PD/PS	Convert packed signed 64-bit integers to packed DP/SP FP elements.
VCVT(T)SD2USI	Convert the low DP FP element of a vector to an unsigned integer.
VCVT(T)SS2USI	Convert the low SP FP element of a vector to an unsigned integer.
VCVTUDQ2PD/PS	Convert packed unsigned 32-bit integers to packed DP/SP FP elements.
VCVTUSI2USD/S	Convert an unsigned integer to the low DP/SP FP element and merge to a vector.
VEXPANDPD/PS	Expand packed DP or SP elements of a vector.
VEXTRACTF32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.



VEXTRACTI32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VFIXUPIMMPD/PS	Perform fix-up to special values in DP/SP FP vectors.
VFIXUPIMMSD/SS	Perform fix-up to special values of the low DP/SP FP element.
VGETEXPPD/PS	Convert the exponent of DP/SP FP elements of a vector into FP values.
VGETEXPSD/SS	Convert the exponent of the low DP/SP FP element in a vector into FP value.
VGETMANTPD/PS	Convert the mantissa of DP/SP FP elements of a vector into FP values.
VGETMANTSD/SS	Convert the mantissa of the low DP/SP FP element of a vector into FP value.
VINSERTF32X4/64X4	Insert a 128/256-bit vector into a full-length vector with 32/64-bit granular update.
VMOVDQA32/64	VMOVDQA with 32/64-bit granular conditional update.
VMOVDQU32/64	VMOVDQU with 32/64-bit granular conditional update.
VPBLENDMD/Q	Blend dword/qword elements using opmask as select control.
VPBROADCAST/Q	Broadcast from general-purpose register to vector register.
VPCMPD/UD	Compare packed signed/unsigned dwords using specified primitive.
VPCMPQ/UQ	Compare packed signed/unsigned quadwords using specified primitive.
VPCOMPRESSQ/D	Compress packed 64/32-bit elements of a vector.
VPERMI2D/Q	Full permute of two tables of dword/qword elements overwriting the index vector.
VPERMI2PD/PS	Full permute of two tables of DP/SP elements overwriting the index vector.
VPERMT2D/Q	Full permute of two tables of dword/qword elements overwriting one source table.
VPERMT2PD/PS	Full permute of two tables of DP/SP elements overwriting one source table.
VEXPANDD/Q	Expand packed dword/qword elements of a vector.
VPMAXSQ	Compute maximum of packed signed 64-bit integer elements.
VPMAXUD/UQ	Compute maximum of packed unsigned 32/64-bit integer elements.
VPMINSQ	Compute minimum of packed signed 64-bit integer elements.
VPMINUD/UQ	Compute minimum of packed unsigned 32/64-bit integer elements.
VPMOV(S US)QB	Down convert qword elements in a vector to byte elements using truncation (saturation   unsigned saturation).
VPMOV(S US)QW	Down convert qword elements in a vector to word elements using truncation (saturation   unsigned saturation).
VPMOV(S US)QD	Down convert qword elements in a vector to dword elements using truncation (saturation   unsigned saturation).
VPMOV(S US)DB	Down convert dword elements in a vector to byte elements using truncation (saturation   unsigned saturation).
VPMOV(S US)DW	Down convert dword elements in a vector to word elements using truncation (saturation   unsigned saturation).
VPROLD/Q	Rotate dword/qword element left by a constant shift count with conditional update.
VPROLVD/Q	Rotate dword/qword element left by shift counts specified in a vector with conditional update.
VPRORD/Q	Rotate dword/qword element right by a constant shift count with conditional update.
VPRORRD/Q	Rotate dword/qword element right by shift counts specified in a vector with conditional update.
VPSCATTERDD/DQ	Scatter dword/qword elements in a vector to memory using dword indices.
VPSCATTERQD/QQ	Scatter dword/qword elements in a vector to memory using qword indices.
VPSRAQ	Shift qwords right by a constant shift count and shifting in sign bits.
VPSRAVQ	Shift qwords right by shift counts in a vector and shifting in sign bits.
VPTESTNMD/Q	Perform bitwise NAND of dword/qword elements of two vectors and write results to opmask.
VPTERLOGD/Q	Perform bitwise ternary logic operation of three vectors with 32/64 bit granular conditional update.

VPTSTMD/Q	Perform bitwise AND of dword/qword elements of two vectors and write results to opmask.
VRCP14PD/PS	Compute approximate reciprocals of packed DP/SP FP elements of a vector.
VRCP14SD/SS	Compute the approximate reciprocal of the low DP/SP FP element of a vector.
VRNDSCALEPD/PS	Round packed DP/SP FP elements of a vector to specified number of fraction bits.
VRNDSCALESD/SS	Round the low DP/SP FP element of a vector to specified number of fraction bits.
VRSQRT14PD/PS	Compute approximate reciprocals of square roots of packed DP/SP FP elements of a vector.
VRSQRT14SD/SS	Compute the approximate reciprocal of square root of the low DP/SP FP element of a vector.
VSCALEPD/PS	Multiply packed DP/SP FP elements of a vector by powers of two with exponents specified in a second vector.
VSCALESD/SS	Multiply the low DP/SP FP element of a vector by powers of two with exponent specified in the corresponding element of a second vector.
VSCATTERDD/DQ	Scatter SP/DP FP elements in a vector to memory using dword indices.
VSCATTERQD/QQ	Scatter SP/DP FP elements in a vector to memory using qword indices.
VSHUFF32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.
VSHUFI32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.

512-bit instruction mnemonics in AVX-512DQ that are not AVX/AVX2 promotions include:

VCVT(T)PD2QQ	Convert packed DP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PD2UQQ	Convert packed DP FP elements of a vector to packed unsigned 64-bit integers.
VCVT(T)PS2QQ	Convert packed SP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PS2UQQ	Convert packed SP FP elements of a vector to packed unsigned 64-bit integers.
VCVTUQQ2PD/PS	Convert packed unsigned 64-bit integers to packed DP/SP FP elements.
VEXTRACTF64X2	Extract a vector from a full-length vector with 64-bit granular update.
VEXTRACTI64X2	Extract a vector from a full-length vector with 64-bit granular update.
VFPCLASSPD/PS	Test packed DP/SP FP elements in a vector by numeric/special-value category.
VFPCLASSSD/SS	Test the low DP/SP FP element by numeric/special-value category.
VINSERTF64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VINSERTI64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VPMOVM2D/Q	Convert opmask register to vector register in 32/64-bit granularity.
VPMOVB2D/Q2M	Convert a vector register in 32/64-bit granularity to an opmask register.
VPMULLQ	Multiply packed signed 64-bit integer elements of two vectors and store low 64-bit signed result.
VRANGEPD/PS	Perform RANGE operation on each pair of DP/SP FP elements of two vectors using specified range primitive in imm8.
VRANGESD/SS	Perform RANGE operation on the pair of low DP/SP FP element of two vectors using specified range primitive in imm8.
VREDUCEPD/PS	Perform Reduction operation on packed DP/SP FP elements of a vector using specified reduction primitive in imm8.
VREDUCESD/SS	Perform Reduction operation on the low DP/SP FP element of a vector using specified reduction primitive in imm8.

512-bit instruction mnemonics in AVX-512BW that are not AVX/AVX2 promotions include:

VDBPSADBW	Double block packed Sum-Absolute-Differences on unsigned bytes.
VMOVDQU8/16	VMOVDQU with 8/16-bit granular conditional update.
VPBLENDMB	Replaces the VPBLENDVB instruction (using opmask as select control).
VPBLENDMW	Blend word elements using opmask as select control.
VPBROADCASTB/W	Broadcast from general-purpose register to vector register.

VPCMPB/UB	Compare packed signed/unsigned bytes using specified primitive.
VPCMPW/UW	Compare packed signed/unsigned words using specified primitive.
VPERMW	Permute packed word elements.
VPERMI2B/W	Full permute from two tables of byte/word elements overwriting the index vector.
VPMOVM2B/W	Convert opmask register to vector register in 8/16-bit granularity.
VPMOVB2M/W2M	Convert a vector register in 8/16-bit granularity to an opmask register.
VPMOV(S US)WB	Down convert word elements in a vector to byte elements using truncation (saturation   unsigned saturation).
VPSLLVW	Shift word elements in a vector left by shift counts in a vector.
VPSRAVW	Shift words right by shift counts in a vector and shifting in sign bits.
VPSRLVW	Shift word elements in a vector right by shift counts in a vector.
VPTESTNMB/W	Perform bitwise NAND of byte/word elements of two vectors and write results to opmask.
VPTESTMB/W	Perform bitwise AND of byte/word elements of two vectors and write results to opmask.

512-bit instruction mnemonics in AVX-512CD that are not AVX/AVX2 promotions include:

VPBROADCASTM	Broadcast from opmask register to vector register.
VPCONFLICTD/Q	Detect conflicts within a vector of packed 32/64-bit integers.
VPLZCNTD/Q	Count the number of leading zero bits of packed dword/qword elements.

Opmask instructions include:

KADDB/W/D/Q	Add two 8/16/32/64-bit opmasks.
KANDB/W/D/Q	Logical AND two 8/16/32/64-bit opmasks.
KANDNB/W/D/Q	Logical AND NOT two 8/16/32/64-bit opmasks.
KMOVB/W/D/Q	Move from or move to opmask register of 8/16/32/64-bit data.
KNOTB/W/D/Q	Bitwise NOT of two 8/16/32/64-bit opmasks.
KORB/W/D/Q	Logical OR two 8/16/32/64-bit opmasks.
KORTESTB/W/D/Q	Update EFLAGS according to the result of bitwise OR of two 8/16/32/64-bit opmasks.
KSHIFTLB/W/D/Q	Shift left 8/16/32/64-bit opmask by specified count.
KSHIFTRB/W/D/Q	Shift right 8/16/32/64-bit opmask by specified count.
KTESTB/W/D/Q	Update EFLAGS according to the result of bitwise TEST of two 8/16/32/64-bit opmasks.
KUNPCKBW/WD/DQ	Unpack and interleave two 8/16/32-bit opmasks into 16/32/64-bit mask.
KXNORB/W/D/Q	Bitwise logical XNOR of two 8/16/32/64-bit opmasks.
KXORB/W/D/Q	Logical XOR of two 8/16/32/64-bit opmasks.

512-bit instruction mnemonics in AVX-512ER include:

VEXP2PD/PS	Compute approximate base-2 exponential of packed DP/SP FP elements of a vector.
VEXP2SD/SS	Compute approximate base-2 exponential of the low DP/SP FP element of a vector.
VRCP28PD/PS	Compute approximate reciprocals to 28 bits of packed DP/SP FP elements of a vector.
VRCP28SD/SS	Compute the approximate reciprocal to 28 bits of the low DP/SP FP element of a vector.
VRSQRT28PD/PS	Compute approximate reciprocals of square roots to 28 bits of packed DP/SP FP elements of a vector.
VRSQRT28SD/SS	Compute the approximate reciprocal of square root to 28 bits of the low DP/SP FP element of a vector.

512-bit instruction mnemonics in AVX-512PF include:

VGATHERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using dword indices.
------------------	---

VGATHERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using qword indices.
VGATHERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using dword indices.
VGATHERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using qword indices.
VSCATTERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using dword indices.
VSCATTERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using qword indices.
VSCATTERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using dword indices.
VSCATTERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using qword indices.

## 5.20 SYSTEM INSTRUCTIONS

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

CLAC	Clear AC Flag in EFLAGS register.
STAC	Set AC Flag in EFLAGS register.
LGDT	Load global descriptor table (GDT) register.
SGDT	Store global descriptor table (GDT) register.
LLDT	Load local descriptor table (LDT) register.
SLDT	Store local descriptor table (LDT) register.
LTR	Load task register.
STR	Store task register.
LIDT	Load interrupt descriptor table (IDT) register.
SIDT	Store interrupt descriptor table (IDT) register.
MOV	Load and store control registers.
LMSW	Load machine status word.
SMSW	Store machine status word.
CLTS	Clear the task-switched flag.
ARPL	Adjust requested privilege level.
LAR	Load access rights.
LSL	Load segment limit.
VERR	Verify segment for reading
VERW	Verify segment for writing.
MOV	Load and store debug registers.
INVD	Invalidate cache, no writeback.
WBINVD	Invalidate cache, with writeback.
INVLPG	Invalidate TLB Entry.
INVPID	Invalidate Process-Context Identifier.
LOCK (prefix)	Perform atomic access to memory (can be applied to a number of general purpose instructions that provide memory source/destination access).
HLT	Halt processor.
RSM	Return from system management mode (SMM).
RDMSR	Read model-specific register.
WRMSR	Write model-specific register.
RDPMSR	Read performance monitoring counters.
RDTSC	Read time stamp counter.
RDTSCP	Read time stamp counter and processor ID.
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0.

SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XSAVES	Save processor supervisor-mode extended states to memory.
XRSTOR	Restore processor extended states from memory.
XRSTORS	Restore processor supervisor-mode extended states from memory.
XGETBV	Reads the state of an extended control register.
XSETBV	Writes the state of an extended control register.
RDFSBASE	Reads from FS base address at any privilege level.
RDGSBASE	Reads from GS base address at any privilege level.
WRFSBASE	Writes to FS base address at any privilege level.
WRGSBASE	Writes to GS base address at any privilege level.

## 5.21 64-BIT MODE INSTRUCTIONS

The following instructions are introduced in 64-bit mode. This mode is a sub-mode of IA-32e mode.

CDQE	Convert doubleword to quadword.
CMPSQ	Compare string operands.
CMPXCHG16B	Compare RDX:RAX with m128.
LODSQ	Load qword at address (R)SI into RAX.
MOVSQ	Move qword from address (R)SI to (R)DI.
MOVZX (64-bits)	Move bytes/words to doublewords/quadwords, zero-extension.
STOSQ	Store RAX at address RDI.
SWAPGS	Exchanges current GS base register value with value in MSR address C0000102H.
SYSCALL	Fast call to privilege level 0 system procedures.
SYSRET	Return from fast systemcall.

## 5.22 VIRTUAL-MACHINE EXTENSIONS

The behavior of the VMCS-maintenance instructions is summarized below:

VMPTRLD	Takes a single 64-bit source operand in memory. It makes the referenced VMCS active and current.
VMPTRST	Takes a single 64-bit destination operand that is in memory. Current-VMCS pointer is stored into the destination operand.
VMCLEAR	Takes a single 64-bit operand in memory. The instruction sets the launch state of the VMCS referenced by the operand to "clear", renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region.
VMREAD	Reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand.
VMWRITE	Writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand.

The behavior of the VMX management instructions is summarized below:

VMLAUNCH	Launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMRESUME	Resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.

VMXOFF	Causes the processor to leave VMX operation.
VMXON	Takes a single 64-bit source operand in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

The behavior of the VMX-specific TLB-management instructions is summarized below:

INVEPT	Invalidate cached <b>Extended Page Table</b> (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.
INVVPID	Invalidate cached mappings of address translation based on the <b>Virtual Processor ID</b> (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:

VMCALL	Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
VMFUNC	This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

## 5.23 SAFER MODE EXTENSIONS

The behavior of the GETSEC instruction leaves of the Safer Mode Extensions (SMX) are summarized below:

GETSEC[CAPABILITIES]	Returns the available leaf functions of the GETSEC instruction.
GETSEC[ENTERACCS]	Loads an authenticated code chipset module and enters authenticated code execution mode.
GETSEC[EXITAC]	Exits authenticated code execution mode.
GETSEC[SENDER]	Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.
GETSEC[SEXIT]	Exits the MLE.
GETSEC[PARAMETERS]	Returns SMX related parameter information.
GETSEC[SMCTRL]	SMX mode control.
GETSEC[WAKEUP]	Wakes up sleeping logical processors inside an MLE.

## 5.24 INTEL® MEMORY PROTECTION EXTENSIONS

Intel Memory Protection Extensions (MPX) provides a set of instructions to enable software to add robust bounds checking capability to memory references. Details of Intel MPX are described in Chapter 17, “Intel® MPX”.

BNDMK	Create a LowerBound and a UpperBound in a register.
BNDCL	Check the address of a memory reference against a LowerBound.
BNDUC	Check the address of a memory reference against an UpperBound in 1’s compliment form.
BNDCN	Check the address of a memory reference against an UpperBound not in 1’s compliment form.
BNDMOV	Copy or load from memory of the LowerBound and UpperBound to a register.
BNDMOV	Store to memory of the LowerBound and UpperBound from a register.
BNDLDX	Load bounds using address translation.
BNDSTX	Store bounds using address translation.

## 5.25 INTEL® SOFTWARE GUARD EXTENSIONS

Intel Software Guard Extensions (Intel SGX) provide two sets of instruction leaf functions to enable application software to instantiate a protected container, referred to as an enclave. The enclave instructions are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Details of Intel SGX are described in CHAPTER 32 through CHAPTER 38 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D*.

The first implementation of Intel SGX is also referred to as SGX1, it is introduced with the 6th Generation Intel Core Processors. The leaf functions supported in SGX1 is shown in Table 5-3.

**Table 5-3. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1**

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add a page	ENCLU[EENTER]	Enter an Enclave
ENCLS[EBLOCK]	Block an EPC page	ENCLU[EEXIT]	Exit an Enclave
ENCLS[ECREATE]	Create an enclave	ENCLU[EGETKEY]	Create a cryptographic key
ENCLS[EDBGDR]	Read data by debugger	ENCLU[EREPORT]	Create a cryptographic report
ENCLS[EDBGWR]	Write data by debugger	ENCLU[ERESUME]	Re-enter an Enclave
ENCLS[EEXTEND]	Extend EPC page measurement		
ENCLS[EINIT]	Initialize an enclave		
ENCLS[ELDB]	Load an EPC page as blocked		
ENCLS[ELDU]	Load an EPC page as unblocked		
ENCLS[EPA]	Add version array		
ENCLS[EREMOVE]	Remove a page from EPC		
ENCLS[ETRACK]	Activate EBLOCK checks		
ENCLS[EWB]	Write back/invalidate an EPC page		

## 5.26 SHADOW STACK MANAGEMENT INSTRUCTIONS

Shadow stack management instructions allow the program and run-time to perform operations like recovering from control protection faults, shadow stack switching, etc. The following instructions are provided.

CLRSSBSY	Clear busy bit in a supervisor shadow stack token.
INCSSP	Increment the shadow stack pointer (SSP).
RDSSP	Read shadow stack point (SSP).
RSTORSSP	Restore a shadow stack pointer (SSP).
SAVEPREVSSP	Save previous shadow stack pointer (SSP).
SETSSBSY	Set busy bit in a supervisor shadow stack token.
WRSS	Write to a shadow stack.
WRUSS	Write to a user mode shadow stack.

## 5.27 CONTROL TRANSFER TERMINATING INSTRUCTIONS

ENDBR32	Terminate an Indirect Branch in 32-bit and Compatibility Mode.
ENDBR64	Terminate an Indirect Branch in 64-bit Mode.





### 3. Updates to Chapter 8, Volume 1

Change bars and green text show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----  
Changes to this chapter: Removed references to Appendix D, which has been removed from the manual. The appendix removed is Appendix D, "Guidelines for Writing x87 FPU Exception Handlers".

The x87 Floating-Point Unit (FPU) provides high-performance floating-point processing capabilities for use in graphics processing, scientific, engineering, and business applications. It supports the floating-point, integer, and packed BCD integer data types and the floating-point processing algorithms and exception handling architecture defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic.

This chapter describes the x87 FPU's execution environment and instruction set. It also provides exception handling information that is specific to the x87 FPU. Refer to the following chapters or sections of chapters for additional information about x87 FPU instructions and floating-point operations:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B*, provide detailed descriptions of x87 FPU instructions.
- Section 4.2.2, "Floating-Point Data Types," Section 4.2.1.2, "Signed Integers," and Section 4.7, "BCD and Packed BCD Integers," describe the floating-point, integer, and BCD data types.
- Section 4.9, "Overview of Floating-Point Exceptions," Section 4.9.1, "Floating-Point Exception Conditions," and Section 4.9.2, "Floating-Point Exception Priority," give an overview of the floating-point exceptions that the x87 FPU can detect and report.

## 8.1 X87 FPU EXECUTION ENVIRONMENT

The x87 FPU represents a separate execution environment within the IA-32 architecture (see Figure 8-1). This execution environment consists of eight data registers (called the x87 FPU data registers) and the following special-purpose registers:

- Status register
- Control register
- Tag word register
- Last instruction pointer register
- Last data (operand) pointer register
- Opcode register

These registers are described in the following sections.

The x87 FPU executes instructions from the processor's normal instruction stream. The state of the x87 FPU is independent from the state of the basic execution environment and from the state of SSE/SSE2/SSE3 extensions.

However, the x87 FPU and Intel MMX technology share state because the MMX registers are aliased to the x87 FPU data registers. Therefore, when writing code that uses x87 FPU and MMX instructions, the programmer must explicitly manage the x87 FPU and MMX state (see Section 9.5, "Compatibility with x87 FPU Architecture").

### 8.1.1 x87 FPU in 64-Bit Mode and Compatibility Mode

In compatibility mode and 64-bit mode, x87 FPU instructions function like they do in protected mode. Memory operands are specified using the ModR/M, SIB encoding that is described in Section 3.7.5, "Specifying an Offset."

### 8.1.2 x87 FPU Data Registers

The x87 FPU data registers (shown in Figure 8-1) consist of eight 80-bit registers. Values are stored in these registers in the double extended-precision floating-point format shown in Figure 4-3. When floating-point, integer, or packed BCD integer values are loaded from memory into any of the x87 FPU data registers, the values are automatically converted into double extended-precision floating-point format (if they are not already in that format). When computation results are subsequently transferred back into memory from any of the x87 FPU registers, the

results can be left in the double extended-precision floating-point format or converted back into a shorter floating-point format, an integer format, or the packed BCD integer format. (See Section 8.2, "x87 FPU Data Types," for a description of the data types operated on by the x87 FPU.)

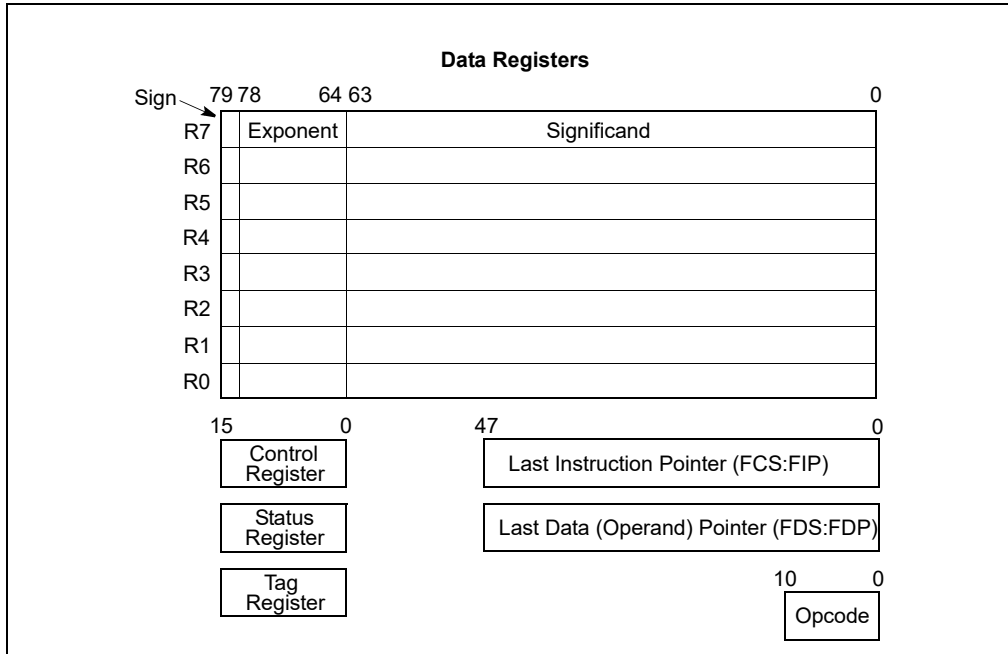


Figure 8-1. x87 FPU Execution Environment

The x87 FPU instructions treat the eight x87 FPU data registers as a register stack (see Figure 8-2). All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP (stack TOP) field in the x87 FPU status word. Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory and then increment TOP by one. (For the x87 FPU, a load operation is equivalent to a push and a store operation is equivalent to a pop.) Note that load and store operations are also available that do not push and pop the stack.

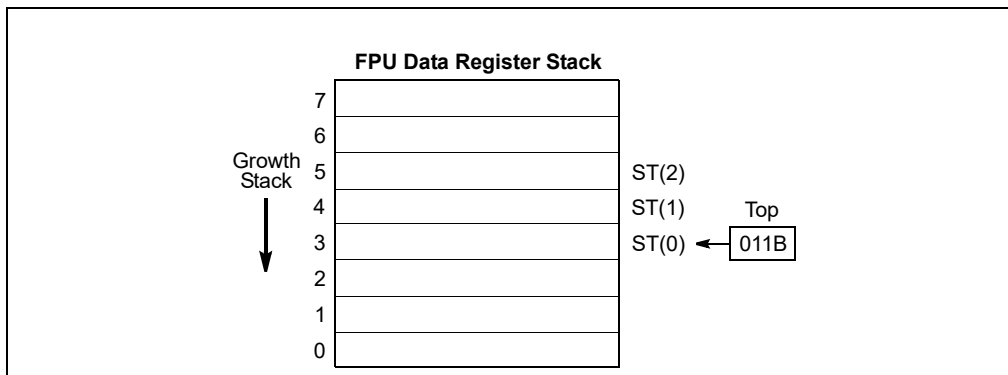


Figure 8-2. x87 FPU Data Register Stack

If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception indicates when wraparound might cause an unsaved value to be overwritten (see Section 8.5.1.1, "Stack Overflow or Underflow Exception (#IS)").

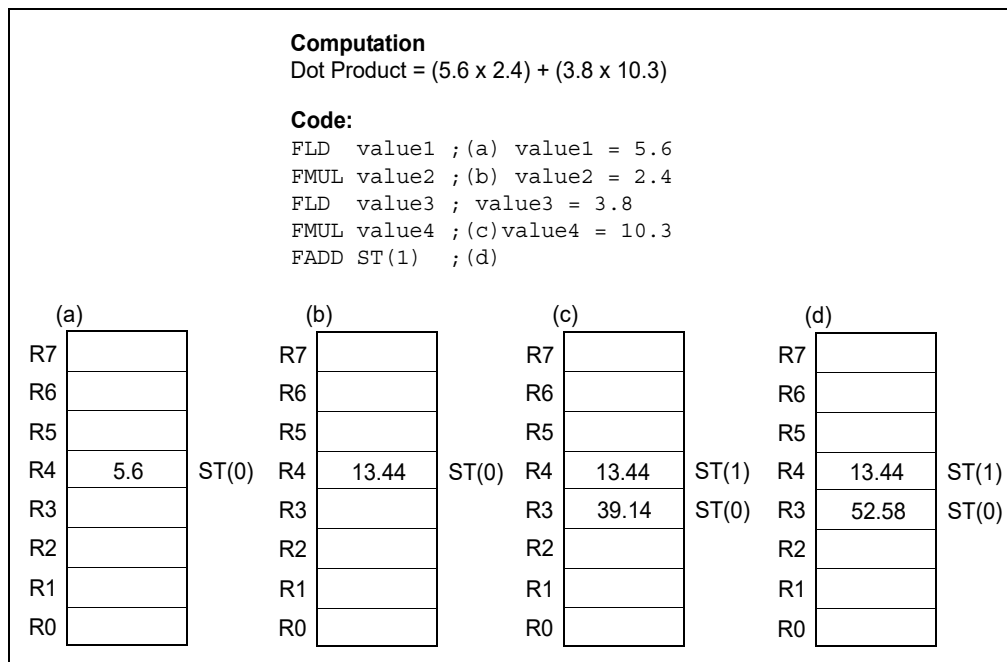
Many floating-point instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP. Assemblers support these

register addressing modes, using the expression  $ST(0)$ , or simply  $ST$ , to represent the current stack top and  $ST(i)$  to specify the  $i$ th register from  $TOP$  in the stack ( $0 \leq i \leq 7$ ). For example, if  $TOP$  contains 011B (register 3 is the top of the stack), the following instruction would add the contents of two registers in the stack (registers 3 and 5):

```
FADD ST, ST(2);
```

Figure 8-3 shows an example of how the stack structure of the x87 FPU registers and instructions are typically used to perform a series of computations. Here, a two-dimensional dot product is computed, as follows:

1. The first instruction (`FLD value1`) decrements the stack register pointer ( $TOP$ ) and loads the value 5.6 from memory into  $ST(0)$ . The result of this operation is shown in snap-shot (a).
2. The second instruction multiplies the value in  $ST(0)$  by the value 2.4 from memory and stores the result in  $ST(0)$ , shown in snap-shot (b).
3. The third instruction decrements  $TOP$  and loads the value 3.8 in  $ST(0)$ .
4. The fourth instruction multiplies the value in  $ST(0)$  by the value 10.3 from memory and stores the result in  $ST(0)$ , shown in snap-shot (c).
5. The fifth instruction adds the value and the value in  $ST(1)$  and stores the result in  $ST(0)$ , shown in snap-shot (d).



**Figure 8-3. Example x87 FPU Dot Product Computation**

The style of programming demonstrated in this example is supported by the floating-point instruction set. In cases where the stack structure causes computation bottlenecks, the `FXCH` (exchange x87 FPU register contents) instruction can be used to streamline a computation.

### 8.1.2.1 Parameter Passing With the x87 FPU Register Stack

Like the general-purpose registers, the contents of the x87 FPU data registers are unaffected by procedure calls, or in other words, the values are maintained across procedure boundaries. A calling procedure can thus use the x87 FPU data registers (as well as the procedure stack) for passing parameter between procedures. The called procedure can reference parameters passed through the register stack using the current stack register pointer ( $TOP$ ) and the  $ST(0)$  and  $ST(i)$  nomenclature. It is also common practice for a called procedure to leave a return value or result in register  $ST(0)$  when returning execution to the calling procedure or program.

When mixing MMX and x87 FPU instructions in the procedures or code sequences, the programmer is responsible for maintaining the integrity of parameters being passed in the x87 FPU data registers. If an MMX instruction is executed before the parameters in the x87 FPU data registers have been passed to another procedure, the parameters may be lost (see Section 9.5, "Compatibility with x87 FPU Architecture").

### 8.1.3 x87 FPU Status Register

The 16-bit x87 FPU status register (see Figure 8-4) indicates the current state of the x87 FPU. The flags in the x87 FPU status register include the FPU busy flag, top-of-stack (TOP) pointer, condition code flags, exception summary status flag, stack fault flag, and exception flags. The x87 FPU sets the flags in this register to show the results of operations.

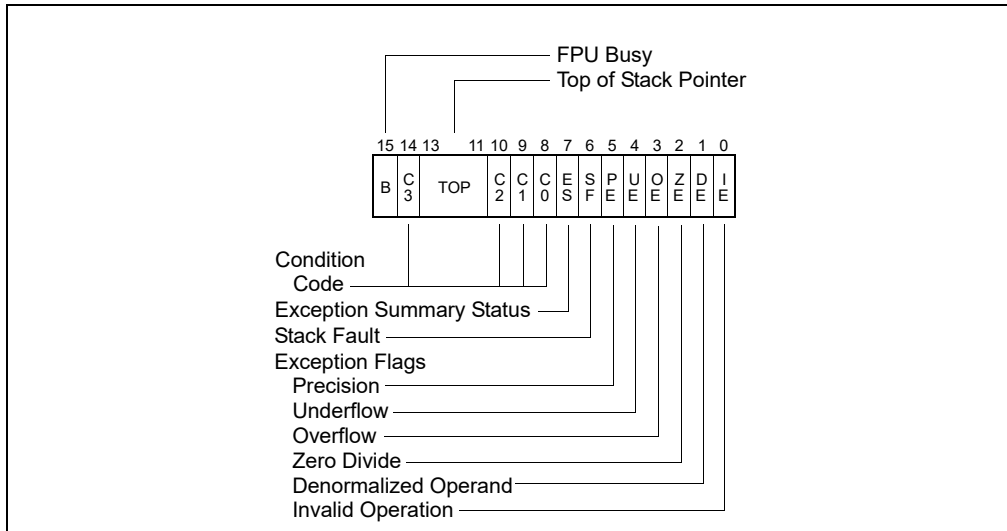


Figure 8-4. x87 FPU Status Word

The contents of the x87 FPU status register (referred to as the x87 FPU status word) can be stored in memory using the FSTSW/FNSTSW, FSTENV/FNSTENV, FSAVE/FNSAVE, and FXSAVE instructions. It can also be stored in the AX register of the integer unit, using the FSTSW/FNSTSW instructions.

#### 8.1.3.1 Top of Stack (TOP) Pointer

A pointer to the x87 FPU data register that is currently at the top of the x87 FPU register stack is contained in bits 11 through 13 of the x87 FPU status word. This pointer, which is commonly referred to as TOP (for top-of-stack), is a binary value from 0 to 7. See Section 8.1.2, "x87 FPU Data Registers," for more information about the TOP pointer.

#### 8.1.3.2 Condition Code Flags

The four condition code flags (C0 through C3) indicate the results of floating-point comparison and arithmetic operations. Table 8-1 summarizes the manner in which the floating-point instructions set the condition code flags. These condition code bits are used principally for conditional branching and for storage of information used in exception handling (see Section 8.1.4, "Branching and Conditional Moves on Condition Codes").

As shown in Table 8-1, the C1 condition code flag is used for a variety of functions. When both the IE and SF flags in the x87 FPU status word are set, indicating a stack overflow or underflow exception (#IS), the C1 flag distinguishes between overflow (C1 = 1) and underflow (C1 = 0). When the PE flag in the status word is set, indicating an inexact (rounded) result, the C1 flag is set to 1 if the last rounding by the instruction was upward. The FXAM instruction sets C1 to the sign of the value being examined.

The C2 condition code flag is used by the FPREM and FPREM1 instructions to indicate an incomplete reduction (or partial remainder). When a successful reduction has been completed, the C0, C3, and C1 condition code flags are set to the three least-significant bits of the quotient (Q2, Q1, and Q0, respectively). See “FPREM1—Partial Remainder” in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for more information on how these instructions use the condition code flags.

The FPTAN, FSIN, FCOS, and FSINCOS instructions set the C2 flag to 1 to indicate that the source operand is beyond the allowable range of  $\pm 2^{63}$  and clear the C2 flag if the source operand is within the allowable range.

Where the state of the condition code flags are listed as undefined in Table 8-1, do not rely on any specific value in these flags.

### 8.1.3.3 x87 FPU Floating-Point Exception Flags

The six x87 FPU floating-point exception flags (bits 0 through 5) of the x87 FPU status word indicate that one or more floating-point exceptions have been detected since the bits were last cleared. The individual exception flags (IE, DE, ZE, OE, UE, and PE) are described in detail in Section 8.4, “x87 FPU Floating-Point Exception Handling.” Each of the exception flags can be masked by an exception mask bit in the x87 FPU control word (see Section 8.1.5, “x87 FPU Control Word”). The exception summary status flag (ES, bit 7) is set when any of the unmasked exception flags are set. When the ES flag is set, the x87 FPU exception handler is invoked, using one of the techniques described in Section 8.7, “Handling x87 FPU Exceptions in Software.” (Note that if an exception flag is masked, the x87 FPU will still set the appropriate flag if the associated exception occurs, but it will not set the ES flag.)

The exception flags are “sticky” bits (once set, they remain set until explicitly cleared). They can be cleared by executing the FCLEX/FNCLEX (clear exceptions) instructions, by reinitializing the x87 FPU with the FINIT/FNINIT or FSAVE/FNSAVE instructions, or by overwriting the flags with an FRSTOR or FLDENV instruction.

The B-bit (bit 15) is included for 8087 compatibility only. It reflects the contents of the ES flag.

**Table 8-1. Condition Code Interpretation**

Instruction	C0	C3	C2	C1
FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FTST, FUCOM, FUCOMP, FUCOMPP	Result of Comparison		Operands are not Comparable	0 or #IS
FCOMI, FCOMIP, FUCOMI, FUCOMIP	Undefined. (These instructions set the status flags in the EFLAGS register.)			#IS
FXAM	Operand class			Sign
FPREM, FPREM1	Q2	Q1	0 = reduction complete 1 = reduction incomplete	Q0 or #IS
F2XM1, FADD, FADDP, FBSTP, FCMOVcc, FIADD, FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR, FIMUL, FIST, FISTP, FISUB, FISUBR, FMUL, FMULP, FPATAN, FRNDINT, FSCALE, FST, FSTP, FSUB, FSUBP, FSUBR, FSUBRP, FSQRT, FYL2X, FYL2XP1	Undefined			Roundup or #IS
FCOS, FSIN, FSINCOS, FPTAN	Undefined		0 = source operand within range 1 = source operand out of range	Roundup or #IS (Undefined if C2 = 1)
FABS, FBLD, FCHS, FDECSTP, FILD, FINCSTP, FLD, Load Constants, FSTP (ext. prec.), FXCH, FXPTRACT	Undefined			0 or #IS

**Table 8-1. Condition Code Interpretation (Contd.)**

FLDENV, FRSTOR	Each bit loaded from memory			
FFREE, FLDCW, FCLEX/FNCLEX, FNOP, FSTCW/FNSTCW, FSTENV/FNSTENV, FSTSW/FNSTSW,	Undefined			
FINIT/FNINIT, FSAVE/FNSAVE	0	0	0	0

### 8.1.3.4 Stack Fault Flag

The stack fault flag (bit 6 of the x87 FPU status word) indicates that stack overflow or stack underflow has occurred with data in the x87 FPU data register stack. The x87 FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition.

When this flag is set, the condition code flag C1 indicates the nature of the fault: overflow (C1 = 1) and underflow (C1 = 0). The SF flag is a “sticky” flag, meaning that after it is set, the processor does not clear it until it is explicitly instructed to do so (for example, by an FINIT/FNINIT, FCLEX/FNCLEX, or FSAVE/FNSAVE instruction).

See Section 8.1.7, “x87 FPU Tag Word,” for more information on x87 FPU stack faults.

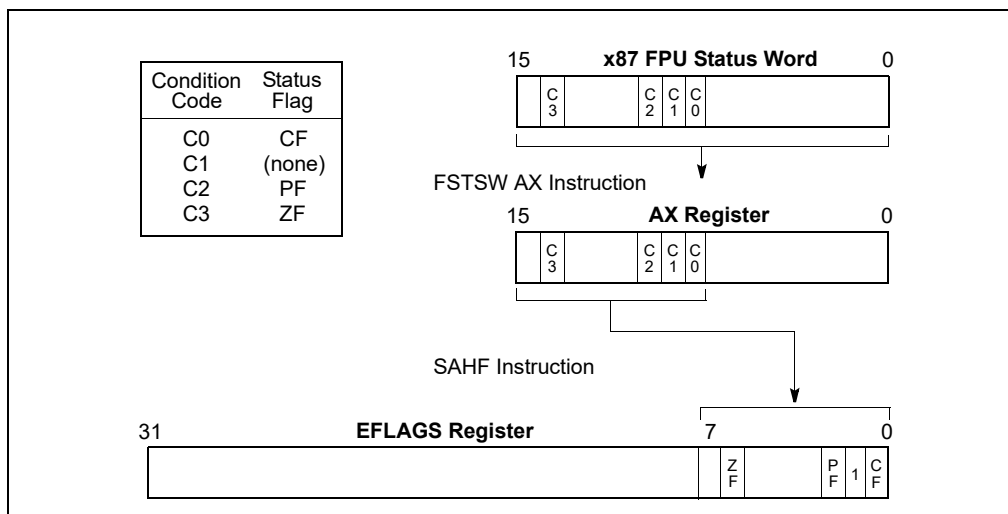
## 8.1.4 Branching and Conditional Moves on Condition Codes

The x87 FPU (beginning with the P6 family processors) supports two mechanisms for branching and performing conditional moves according to comparisons of two floating-point values. These mechanism are referred to here as the “old mechanism” and the “new mechanism.”

The old mechanism is available in x87 FPU’s prior to the P6 family processors and in P6 family processors. This mechanism uses the floating-point compare instructions (FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, and FICOMP) to compare two floating-point values and set the condition code flags (C0 through C3) according to the results. The contents of the condition code flags are then copied into the status flags of the EFLAGS register using a two step process (see Figure 8-5):

1. The FSTSW AX instruction moves the x87 FPU status word into the AX register.
2. The SAHF instruction copies the upper 8 bits of the AX register, which includes the condition code flags, into the lower 8 bits of the EFLAGS register.

When the condition code flags have been loaded into the EFLAGS register, conditional jumps or conditional moves can be performed based on the new settings of the status flags in the EFLAGS register.



**Figure 8-5. Moving the Condition Codes to the EFLAGS Register**

The new mechanism is available beginning with the P6 family processors. Using this mechanism, the new floating-point compare and set EFLAGS instructions (FCOMI, FCOMIP, FUCOMI, and FUCOMIP) compare two floating-point values and set the ZF, PF, and CF flags in the EFLAGS register directly. A single instruction thus replaces the three instructions required by the old mechanism.

Note also that the FCMOVcc instructions (also new in the P6 family processors) allow conditional moves of floating-point values (values in the x87 FPU data registers) based on the setting of the status flags (ZF, PF, and CF) in the EFLAGS register. These instructions eliminate the need for an IF statement to perform conditional moves of floating-point values.

### 8.1.5 x87 FPU Control Word

The 16-bit x87 FPU control word (see Figure 8-6) controls the precision of the x87 FPU and rounding method used. It also contains the x87 FPU floating-point exception mask bits. The control word is cached in the x87 FPU control register. The contents of this register can be loaded with the FLDCW instruction and stored in memory with the FSTCW/FNSTCW instructions.

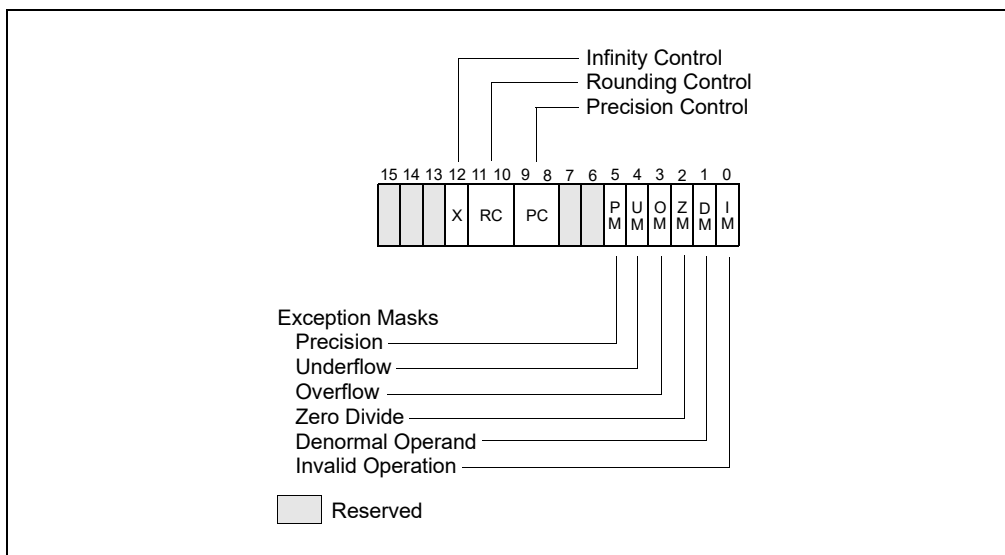


Figure 8-6. x87 FPU Control Word

When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU control word is set to 037FH, which masks all floating-point exceptions, sets rounding to nearest, and sets the x87 FPU precision to 64 bits.

#### 8.1.5.1 x87 FPU Floating-Point Exception Mask Bits

The exception-flag mask bits (bits 0 through 5 of the x87 FPU control word) mask the 6 floating-point exception flags in the x87 FPU status word. When one of these mask bits is set, its corresponding x87 FPU floating-point exception is blocked from being generated.

#### 8.1.5.2 Precision Control Field

The precision-control (PC) field (bits 8 and 9 of the x87 FPU control word) determines the precision (64, 53, or 24 bits) of floating-point calculations made by the x87 FPU (see Table 8-2). The default precision is double extended precision, which uses the full 64-bit significand available with the double extended-precision floating-point format of the x87 FPU data registers. This setting is best suited for most applications, because it allows applications to take full advantage of the maximum precision available with the x87 FPU data registers.



**Table 8-2. Precision Control Field (PC)**

Precision	PC Field
Single Precision (24 bits)	00B
Reserved	01B
Double Precision (53 bits)	10B
Double Extended Precision (64 bits)	11B

The double precision and single precision settings reduce the size of the significand to 53 bits and 24 bits, respectively. These settings are provided to support IEEE Standard 754 and to provide compatibility with the specifications of certain existing programming languages. Using these settings nullifies the advantages of the double extended-precision floating-point format's 64-bit significand length. When reduced precision is specified, the rounding of the significand value clears the unused bits on the right to zeros.

The precision-control bits only affect the results of the following floating-point instructions: FADD, FADDP, FIADD, FSUB, FSUBP, FISUB, FSUBR, FSUBRP, FISUBR, FMUL, FMULP, FIMUL, FDIV, FDIVP, FIDIV, FDIVR, FDIVRP, FIDIVR, and FSQRT.

### 8.1.5.3 Rounding Control Field

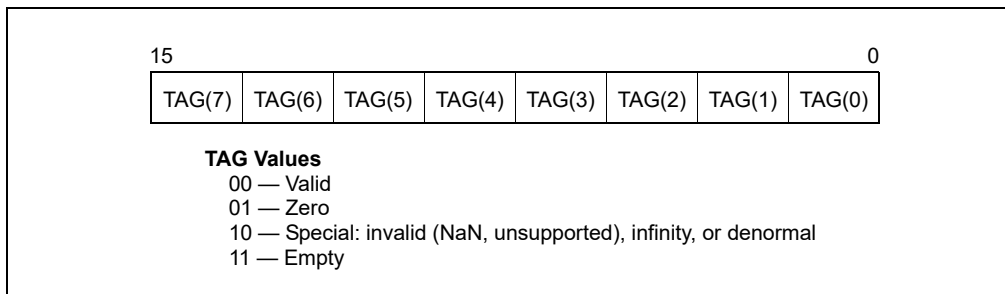
The rounding-control (RC) field of the x87 FPU control register (bits 10 and 11) controls how the results of x87 FPU floating-point instructions are rounded. See Section 4.8.4, "Rounding," for a discussion of rounding of floating-point values; See Section 4.8.4.1, "Rounding Control (RC) Fields", for the encodings of the RC field.

### 8.1.6 Infinity Control Flag

The infinity control flag (bit 12 of the x87 FPU control word) is provided for compatibility with the Intel 287 Math Coprocessor; it is not meaningful for later version x87 FPU coprocessors or IA-32 processors. See Section 4.8.3.3, "Signed Infinities," for information on how the x87 FPUs handle infinity values.

### 8.1.7 x87 FPU Tag Word

The 16-bit tag word (see Figure 8-7) indicates the contents of each the 8 registers in the x87 FPU data-register stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number, zero, or a special floating-point number (NaN, infinity, denormal, or unsupported format), or whether it is empty. The x87 FPU tag word is cached in the x87 FPU in the x87 FPU tag word register. When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU tag word is set to FFFFH, which marks all the x87 FPU data registers as empty.

**Figure 8-7. x87 FPU Tag Word**

Each tag in the x87 FPU tag word corresponds to a physical register (numbers 0 through 7). The current top-of-stack (TOP) pointer stored in the x87 FPU status word can be used to associate tags with registers relative to ST(0).

The x87 FPU uses the tag values to detect stack overflow and underflow conditions (see Section 8.5.1.1, “Stack Overflow or Underflow Exception (#IS)”).

Application programs and exception handlers can use this tag information to check the contents of an x87 FPU data register without performing complex decoding of the actual data in the register. To read the tag register, it must be stored in memory using either the FSTENV/FNSTENV or FSAVE/FNSAVE instructions. The location of the tag word in memory after being saved with one of these instructions is shown in Figures 8-9 through 8-12.

Software cannot directly load or modify the tags in the tag register. The FLDENV and FRSTOR instructions load an image of the tag register into the x87 FPU; however, the x87 FPU uses those tag values only to determine if the data registers are empty (11B) or non-empty (00B, 01B, or 10B).

If the tag register image indicates that a data register is empty, the tag in the tag register for that data register is marked empty (11B); if the tag register image indicates that the data register is non-empty, the x87 FPU reads the actual value in the data register and sets the tag for the register accordingly. This action prevents a program from setting the values in the tag register to incorrectly represent the actual contents of non-empty data registers.

## 8.1.8 x87 FPU Instruction and Data (Operand) Pointers

The x87 FPU stores pointers to the instruction and data (operand) for the last non-control instruction executed. These are the x87 FPU instruction pointer and x87 FPU data (operand) pointers; software can save these pointers to provide state information for exception handlers. The pointers are illustrated in Figure 8-1 (the figure illustrates the pointers as used outside 64-bit mode; see below).

Note that the value in the x87 FPU data pointer is always a pointer to a memory operand. If the last non-control instruction that was executed did not have a memory operand, the value in the data pointer is undefined (reserved). If CPUID.(EAX=07H,ECX=0H):EBX[bit 6] = 1, the data pointer is updated only for x87 non-control instructions that incur unmasked x87 exceptions.

The contents of the x87 FPU instruction and data pointers remain unchanged when any of the following instructions are executed: FCLEX/FNCLEX, FLDCW, FSTCW/FNSTCW, FSTSW/FNSTSW, FSTENV/FNSTENV, FLDENV, and WAIT/FWAIT.

For all the x87 FPUs and Numeric Processor Extensions (NPXs) except the 8087, the x87 FPU instruction pointer points to any prefixes that preceded the instruction. For the 8087, the x87 FPU instruction pointer points only to the actual opcode.

The x87 FPU instruction and data pointers each consists of an offset and a segment selector:

- The x87 FPU Instruction Pointer Offset (FIP) comprises 64 bits on processors that support IA-32e mode; on other processors, it offset comprises 32 bits.
- The x87 FPU Instruction Pointer Selector (FCS) comprises 16 bits.
- The x87 FPU Data Pointer Offset (FDP) comprises 64 bits on processors that support IA-32e mode; on other processors, it offset comprises 32 bits.
- The x87 FPU Data Pointer Selector (FDS) comprises 16 bits.

The pointers are accessed by the FINIT/FNINIT, FLDENV, FRSTOR, FSAVE/FNSAVE, FSTENV/FNSTENV, FXRSTOR, FXSAVE, XRSTOR, XSAVE, and XSAVEOPT instructions as follows:

- FINIT/FNINIT. Each instruction clears FIP, FCS, FDP, and FDS.
- FLDENV, FRSTOR. These instructions use the memory formats given in Figures 8-9 through 8-12:
  - For each of FIP and FDP, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
  - If CR0.PE = 1, each instruction loads FCS and FDS from memory; otherwise, it clears them.
- FSAVE/FNSAVE, FSTENV/FNSTENV. These instructions use the memory formats given in Figures 8-9 through 8-12.
  - Each instruction saves the lower 32 bits of each FIP and FDP into memory. the upper 32 bits are not saved.
  - If CR0.PE = 1, each instruction saves FCS and FDS into memory. If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS; it saves each as 0000H.

- After saving these data into memory, FSAVE/FNSAVE clears FIP, FCS, FDP, and FDS.
- FXRSTOR, XRSTOR. These instructions load data from a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-43, 3-46, and 3-47 in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.
  - Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
    - For each of FIP and FDP, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
    - Each instruction loads FCS and FDS from memory.
  - In 64-bit mode with REX.W = 1, the instructions operate as follows:
    - Each instruction loads FIP and FDP from memory.
    - Each instruction clears FCS and FDS.
- FXSAVE, XSAVE, and XSAVEOPT. These instructions store data into a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-43, 3-46, and 3-47 in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.
  - Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
    - Each instruction saves the lower 32 bits of each of FIP and FDP into memory. The upper 32 bits are not saved.
    - Each instruction saves FCS and FDS into memory. If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS; it saves each as 0000H.
  - In 64-bit mode with REX.W = 1, each instruction saves FIP and FDP into memory. FCS and FDS are not saved.

## 8.1.9 Last Instruction Opcode

The x87 FPU stores in the 11-bit x87 FPU opcode register (FOP) the opcode of the last x87 non-control instruction executed that incurred an unmasked x87 exception. (This information provides state information for exception handlers.) Only the first and second opcode bytes (after all prefixes) are stored in the x87 FPU opcode register. Figure 8-8 shows the encoding of these two bytes. Since the upper 5 bits of the first opcode byte are the same for all floating-point opcodes (11011B), only the lower 3 bits of this byte are stored in the opcode register.

### 8.1.9.1 Fopcode Compatibility Sub-mode

Some Pentium 4 and Intel Xeon processors provide program control over the value stored into FOP. Here, bit 2 of the IA32\_MISC\_ENABLE MSR enables (set) or disables (clear) the fopcode compatibility mode.

If fopcode compatibility mode is enabled, FOP is defined as it had been in previous IA-32 implementations, as the opcode of the last x87 non-control instruction executed (even if that instruction did not incur an unmasked x87 exception).

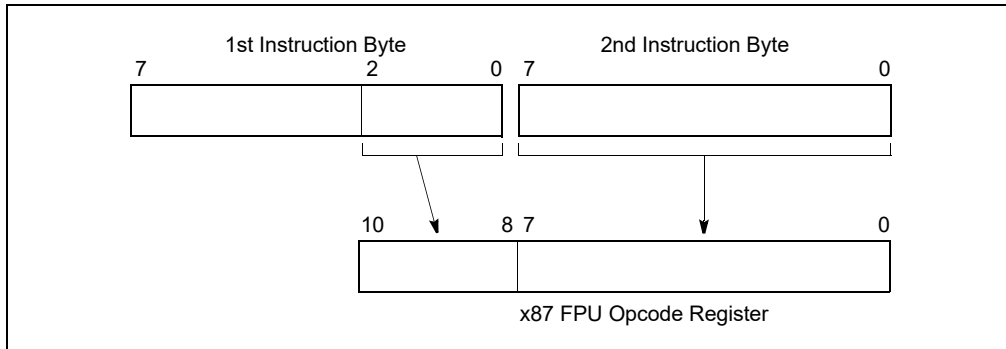


Figure 8-8. Contents of x87 FPU Opcode Registers

The fopcode compatibility mode should be enabled only when x87 FPU floating-point exception handlers are designed to use the fopcode to analyze program performance or restart a program after an exception has been handled.

More recent Intel 64 processors do not support fopcode compatibility mode and do not allow software to set bit 2 of the IA32\_MISC\_ENABLE MSR.

### 8.1.10 Saving the x87 FPU's State with FSTENV/FNSTENV and FSAVE/FNSAVE

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions store x87 FPU state information in memory for use by exception handlers and other system and application software. The FSTENV/FNSTENV instruction saves the contents of the status, control, tag, x87 FPU instruction pointer, x87 FPU data pointer, and opcode registers. The FSAVE/FNSAVE instruction stores that information plus the contents of the x87 FPU data registers. Note that the FSAVE/FNSAVE instruction also initializes the x87 FPU to default values (just as the FINIT/FNINIT instruction does) after it has saved the original state of the x87 FPU.

The manner in which this information is stored in memory depends on the operating mode of the processor (protected mode or real-address mode) and on the operand-size attribute in effect (32-bit or 16-bit). See Figures 8-9 through 8-12. In virtual-8086 mode or SMM, the real-address mode formats shown in Figure 8-12 is used. See Chapter 34, "System Management Mode," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for information on using the x87 FPU while in SMM.

The FLDENV and FRSTOR instructions allow x87 FPU state information to be loaded from memory into the x87 FPU. Here, the FLDENV instruction loads only the status, control, tag, x87 FPU instruction pointer, x87 FPU data pointer, and opcode registers, and the FRSTOR instruction loads all the x87 FPU registers, including the x87 FPU stack registers.

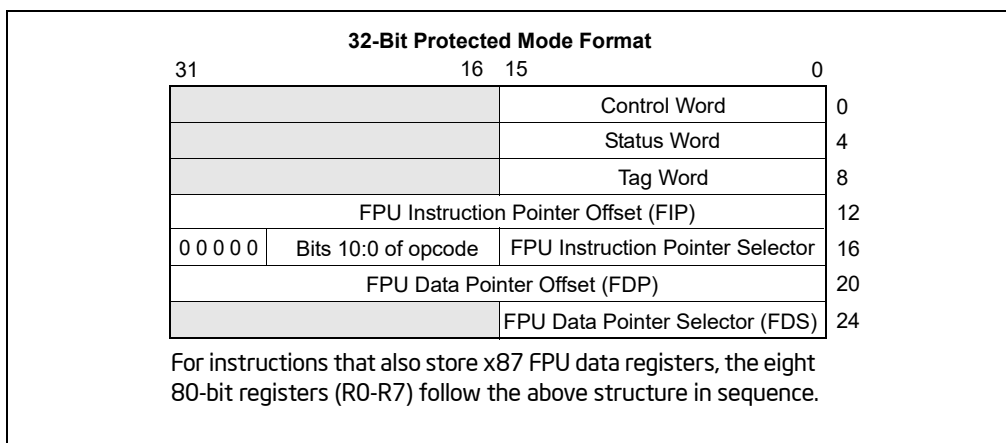
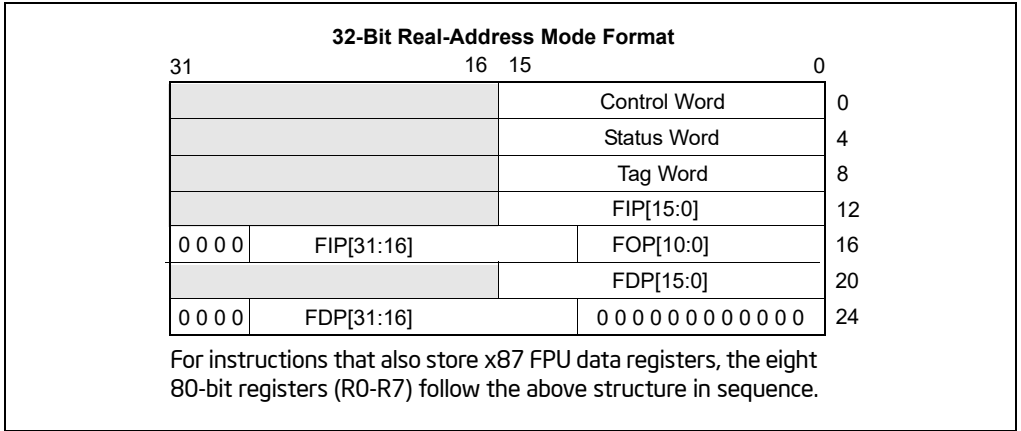
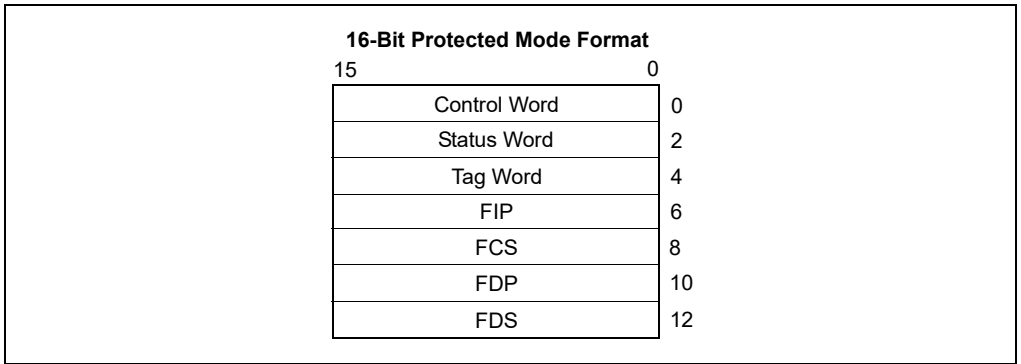


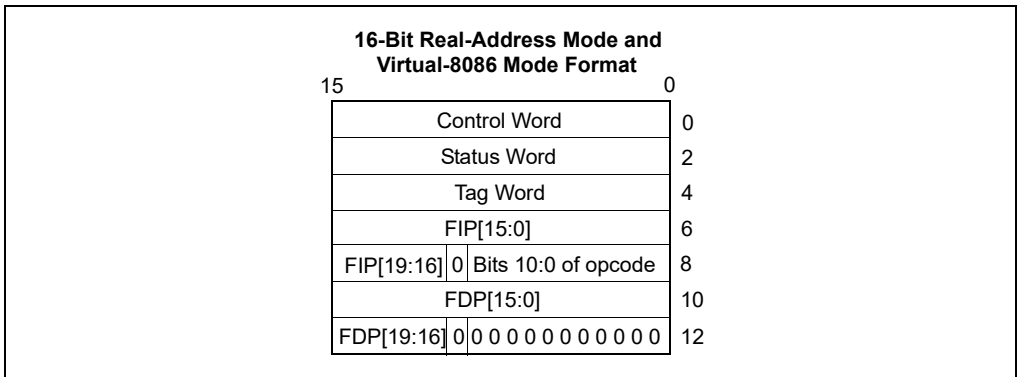
Figure 8-9. Protected Mode x87 FPU State Image in Memory, 32-Bit Format



**Figure 8-10. Real Mode x87 FPU State Image in Memory, 32-Bit Format**



**Figure 8-11. Protected Mode x87 FPU State Image in Memory, 16-Bit Format**



**Figure 8-12. Real Mode x87 FPU State Image in Memory, 16-Bit Format**

### 8.1.11 Saving the x87 FPU's State with FXSAVE

The FXSAVE and FXRSTOR instructions save and restore, respectively, the x87 FPU state along with the state of the XMM registers and the MXCSR register. Using the FXSAVE instruction to save the x87 FPU state has two benefits: (1) FXSAVE executes faster than FSAVE, and (2) FXSAVE saves the entire x87 FPU, MMX, and XMM state in one operation. See Section 10.5, "FXSAVE and FXRSTOR Instructions," for additional information about these instructions.

## 8.2 X87 FPU DATA TYPES

The x87 FPU recognizes and operates on the following seven data types (see Figures 8-13): single-precision floating point, double-precision floating point, double extended-precision floating point, signed word integer, signed doubleword integer, signed quadword integer, and packed BCD decimal integers.

For detailed information about these data types, see Section 4.2.2, "Floating-Point Data Types," Section 4.2.1.2, "Signed Integers," and Section 4.7, "BCD and Packed BCD Integers."

With the exception of the 80-bit double extended-precision floating-point format, all of these data types exist in memory only. When they are loaded into x87 FPU data registers, they are converted into double extended-precision floating-point format and operated on in that format.

Denormal values are also supported in each of the floating-point types, as required by IEEE Standard 754. When a denormal number in single-precision or double-precision floating-point format is used as a source operand and the denormal exception is masked, the x87 FPU automatically **normalizes** the number when it is converted to double extended-precision format.

When stored in memory, the least significant byte of an x87 FPU data-type value is stored at the initial address specified for the value. Successive bytes from the value are then stored in successively higher addresses in memory. The floating-point instructions load and store memory operands using only the initial address of the operand.

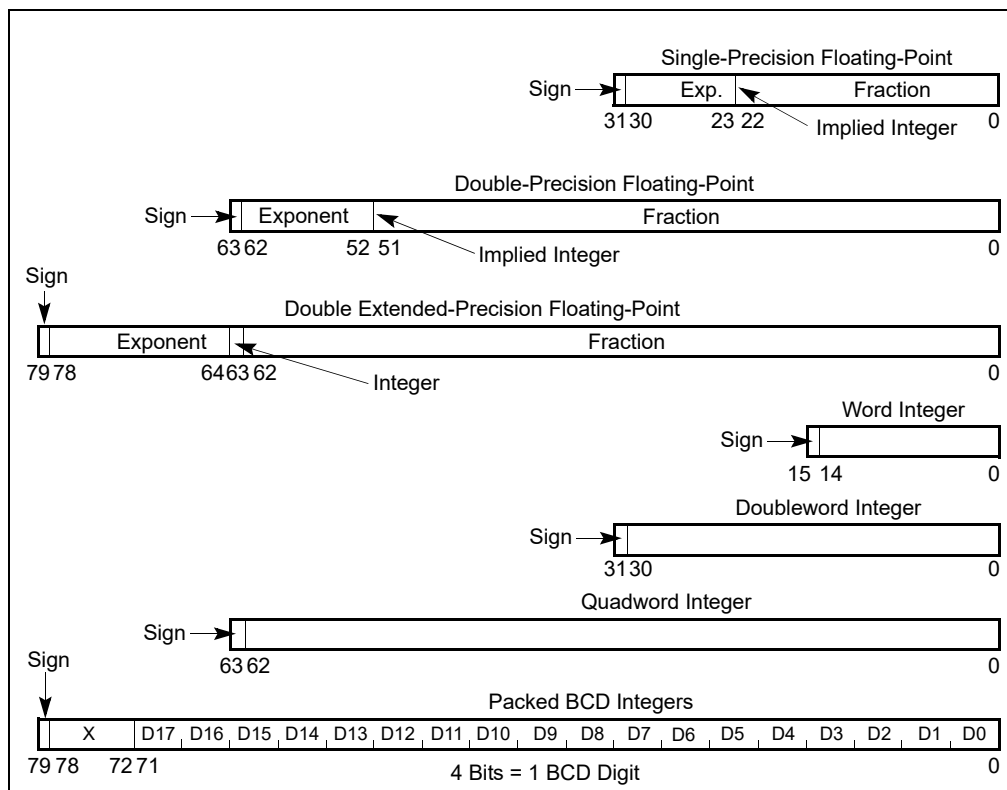


Figure 8-13. x87 FPU Data Type Formats

As a general rule, values should be stored in memory in double-precision format. This format provides sufficient range and precision to return correct results with a minimum of programmer attention. The single-precision format is useful for debugging algorithms, because rounding problems will manifest themselves more quickly in this format. The double extended-precision format is normally reserved for holding intermediate results in the x87 FPU registers and constants. Its extra length is designed to shield final results from the effects of rounding and overflow/underflow in intermediate calculations. However, when an application requires the maximum range and precision of the x87 FPU (for data storage, computations, and results), values can be stored in memory in double extended-precision format.

### 8.2.1 Indefinites

For each x87 FPU data type, one unique encoding is reserved for representing the special value **indefinite**. The x87 FPU produces indefinite values as responses to some masked floating-point invalid-operation exceptions. See Tables 4-1, 4-3, and 4-4 for the encoding of the integer indefinite, QNaN floating-point indefinite, and packed BCD integer indefinite, respectively.

The binary integer encoding 100..00B represents either of two things, depending on the circumstances of its use:

- The largest negative number supported by the format ( $-2^{15}$ ,  $-2^{31}$ , or  $-2^{63}$ )
- The **integer indefinite** value

If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the x87 FPU interprets it as the largest negative number representable in the format being used. If the x87 FPU detects an invalid operation when storing an integer value in memory with an FIST/FISTP instruction and the invalid-operation exception is masked, the x87 FPU stores the integer indefinite encoding in the destination operand as a masked response to the exception. In situations where the origin of a value with this encoding may be ambiguous, the invalid-operation exception flag can be examined to see if the value was produced as a response to an exception.

### 8.2.2 Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals

The double extended-precision floating-point format permits many encodings that do not fall into any of the categories shown in Table 4-3. Table 8-3 shows these unsupported encodings. Some of these encodings were supported by the Intel 287 math coprocessor; however, most of them are not supported by the Intel 387 math coprocessor and later IA-32 processors. These encodings are no longer supported due to changes made in the final version of IEEE Standard 754 that eliminated these encodings.

Specifically, the categories of encodings formerly known as pseudo-NaNs, pseudo-infinities, and un-normal numbers are not supported and should not be used as operand values. The Intel 387 math coprocessor and later IA-32 processors generate an invalid-operation exception when these encodings are encountered as operands.

Beginning with the Intel 387 math coprocessor, the encodings formerly known as pseudo-denormal numbers are not generated by IA-32 processors. When encountered as operands, however, they are handled correctly; that is, they are treated as denormals and a denormal exception is generated. Pseudo-denormal numbers should not be used as operand values. They are supported by current IA-32 processors (as described here) to support legacy code.

**Table 8-3. Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals**

Class		Sign	Biased Exponent	Significand	
				Integer	Fraction
Positive Pseudo-NaNs	Quiet	0	11..11	0	11..11
		.	.		.
	0	11..11		10..00	
	.	.		.	
Signaling	0	11..11	0	01..11	
	.	.		.	
	0	11..11		00..01	
	.	.		.	
Positive Floating Point	Pseudo-infinity	0	11..11	0	00..00
	Unnormals	0	11..10	0	11..11
		.	.		.
		0	00..01		00..00
	Pseudo-denormals	0	00..00	1	11..11
		.	.		.
	0	00..00		00..00	
	.	.		.	

**Table 8-3. Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals (Contd.)**

Negative Floating Point	Pseudo-denormals	1 . 1	00..00 . 00..00	1	11..11 . 00..00
	Unnormals	1 . 1	11..10 . 00..01	0	11..01 . 00..00
		Pseudo-infinity	1 . 1	11..11 . 11..11	0
Negative Pseudo-NaNs	Signaling	1 . 1	11..11 . 11..11	0	01..11 . 00..01
		Quiet	1 . 1	11..11 . 11..11	0
			← 15 bits →		← 63 bits →

## 8.3 X87 FPU INSTRUCTION SET

The floating-point instructions that the x87 FPU supports can be grouped into six functional categories:

- Data transfer instructions
- Basic arithmetic instructions
- Comparison instructions
- Transcendental instructions
- Load constant instructions
- x87 FPU control instructions

See Section , “CPUID.EAX=8000001H:ECX.PREFTEHCHW[bit 8]: if 1 indicates the processor supports the PREFTEHCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFTEHCHWT1[bit 0]: if 1 indicates the processor supports the PREFTEHCHWT1 instruction.” for a list of the floating-point instructions by category.

The following section briefly describes the instructions in each category. Detailed descriptions of the floating-point instructions are given in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.

### 8.3.1 Escape (ESC) Instructions

All of the instructions in the x87 FPU instruction set fall into a class of instructions known as escape (ESC) instructions. All of these instructions have a common opcode format, where the first byte of the opcode is one of the numbers from D8H through DFH.

### 8.3.2 x87 FPU Instruction Operands

Most floating-point instructions require one or two operands, located on the x87 FPU data-register stack or in memory. (None of the floating-point instructions accept immediate operands.)

When an operand is located in a data register, it is referenced relative to the ST(0) register (the register at the top of the register stack), rather than by a physical register number. Often the ST(0) register is an implied operand.

Operands in memory can be referenced using the same operand addressing methods described in Section 3.7, “Operand Addressing.”



### 8.3.3 Data Transfer Instructions

The data transfer instructions (see Table 8-4) perform the following operations:

- Load a floating-point, integer, or packed BCD operand from memory into the ST(0) register.
- Store the value in an ST(0) register to memory in floating-point, integer, or packed BCD format.
- Move values between registers in the x87 FPU register stack.

The FLD (load floating point) instruction pushes a floating-point operand from memory onto the top of the x87 FPU data-register stack. If the operand is in single-precision or double-precision floating-point format, it is automatically converted to double extended-precision floating-point format. This instruction can also be used to push the value in a selected x87 FPU data register onto the top of the register stack.

The FILD (load integer) instruction converts an integer operand in memory into double extended-precision floating-point format and pushes the value onto the top of the register stack. The FBLD (load packed decimal) instruction performs the same load operation for a packed BCD operand in memory.

**Table 8-4. Data Transfer Instructions**

Floating Point		Integer		Packed Decimal	
FLD	Load Floating Point	FILD	Load Integer	FBLD	Load Packed Decimal
FST	Store Floating Point	FIST	Store Integer		
FSTP	Store Floating Point and Pop	FISTP	Store Integer and Pop	FBSTP	Store Packed Decimal and Pop
FXCH	Exchange Register Contents				
FCMOV <sub>cc</sub>	Conditional Move				

The FST (store floating point) and FIST (store integer) instructions store the value in register ST(0) in memory in the destination format (floating point or integer, respectively). Again, the format conversion is carried out automatically.

The FSTP (store floating point and pop), FISTP (store integer and pop), and FBSTP (store packed decimal and pop) instructions store the value in the ST(0) registers into memory in the destination format (floating point, integer, or packed BCD), then performs a **pop** operation on the register stack. A pop operation causes the ST(0) register to be marked empty and the stack pointer (TOP) in the x87 FPU control work to be incremented by 1. The FSTP instruction can also be used to copy the value in the ST(0) register to another x87 FPU register [ST(i)].

The FXCH (exchange register contents) instruction exchanges the value in a selected register in the stack [ST(i)] with the value in ST(0).

The FCMOV<sub>cc</sub> (conditional move) instructions move the value in a selected register in the stack [ST(i)] to register ST(0) if a condition specified with a condition code (cc) is satisfied (see Table 8-5). The condition being tested for is represented by the status flags in the EFLAGS register. The condition code mnemonics are appended to the letters "FCMOV" to form the mnemonic for a FCMOV<sub>cc</sub> instruction.

**Table 8-5. Floating-Point Conditional Move Instructions**

Instruction Mnemonic	Status Flag States	Condition Description
FCMOVB	CF=1	Below
FCMOVNB	CF=0	Not below
FCMOVE	ZF=1	Equal
FCMOVNE	ZF=0	Not equal

**Table 8-5. Floating-Point Conditional Move Instructions (Contd.)**

Instruction Mnemonic	Status Flag States	Condition Description
FCMOVBE	CF=1 or ZF=1	Below or equal
FCMOVNBE	CF=0 or ZF=0	Not below nor equal
FCMOVU	PF=1	Unordered
FCMOVNU	PF=0	Not unordered

Like the CMOVcc instructions, the FCMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor. Software can check if the FCMOVcc instructions are supported by checking the processor's feature information with the CPUID instruction.

### 8.3.4 Load Constant Instructions

The following instructions push commonly used constants onto the top [ST(0)] of the x87 FPU register stack:

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load $\pi$
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

The constant values have full double extended-precision floating-point precision (64 bits) and are accurate to approximately 19 decimal digits. They are stored internally in a format more precise than double extended-precision floating point. When loading the constant, the x87 FPU rounds the more precise internal constant according to the RC (rounding control) field of the x87 FPU control word. The inexact-result exception (#P) is not generated as a result of this rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up. See Section 8.3.8, "Approximation of Pi," for information on the  $\pi$  constant.

### 8.3.5 Basic Arithmetic Instructions

The following floating-point instructions perform basic arithmetic operations on floating-point numbers. Where applicable, these instructions match IEEE Standard 754:

FADD/FADDP	Add floating point
FIADD	Add integer to floating point
FSUB/FSUBP	Subtract floating point
FISUB	Subtract integer from floating point
FSUBR/FSUBRP	Reverse subtract floating point
FISUBR	Reverse subtract floating point from integer
FMUL/FMULP	Multiply floating point
FIMUL	Multiply integer by floating point
FDIV/FDIVP	Divide floating point
FIDIV	Divide floating point by integer
FDIVR/FDIVRP	Reverse divide
FIDIVR	Reverse divide integer by floating point
FABS	Absolute value
FCHS	Change sign

FSQRT	Square root
FPREM	Partial remainder
FPREM1	IEEE partial remainder
FRNDINT	Round to integral value
FXTRACT	Extract exponent and significand

The add, subtract, multiply and divide instructions operate on the following types of operands:

- Two x87 FPU data registers
- An x87 FPU data register and a floating-point or integer value in memory

See Section 8.1.2, “x87 FPU Data Registers,” for a description of how operands are referenced on the data register stack.

Operands in memory can be in single-precision floating-point, double-precision floating-point, word-integer, or doubleword-integer format. They are converted to double extended-precision floating-point format automatically.

Reverse versions of the subtract (FSUBR) and divide (FDIVR) instructions enable efficient coding. For example, the following options are available with the FSUB and FSUBR instructions for operating on values in a specified x87 FPU data register  $ST(i)$  and the  $ST(0)$  register:

FSUB:

```
ST(0) := ST(0) - ST(i)
ST(i) := ST(i) - ST(0)
```

FSUBR:

```
ST(0) := ST(i) - ST(0)
ST(i) := ST(0) - ST(i)
```

These instructions eliminate the need to exchange values between the  $ST(0)$  register and another x87 FPU register to perform a subtraction or division.

The pop versions of the add, subtract, multiply, and divide instructions offer the option of popping the x87 FPU register stack following the arithmetic operation. These instructions operate on values in the  $ST(i)$  and  $ST(0)$  registers, store the result in the  $ST(i)$  register, and pop the  $ST(0)$  register.

The FPREM instruction computes the remainder from the division of two operands in the manner used by the Intel 8087 and Intel 287 math coprocessors; the FPREM1 instruction computes the remainder in the manner specified in IEEE Standard 754.

The FSQRT instruction computes the square root of the source operand.

The FRNDINT instruction returns a floating-point value that is the integral value closest to the source value in the direction of the rounding mode specified in the RC field of the x87 FPU control word.

The FABS, FCHS, and FXTRACT instructions perform convenient arithmetic operations. The FABS instruction produces the absolute value of the source operand. The FCHS instruction changes the sign of the source operand. The FXTRACT instruction separates the source operand into its exponent and fraction and stores each value in a register in floating-point format.

### 8.3.6 Comparison and Classification Instructions

The following instructions compare or classify floating-point values:

FCOM/FCOMP/FCOMPP Compare floating point and set x87 FPU condition code flags.

FUCOM/FUCOMP/FUCOMPP Unordered compare floating point and set x87 FPU condition code flags.

FICOM/FICOMP Compare integer and set x87 FPU condition code flags.

**FCOMI/FCOMIP** Compare floating point and set EFLAGS status flags.

**FUCOMI/FUCOMIP** Unordered compare floating point and set EFLAGS status flags.

**FTST** Test (compare floating point with 0.0).  
**FXAM** Examine.

Comparison of floating-point values differ from comparison of integers because floating-point values have four (rather than three) mutually exclusive relationships: less than, equal, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN or in an unsupported format. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have less than, equal, or greater than relationships with other floating-point values.

The FCOM, FCOMP, and FCOMPP instructions compare the value in register ST(0) with a floating-point source operand and set the condition code flags (C0, C2, and C3) in the x87 FPU status word according to the results (see Table 8-6).

If an unordered condition is detected (one or both of the values are NaNs or in an undefined format), a floating-point invalid-operation exception is generated.

The pop versions of the instruction pop the x87 FPU register stack once or twice after the comparison operation is complete.

The FUCOM, FUCOMP, and FUCOMPP instructions operate the same as the FCOM, FCOMP, and FCOMPP instructions. The only difference is that with the FUCOM, FUCOMP, and FUCOMPP instructions, if an unordered condition is detected because one or both of the operands are QNaNs, the floating-point invalid-operation exception is not generated.

**Table 8-6. Setting of x87 FPU Condition Code Flags for Floating-Point Number Comparisons**

Condition	C3	C2	C0
ST(0) > Source Operand	0	0	0
ST(0) < Source Operand	0	0	1
ST(0) = Source Operand	1	0	0
Unordered	1	1	1

The FICOM and FICOMP instructions also operate the same as the FCOM and FCOMP instructions, except that the source operand is an integer value in memory. The integer value is automatically converted into an double extended-precision floating-point value prior to making the comparison. The FICOMP instruction pops the x87 FPU register stack following the comparison operation.

The FTST instruction performs the same operation as the FCOM instruction, except that the value in register ST(0) is always compared with the value 0.0.

The FCOMI and FCOMIP instructions were introduced into the IA-32 architecture in the P6 family processors. They perform the same comparison as the FCOM and FCOMP instructions, except that they set the status flags (ZF, PF, and CF) in the EFLAGS register to indicate the results of the comparison (see Table 8-7) instead of the x87 FPU condition code flags. The FCOMI and FCOMIP instructions allow condition branch instructions (Jcc) to be executed directly from the results of their comparison.

**Table 8-7. Setting of EFLAGS Status Flags for Floating-Point Number Comparisons**

Comparison Results	ZF	PF	CF
ST0 > ST( <i>i</i> )	0	0	0
ST0 < ST( <i>i</i> )	0	0	1
ST0 = ST( <i>i</i> )	1	0	0
Unordered	1	1	1

Software can check if the FCOMI and FCOMIP instructions are supported by checking the processor’s feature information with the CPUID instruction.

The FUCOMI and FUCOMIP instructions operate the same as the FCOMI and FCOMIP instructions, except that they do not generate a floating-point invalid-operation exception if the unordered condition is the result of one or both of the operands being a QNaN. The FCOMIP and FUCOMIP instructions pop the x87 FPU register stack following the comparison operation.

The FXAM instruction determines the classification of the floating-point value in the ST(0) register (that is, whether the value is zero, a denormal number, a normal finite number,  $\infty$ , a NaN, or an unsupported format) or that the register is empty. It sets the x87 FPU condition code flags to indicate the classification (see “FXAM—Examine” in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). It also sets the C1 flag to indicate the sign of the value.

### 8.3.6.1 Branching on the x87 FPU Condition Codes

The processor does not offer any control-flow instructions that branch on the setting of the condition code flags (C0, C2, and C3) in the x87 FPU status word. To branch on the state of these flags, the x87 FPU status word must first be moved to the AX register in the integer unit. The FSTSW AX (store status word) instruction can be used for this purpose. When these flags are in the AX register, the TEST instruction can be used to control conditional branching as follows:

1. Check for an unordered result. Use the TEST instruction to compare the contents of the AX register with the constant 0400H (see Table 8-8). This operation will clear the ZF flag in the EFLAGS register if the condition code flags indicate an unordered result; otherwise, the ZF flag will be set. The JNZ instruction can then be used to transfer control (if necessary) to a procedure for handling unordered operands.

**Table 8-8. TEST Instruction Constants for Conditional Branching**

Order	Constant	Branch
ST(0) > Source Operand	4500H	JZ
ST(0) < Source Operand	0100H	JNZ
ST(0) = Source Operand	4000H	JNZ
Unordered	0400H	JNZ

2. Check ordered comparison result. Use the constants given in Table 8-8 in the TEST instruction to test for a less than, equal to, or greater than result, then use the corresponding conditional branch instruction to transfer program control to the appropriate procedure or section of code.

If a program or procedure has been thoroughly tested and it incorporates periodic checks for QNaN results, then it is not necessary to check for the unordered result every time a comparison is made.

See Section 8.1.4, “Branching and Conditional Moves on Condition Codes,” for another technique for branching on x87 FPU condition codes.

Some non-comparison x87 FPU instructions update the condition code flags in the x87 FPU status word. To ensure that the status word is not altered inadvertently, store it immediately following a comparison operation.

### 8.3.7 Trigonometric Instructions

The following instructions perform four common trigonometric functions:

- FSIN                      Sine
- FCOS                     Cosine
- FSINCOS                Sine and cosine
- FPTAN                    Tangent
- FPATAN                 Arctangent

These instructions operate on the top one or two registers of the x87 FPU register stack and they return their results to the stack. The source operands for the FSIN, FCOS, FSINCOS, and FPTAN instructions must be given in radians; the source operand for the FPATAN instruction is given in rectangular coordinate units.

The FSINCOS instruction returns both the sine and the cosine of a source operand value. It operates faster than executing the FSIN and FCOS instructions in succession.

The FPATAN instruction computes the arctangent of ST(1) divided by ST(0), returning a result in radians. It is useful for converting rectangular coordinates to polar coordinates.

See Section 8.3.8, "Approximation of Pi" and Section 8.3.10, "Transcendental Instruction Accuracy" for information regarding the accuracy of these instructions.

### 8.3.8 Approximation of Pi

When the argument (source operand) of a trigonometric function is within the domain of the function, the argument is automatically reduced by the appropriate multiple of  $2\pi$  through the same reduction mechanism used by the FPREM and FPREM1 instructions. The internal value of  $\pi$  (3.1415926...) that the x87 FPU uses for argument reduction and other computations, denoted as Pi in the expression below. The numerical value of Pi can be written as:

$$\text{Pi} = 0.f * 2^2$$

where the fraction f is expressed in binary form as:

$$f = \text{C90FDAA2 2168C234 C}$$

(The spaces in the fraction above indicate 32-bit boundaries.)

The internal approximation Pi of the value  $\pi$  has a 66 significant bits. Since the exact value of  $\pi$  represented in binary has the next 3 bits equal to 0, it means that Pi is the value of  $\pi$  rounded to nearest-even to 68 bits, and also the value of  $\pi$  rounded toward zero (truncated) to 69 bits.

However, accuracy problems may arise because this relatively short finite approximation Pi of the number  $\pi$  is used for calculating the reduced argument of the trigonometric function approximations in the implementations of FSIN, FCOS, FSINCOS, and FPTAN. Alternately, this means that FSIN (x), FCOS (x), and FPTAN (x) are really approximating the mathematical functions  $\sin(x * \pi / \text{Pi})$ ,  $\cos(x * \pi / \text{Pi})$ , and  $\tan(x * \pi / \text{Pi})$ , and not exactly  $\sin(x)$ ,  $\cos(x)$ , and  $\tan(x)$ . (Note that FSINCOS is the equivalent of FSIN and FCOS combined together). The period of  $\sin(x * \pi / \text{Pi})$  for example is  $2 * \text{Pi}$ , and not  $2\pi$ .

See also Section 8.3.10, "Transcendental Instruction Accuracy" for more information on the accuracy of these functions.

### 8.3.9 Logarithmic, Exponential, and Scale

The following instructions provide two different logarithmic functions, an exponential function and a scale function:

FYL2X	Logarithm
FYL2XP1	Logarithm epsilon
F2XM1	Exponential
FSCALE	Scale

The FYL2X and FYL2XP1 instructions perform two different base 2 logarithmic operations. The FYL2X instruction computes  $(y * \log_2 x)$ . This operation permits the calculation of the log of any base using the following equation:

$$\log_b x = (1 / \log_2 b) * \log_2 x$$

The FYL2XP1 instruction computes  $(y * \log_2(x + 1))$ . This operation provides optimum accuracy for values of x that are close to 0.

The F2XM1 instruction computes  $(2^x - 1)$ . This instruction only operates on source values in the range  $-1.0$  to  $+1.0$ .

The FSCALE instruction multiplies the source operand by a power of 2.

### 8.3.10 Transcendental Instruction Accuracy

New transcendental instruction algorithms were incorporated into the IA-32 architecture beginning with the Pentium processors. These new algorithms (used in transcendental instructions FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) allow a higher level of accuracy than was possible in earlier IA-32 processors and x87 math coprocessors. The accuracy of these instructions is measured in terms of **units in the last place (ulp)**. For a given argument  $x$ , let  $f(x)$  and  $F(x)$  be the correct and computed (approximate) function values, respectively. The error in ulps is defined to be:

$$error = \left| \frac{f(x) - F(x)}{2^{k-63}} \right|$$

where  $k$  is an integer such that:

$$1 \leq 2^{-k} f(x) < 2.$$

With the Pentium processor and later IA-32 processors, the worst case error on transcendental functions is less than 1 ulp when rounding to the nearest (even) and less than 1.5 ulps when rounding in other modes. The functions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

However, for FSIN, FCOS, FSINCOS, and FPTAN which approximate periodic trigonometric functions, the previous statement about maximum ulp errors is true only when these instructions are applied to reduced argument (see Section 8.3.8, "Approximation of Pi"). This is due to the fact that only 66 significant bits are retained in the finite approximation Pi of the number  $\pi$  (3.1415926...), used internally for calculating the reduced argument in FSIN, FCOS, FSINCOS, and FPTAN. This approximation of  $\pi$  is not always sufficiently accurate for good argument reduction.

For single precision, the argument of FSIN, FCOS, FSINCOS, and FPTAN must exceed 200,000 radians in order for the error of the result to exceed 1 ulp when rounding to the nearest (even), or 1.5 ulps when rounding in other (directed) rounding modes.

For double and double-extended precision, the ulp errors will grow above these thresholds for arguments much smaller in magnitude. The ulp errors increase significantly when the argument approaches the value of  $\pi$  (or Pi) for FSIN, and when it approaches  $\pi/2$  (or Pi/2) for FCOS, FSINCOS, and FPTAN.

For all three IEEE precisions supported (32-bit single precision, 64-bit double precision, and 80-bit double-extended precision), applying FSIN, FCOS, FSINCOS, or FPTAN to arguments larger than a certain value can lead to reduced arguments (calculated internally) that are inaccurate or even very inaccurate in some cases. This leads to equally inaccurate approximations of the corresponding mathematical functions. In particular, arguments that are close to certain values will lose significance when reduced, leading to increased relative (and ulp) errors in the results of FSIN, FCOS, FSINCOS, and FPTAN. These values are:

- any non-zero multiple of  $\pi$  for FSIN,
- any multiple of  $\pi$ , plus  $\pi/2$  for FCOS, and
- any non-zero multiple of  $\pi/2$  for FSINCOS and FPTAN.

If the arguments passed to FSIN, FCOS, FSINCOS, and FPTAN are not close to these values then even the finite approximation Pi of  $\pi$  used internally for argument reduction will allow for results that have good accuracy.

Therefore, in order to avoid such errors it is recommended to perform accurate argument reduction in software, and to apply FSIN, FCOS, FSINCOS, and FPTAN to reduced arguments only. Regardless of the target precision (single, double, or double-extended), it is safe to reduce the argument to a value smaller in absolute value than about  $3\pi/4$  for FSIN, and smaller than about  $3\pi/8$  for FCOS, FSINCOS, and FPTAN.

The thresholds shown above are not exact. For example, accuracy measurements show that the double-extended precision result of FSIN will not have errors larger than 0.72 ulp for  $|x| < 2.82$  (so  $|x| < 3\pi/4$  will ensure good accuracy, as  $3\pi/4 < 2.82$ ). On the same interval, double precision results from FSIN will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

Likewise, the double-extended precision result of FCOS will not have errors larger than 0.82 ulp for  $|x| < 1.31$  (so  $|x| < 3\pi/8$  will ensure good accuracy, as  $3\pi/8 < 1.31$ ). On the same interval, double precision results from FCOS



will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

FSINCOS behaves similarly to FSIN and FCOS, combined as a pair.

Finally, the double-extended precision result of FPTAN will not have errors larger than 0.78 ulp for  $|x| < 1.25$  (so  $|x| < 3\pi/8$  will ensure good accuracy, as  $3\pi/8 < 1.25$ ). On the same interval, double precision results from FPTAN will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

A recommended alternative in order to avoid the accuracy issues that might be caused by FSIN, FCOS, FSINCOS, and FPTAN, is to use good quality mathematical library implementations of the sin, cos, sincos, and tan functions, for example those from the Intel® Math Library available in the Intel® Compiler.

The instructions FYL2X and FYL2XP1 are two operand instructions and are guaranteed to be within 1 ulp only when y equals 1. When y is not equal to 1, the maximum ulp error is always within 1.35 ulps in round to nearest mode. (For the two operand functions, monotonicity was proved by holding one of the operands constant.)

### 8.3.11 x87 FPU Control Instructions

The following instructions control the state and modes of operation of the x87 FPU. They also allow the status of the x87 FPU to be examined:

FINIT/FNINIT	Initialize x87 FPU
FLDCW	Load x87 FPU control word
FSTCW/FNSTCW	Store x87 FPU control word
FSTSW/FNSTSW	Store x87 FPU status word
FCLEX/FNCLEX	Clear x87 FPU exception flags
FLDENV	Load x87 FPU environment
FSTENV/FNSTENV	Store x87 FPU environment
FRSTOR	Restore x87 FPU state
FSAVE/FNSAVE	Save x87 FPU state
FINCSTP	Increment x87 FPU register stack pointer
FDECSTP	Decrement x87 FPU register stack pointer
FFREE	Free x87 FPU register
FNOP	No operation
WAIT/FWAIT	Check for and handle pending unmasked x87 FPU exceptions

The FINIT/FNINIT instructions initialize the x87 FPU and its internal registers to default values.

The FLDCW instructions loads the x87 FPU control word register with a value from memory. The FSTCW/FNSTCW and FSTSW/FNSTSW instructions store the x87 FPU control and status words, respectively, in memory (or for an FSTSW/FNSTSW instruction in a general-purpose register).

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions save the x87 FPU environment and state, respectively, in memory. The x87 FPU environment includes all the x87 FPU's control and status registers; the x87 FPU state includes the x87 FPU environment and the data registers in the x87 FPU register stack. (The FSAVE/FNSAVE instruction also initializes the x87 FPU to default values, like the FINIT/FNINIT instruction, after it saves the original state of the x87 FPU.)

The FLDENV and FRSTOR instructions load the x87 FPU environment and state, respectively, from memory into the x87 FPU. These instructions are commonly used when switching tasks or contexts.

The WAIT/FWAIT instructions are synchronization instructions. (They are actually mnemonics for the same opcode.) These instructions check the x87 FPU status word for pending unmasked x87 FPU exceptions. If any pending unmasked x87 FPU exceptions are found, they are handled before the processor resumes execution of the instructions (integer, floating-point, or system instruction) in the instruction stream. The WAIT/FWAIT instructions



are provided to allow synchronization of instruction execution between the x87 FPU and the processor's integer unit. See Section 8.6, "x87 FPU Exception Synchronization," for more information on the use of the WAIT/FWAIT instructions.

### 8.3.12 Waiting vs. Non-waiting Instructions

All of the x87 FPU instructions except a few special control instructions perform a wait operation (similar to the WAIT/FWAIT instructions), to check for and handle pending unmasked x87 FPU floating-point exceptions, before they perform their primary operation (such as adding two floating-point numbers). These instructions are called **waiting** instructions. Some of the x87 FPU control instructions, such as FSTSW/FNSTSW, have both a waiting and a non-waiting version. The waiting version (with the "F" prefix) executes a wait operation before it performs its primary operation; whereas, the non-waiting version (with the "FN" prefix) ignores pending unmasked exceptions.

Non-waiting instructions allow software to save the current x87 FPU state without first handling pending exceptions or to reset or reinitialize the x87 FPU without regard for pending exceptions.

#### NOTES

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for a non-waiting instruction to be interrupted prior to being executed to handle a pending x87 FPU exception.

When operating a P6 family, Pentium 4, or Intel Xeon processor in MS-DOS compatibility mode, non-waiting instructions can not be interrupted in this way.

### 8.3.13 Unsupported x87 FPU Instructions

The Intel 8087 instructions FENI and FDISI and the Intel 287 math coprocessor instruction FSETPM perform no function in the Intel 387 math coprocessor and later IA-32 processors. If these opcodes are detected in the instruction stream, the x87 FPU performs no specific operation and no internal x87 FPU states are affected.

## 8.4 X87 FPU FLOATING-POINT EXCEPTION HANDLING

The x87 FPU detects the six classes of exception conditions described in Section 4.9, "Overview of Floating-Point Exceptions":

- Invalid operation (#I), with two subclasses:
  - Stack overflow or underflow (#IS)
  - Invalid arithmetic operation (#IA)
- Denormalized operand (#D)
- Divide-by-zero (#Z)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

Each of the six exception classes has a corresponding flag bit in the x87 FPU status word and a mask bit in the x87 FPU control word (see Section 8.1.3, "x87 FPU Status Register," and Section 8.1.5, "x87 FPU Control Word," respectively). In addition, the exception summary (ES) flag in the status word indicates when one or more unmasked exceptions has been detected. The stack fault (SF) flag (also in the status word) distinguishes between the two types of invalid-operation exceptions.

The mask bits can be set with FLDCW, FRSTOR, or FXRSTOR; they can be read with either FSTCW/FNSTCW, FSAVE/FNSAVE, or FXSAVE. The flag bits can be read with the FSTSW/FNSTSW, FSAVE/FNSAVE, or FXSAVE instruction.

**NOTE**

Section 4.9.1, “Floating-Point Exception Conditions,” provides a general overview of how the IA-32 processor detects and handles the various classes of floating-point exceptions. This information pertains to x87 FPU as well as SSE/SSE2/SSE3 extensions.

The following sections give specific information about how the x87 FPU handles floating-point exceptions that are unique to the x87 FPU.

**8.4.1 Arithmetic vs. Non-arithmetic Instructions**

When dealing with floating-point exceptions, it is useful to distinguish between **arithmetic instructions** and **non-arithmetic instructions**. Non-arithmetic instructions have no operands or do not make substantial changes to their operands. Arithmetic instructions do make significant changes to their operands; in particular, they make changes that could result in floating-point exceptions being signaled. Table 8-9 lists the non-arithmetic and arithmetic instructions. It should be noted that some non-arithmetic instructions can signal a floating-point stack (fault) exception, but this exception is not the result of an operation on an operand.

**Table 8-9. Arithmetic and Non-arithmetic Instructions**

Non-arithmetic Instructions	Arithmetic Instructions
FABS	F2XM1
FCHS	FADD/FADDP
FCLEX	FBLD
FDECSTP	FBSTP
FFREE	FCOM/FCOMP/FCOMPP
FINCSTP	FCOS
FINIT/FNINIT	FDIV/FDIVP/FDIVR/FDIVRP
FLD (register-to-register)	FIADD
FLD (extended format from memory)	FICOM/FICOMP
FLD constant	FIDIV/FIDIVR
FLDCW	FILD
FLDENV	FIMUL
FNOP	FIST/FISTP <sup>1</sup>
FRSTOR	FISUB/FISUBR
FSAVE/FNSAVE	FLD (single and double)
FST/FSTP (register-to-register)	FMUL/FMULP
FSTP (extended format to memory)	FPATAN
FSTCW/FNSTCW	FPREM/FPREM1
FSTENV/FNSTENV	FPTAN
FSTSW/FNSTSW	FRNDINT
WAIT/FWAIT	FSCALE
FXAM	FSIN
FXCH	FSINCOS
	FSQRT
	FST/FSTP (single and double)
	FSUB/FSUBP/FSUBR/FSUBRP

**Table 8-9. Arithmetic and Non-arithmetic Instructions (Contd.)**

Non-arithmetic Instructions	Arithmetic Instructions
	FTST
	FUCOM/FUCOMP/FUCOMPP
	FXTRACT
	FYL2X/FYL2XP1

**NOTE:**

1. The FISTTP instruction in SSE3 is an arithmetic x87 FPU instruction.

## 8.5 X87 FPU FLOATING-POINT EXCEPTION CONDITIONS

The following sections describe the various conditions that cause a floating-point exception to be generated by the x87 FPU and the masked response of the x87 FPU when these conditions are detected. *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A & 2B*, list the floating-point exceptions that can be signaled for each floating-point instruction.

See Section 4.9.2, “Floating-Point Exception Priority,” for a description of the rules for exception precedence when more than one floating-point exception condition is detected for an instruction.

### 8.5.1 Invalid Operation Exception

The floating-point invalid-operation exception occurs in response to two sub-classes of operations:

- Stack overflow or underflow (#IS)
- Invalid arithmetic operand (#IA)

The flag for this exception (IE) is bit 0 of the x87 FPU status word, and the mask bit (IM) is bit 0 of the x87 FPU control word. The stack fault flag (SF) of the x87 FPU status word indicates the type of operation that caused the exception. When the SF flag is set to 1, a stack operation has resulted in stack overflow or underflow; when the flag is cleared to 0, an arithmetic instruction has encountered an invalid operand. Note that the x87 FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition. As a result, the state of the SF flag can be 1 following an invalid-arithmetic-operation exception, if it was not cleared from the last time a stack overflow or underflow condition occurred. See Section 8.1.3.4, “Stack Fault Flag,” for more information about the SF flag.

#### 8.5.1.1 Stack Overflow or Underflow Exception (#IS)

The x87 FPU tag word keeps track of the contents of the registers in the x87 FPU register stack (see Section 8.1.7, “x87 FPU Tag Word”). It then uses this information to detect two different types of stack faults:

- **Stack overflow** — An instruction attempts to load a non-empty x87 FPU register from memory. A non-empty register is defined as a register containing a zero (tag value of 01), a valid value (tag value of 00), or a special value (tag value of 10).
- **Stack underflow** — An instruction references an empty x87 FPU register as a source operand, including attempting to write the contents of an empty register to memory. An empty register has a tag value of 11.

### NOTES

The term stack overflow originates from the situation where the program has loaded (pushed) eight values from memory onto the x87 FPU register stack and the next value pushed on the stack causes a stack wraparound to a register that already contains a value.

The term stack underflow originates from the opposite situation. Here, a program has stored (popped) eight values from the x87 FPU register stack to memory and the next value popped from the stack causes stack wraparound to an empty register.

When the x87 FPU detects stack overflow or underflow, it sets the IE flag (bit 0) and the SF flag (bit 6) in the x87 FPU status word to 1. It then sets condition-code flag C1 (bit 9) in the x87 FPU status word to 1 if stack overflow occurred or to 0 if stack underflow occurred.

If the invalid-operation exception is masked, the x87 FPU returns the floating point, integer, or packed decimal integer indefinite value to the destination operand, depending on the instruction being executed. This value overwrites the destination register or memory location specified by the instruction.

If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software") and the top-of-stack pointer (TOP) and source operands remain unchanged.

### 8.5.1.2 Invalid Arithmetic Operand Exception (#IA)

The x87 FPU is able to detect a variety of invalid arithmetic operations that can be coded in a program. These operations are listed in Table 8-10. (This list includes the invalid operations defined in IEEE Standard 754.)

When the x87 FPU detects an invalid arithmetic operand, it sets the IE flag (bit 0) in the x87 FPU status word to 1. If the invalid-operation exception is masked, the x87 FPU then returns an indefinite value or QNaN to the destination operand and/or sets the floating-point condition codes as shown in Table 8-10. If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software") and the top-of-stack pointer (TOP) and source operands remain unchanged.

**Table 8-10. Invalid Arithmetic Operations and the Masked Responses to Them**

Condition	Masked Response
Any arithmetic operation on an operand that is in an unsupported format.	Return the QNaN floating-point indefinite value to the destination operand.
Any arithmetic operation on a SNaN.	Return a QNaN to the destination operand (see Table 4-7).
Ordered compare and test operations: one or both operands are NaNs.	Set the condition code flags (C0, C2, and C3) in the x87 FPU status word or the CF, PF, and ZF flags in the EFLAGS register to 111B (not comparable).
Addition: operands are opposite-signed infinities. Subtraction: operands are like-signed infinities.	Return the QNaN floating-point indefinite value to the destination operand.
Multiplication: $\infty$ by 0; 0 by $\infty$ .	Return the QNaN floating-point indefinite value to the destination operand.
Division: $\infty$ by $\infty$ ; 0 by 0.	Return the QNaN floating-point indefinite value to the destination operand.
Remainder instructions FPREM, FPREM1: modulus (divisor) is 0 or dividend is $\infty$ .	Return the QNaN floating-point indefinite; clear condition code flag C2 to 0.
Trigonometric instructions FCOS, FPTAN, FSIN, FSINCOS: source operand is $\infty$ .	Return the QNaN floating-point indefinite; clear condition code flag C2 to 0.
FSQRT: negative operand (except FSQRT (-0) = -0); FYL2X: negative operand (except FYL2X (-0) = - $\infty$ ); FYL2XP1: operand more negative than -1.	Return the QNaN floating-point indefinite value to the destination operand.
FBSTP: Converted value cannot be represented in 18 decimal digits, or source value is an SNaN, QNaN, $\pm\infty$ , or in an unsupported format.	Store packed BCD integer indefinite value in the destination operand.
FIST/FISTP: Converted value exceeds representable integer range of the destination operand, or source value is an SNaN, QNaN, $\pm\infty$ , or in an unsupported format.	Store integer indefinite value in the destination operand.
FXCH: one or both registers are tagged empty.	Load empty registers with the QNaN floating-point indefinite value, then perform the exchange.

Normally, when one or both of the source operands is a QNaN (and neither is an SNaN or in an unsupported format), an invalid-operand exception is not generated. An exception to this rule is most of the compare instructions (such as the FCOM and FCOMI instructions) and the floating-point to integer conversion instructions (FIST/FISTP and FBSTP). With these instructions, a QNaN source operand will generate an invalid-operand exception.

### 8.5.2 Denormal Operand Exception (#D)

The x87 FPU signals the denormal-operand exception under the following conditions:

- If an arithmetic instruction attempts to operate on a denormal operand (see Section 4.8.3.2, “Normalized and Denormalized Finite Numbers”).
- If an attempt is made to load a denormal single-precision or double-precision floating-point value into an x87 FPU register. (If the denormal value being loaded is a double extended-precision floating-point value, the denormal-operand exception is not reported.)

The flag (DE) for this exception is bit 1 of the x87 FPU status word, and the mask bit (DM) is bit 1 of the x87 FPU control word.

When a denormal-operand exception occurs and the exception is masked, the x87 FPU sets the DE flag, then proceeds with the instruction. The denormal operand in single- or double-precision floating-point format is automatically normalized when converted to the double extended-precision floating-point format. Subsequent operations will benefit from the additional precision of the internal double extended-precision floating-point format.

When a denormal-operand exception occurs and the exception is not masked, the DE flag is set and a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”). The top-of-stack pointer (TOP) and source operands remain unchanged.

For additional information about the denormal-operation exception, see Section 4.9.1.2, “Denormal Operand Exception (#D).”

### 8.5.3 Divide-By-Zero Exception (#Z)

The x87 FPU reports a floating-point divide-by-zero exception whenever an instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) for this exception is bit 2 of the x87 FPU status word, and the mask bit (ZM) is bit 2 of the x87 FPU control word. The FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, and FIDIVR instructions and the other instructions that perform division internally (FYL2X and FXTRACT) can report the divide-by-zero exception.

When a divide-by-zero exception occurs and the exception is masked, the x87 FPU sets the ZE flag and returns the values shown in Table 8-10. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”), and the top-of-stack pointer (TOP) and source operands remain unchanged.

**Table 8-11. Divide-By-Zero Conditions and the Masked Responses to Them**

Condition	Masked Response
Divide or reverse divide operation with a 0 divisor.	Returns an $\infty$ signed with the exclusive OR of the sign of the two operands to the destination operand.
FYL2X instruction.	Returns an $\infty$ signed with the opposite sign of the non-zero operand to the destination operand.
FXTRACT instruction.	ST(1) is set to $-\infty$ ; ST(0) is set to 0 with the same sign as the source operand.

### 8.5.4 Numeric Overflow Exception (#O)

The x87 FPU reports a floating-point numeric overflow exception (#O) whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that will fit into the floating-point format of the destination operand. (See Section 4.9.1.4, “Numeric Overflow Exception (#O),” for additional information about the numeric overflow exception.)

When using the x87 FPU, numeric overflow can occur on arithmetic operations where the result is stored in an x87 FPU data register. It can also occur on store floating-point operations (using the FST and FSTP instructions), where a within-range value in a data register is stored in memory in a single-precision or double-precision floating-point format. The numeric overflow exception cannot occur when storing values in an integer or BCD integer format. Instead, the invalid-arithmetic-operand exception is signaled.

The flag (OE) for the numeric-overflow exception is bit 3 of the x87 FPU status word, and the mask bit (OM) is bit 3 of the x87 FPU control word.

When a numeric-overflow exception occurs and the exception is masked, the x87 FPU sets the OE flag and returns one of the values shown in Table 4-10. The value returned depends on the current rounding mode of the x87 FPU (see Section 8.1.5.3, "Rounding Control Field").

The action that the x87 FPU takes when numeric overflow occurs and the numeric-overflow exception is not masked, depends on whether the instruction is supposed to store the result in memory or on the register stack.

- **Destination is a memory location** — The OE flag is set and a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software"). The top-of-stack pointer (TOP) and source and destination operands remain unchanged. Because the data in the stack is in double extended-precision format, the exception handler has the option either of re-executing the store instruction after proper adjustment of the operand or of rounding the significand on the stack to the destination's precision as the standard requires. The exception handler should ultimately store a value into the destination location in memory if the program is to continue.
- **Destination is the register stack** — The significand of the result is rounded according to current settings of the precision and rounding control bits in the x87 FPU control word and the exponent of the result is adjusted by dividing it by  $2^{24576}$ . (For instructions not affected by the precision field, the significand is rounded to double-extended precision.) The resulting value is stored in the destination operand. Condition code bit C1 in the x87 FPU status word (called in this situation the "round-up bit") is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the OE flag is set and a software exception handler is invoked. The scaling bias value 24,576 is equal to  $3 * 2^{13}$ . Biasing the exponent by 24,576 normally translates the number as nearly as possible to the middle of the double extended-precision floating-point exponent range so that, if desired, it can be used in subsequent scaled operations with less risk of causing further exceptions.

When using the FSCALE instruction, massive overflow can occur, where the result is too large to be represented, even with a bias-adjusted exponent. Here, if overflow occurs again, after the result has been biased, a properly signed  $\infty$  is stored in the destination operand.

## 8.5.5 Numeric Underflow Exception (#U)

The x87 FPU detects a potential floating-point numeric underflow condition whenever the result of an arithmetic instruction is non-zero and tiny; that is, the magnitude of the rounded result with unbounded exponent is non-zero and less than the smallest possible normalized, finite value that will fit into the floating-point format of the destination operand. (See Section 4.9.1.5, "Numeric Underflow Exception (#U)," for additional information about the numeric underflow exception.)

Like numeric overflow, numeric underflow can occur on arithmetic operations where the result is stored in an x87 FPU data register. It can also occur on store floating-point operations (with the FST and FSTP instructions), where a within-range value in a data register is stored in memory in the smaller single-precision or double-precision floating-point formats. A numeric underflow exception cannot occur when storing values in an integer or BCD integer format, because a value with magnitude less than 1 is always rounded to an integral value of 0 or 1, depending on the rounding mode in effect.

The flag (UE) for the numeric-underflow exception is bit 4 of the x87 FPU status word, and the mask bit (UM) is bit 4 of the x87 FPU control word.

When a numeric-underflow condition occurs and the exception is masked, the x87 FPU performs the operation described in Section 4.9.1.5, "Numeric Underflow Exception (#U)."

When the exception is not masked, the action of the x87 FPU depends on whether the instruction is supposed to store the result in a memory location or on the x87 FPU register stack.

- **Destination is a memory location** — (Can occur only with a store instruction.) The UE flag is set and a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”). The top-of-stack pointer (TOP) and source and destination operands remain unchanged, and no result is stored in memory.

Because the data in the stack is in double extended-precision format, the exception handler has the option either of re-exchanges the store instruction after proper adjustment of the operand or of rounding the significand on the stack to the destination's precision as the standard requires. The exception handler should ultimately store a value into the destination location in memory if the program is to continue.

- **Destination is the register stack** — The significand of the result is rounded according to current settings of the precision and rounding control bits in the x87 FPU control word and the exponent of the result is adjusted by multiplying it by  $2^{24576}$ . (For instructions not affected by the precision field, the significand is rounded to double extended precision.) The resulting value is stored in the destination operand. Condition code bit C1 in the x87 FPU status register (acting here as a “round-up bit”) is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the UE flag is set and a software exception handler is invoked. The scaling bias value 24,576 is the same as is used for the overflow exception and has the same effect, which is to translate the result as nearly as possible to the middle of the double extended-precision floating-point exponent range.

When using the FSCALE instruction, massive underflow can occur, where the magnitude of the result is too small to be represented, even with a bias-adjusted exponent. Here, if underflow occurs again after the result has been biased, a properly signed 0 is stored in the destination operand.

## 8.5.6 Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. (See Section 4.9.1.6, “Inexact-Result (Precision) Exception (#P),” for additional information about the numeric overflow exception.) Note that the transcendental instructions (FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) by nature produce inexact results.

The inexact-result exception flag (PE) is bit 5 of the x87 FPU status word, and the mask bit (PM) is bit 5 of the x87 FPU control word.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the x87 FPU handles the exception as describe in Section 4.9.1.6, “Inexact-Result (Precision) Exception (#P),” with one additional action. The C1 (round-up) bit in the x87 FPU status word is set to indicate whether the inexact result was rounded up (C1 is set) or “not rounded up” (C1 is cleared). In the “not rounded up” case, the least-significant bits of the inexact result are truncated so that the result fits in the destination format.

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the x87 FPU handles the exception as described in the previous paragraph and, in addition, invokes a software exception handler.

If an inexact result occurs in conjunction with numeric overflow or underflow, the x87 FPU carries out one of the following operations:

- If an inexact result occurs in conjunction with masked overflow or underflow, the OE or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions (see Section 8.5.4, “Numeric Overflow Exception (#O),” or Section 8.5.5, “Numeric Underflow Exception (#U)”). If the inexact result exception is unmasked, the x87 FPU also invokes a software exception handler.
- If an inexact result occurs in conjunction with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions (see Section 8.5.4, “Numeric Overflow Exception (#O),” or Section 8.5.5, “Numeric Underflow Exception (#U)”) and a software exception handler is invoked.

If an unmasked numeric overflow or underflow exception occurs and the destination operand is a memory location (which can happen only for a floating-point store), the inexact-result condition is not reported and the C1 flag is cleared.



## 8.6 X87 FPU EXCEPTION SYNCHRONIZATION

Because the integer unit and x87 FPU are separate execution units, it is possible for the processor to execute floating-point, integer, and system instructions concurrently. No special programming techniques are required to gain the advantages of concurrent execution. (Floating-point instructions are placed in the instruction stream along with the integer and system instructions.) However, concurrent execution can cause problems for floating-point exception handlers.

This problem is related to the way the x87 FPU signals the existence of unmasked floating-point exceptions. (Special exception synchronization is not required for masked floating-point exceptions, because the x87 FPU always returns a masked result to the destination operand.)

When a floating-point exception is unmasked and the exception condition occurs, the x87 FPU stops further execution of the floating-point instruction and signals the exception event. On the next occurrence of a floating-point instruction or a WAIT/FWAIT instruction in the instruction stream, the processor checks the ES flag in the x87 FPU status word for pending floating-point exceptions. If floating-point exceptions are pending, the x87 FPU makes an implicit call (traps) to the floating-point software exception handler. The exception handler can then execute recovery procedures for selected or all floating-point exceptions.

Synchronization problems occur in the time between the moment when the exception is signaled and when it is actually handled. Because of concurrent execution, integer or system instructions can be executed during this time. It is thus possible for the source or destination operands for a floating-point instruction that faulted to be overwritten in memory, making it impossible for the exception handler to analyze or recover from the exception.

To solve this problem, an exception synchronizing instruction (either a floating-point instruction or a WAIT/FWAIT instruction) can be placed immediately after any floating-point instruction that might present a situation where state information pertaining to a floating-point exception might be lost or corrupted. Floating-point instructions that store data in memory are prime candidates for synchronization. For example, the following three lines of code have the potential for exception synchronization problems:

```
FILD COUNT      ;Floating-point instruction
INC COUNT      ;Integer instruction
FSQRT          ;Subsequent floating-point instruction
```

In this example, the INC instruction modifies the source operand of the floating-point instruction, FILD. If an exception is signaled during the execution of the FILD instruction, the INC instruction would be allowed to overwrite the value stored in the COUNT memory location before the floating-point exception handler is called. With the COUNT variable modified, the floating-point exception handler would not be able to recover from the error.

Rearranging the instructions, as follows, so that the FSQRT instruction follows the FILD instruction, synchronizes floating-point exception handling and eliminates the possibility of the COUNT variable being overwritten before the floating-point exception handler is invoked.

```
FILD COUNT      ;Floating-point instruction
FSQRT          ;Subsequent floating-point instruction synchronizes
               ;any exceptions generated by the FILD instruction.
INC COUNT      ;Integer instruction
```

The FSQRT instruction does not require any synchronization, because the results of this instruction are stored in the x87 FPU data registers and will remain there, undisturbed, until the next floating-point or WAIT/FWAIT instruction is executed. To absolutely ensure that any exceptions emanating from the FSQRT instruction are handled (for example, prior to a procedure call), a WAIT instruction can be placed directly after the FSQRT instruction.

Note that some floating-point instructions (non-waiting instructions) do not check for pending unmasked exceptions (see Section 8.3.11, "x87 FPU Control Instructions"). They include the FNINIT, FNSTENV, FNSAVE, FNSTSW, FNSTCW, and FNCLEX instructions. When an FNINIT, FNSTENV, FNSAVE, or FNCLEX instruction is executed, all pending exceptions are essentially lost (either the x87 FPU status register is cleared or all exceptions are masked). The FNSTSW and FNSTCW instructions do not check for pending interrupts, but they do not modify the x87 FPU status and control registers. A subsequent "waiting" floating-point instruction can then handle any pending exceptions.



## 8.7 HANDLING X87 FPU EXCEPTIONS IN SOFTWARE

The x87 FPU in Pentium and later IA-32 processors provides two different modes of operation for invoking a software exception handler for floating-point exceptions: native mode and MS-DOS compatibility mode. The mode of operation is selected by CR0.NE[bit 5]. (See Chapter 2, "System Architecture Overview," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the NE flag.)

### 8.7.1 Native Mode

The native mode for handling floating-point exceptions is selected by setting CR0.NE[bit 5] to 1. In this mode, if the x87 FPU detects an exception condition while executing a floating-point instruction and the exception is unmasked (the mask bit for the exception is cleared), the x87 FPU sets the flag for the exception and the ES flag in the x87 FPU status word. It then invokes the software exception handler through the floating-point-error exception (#MF, exception vector 16), immediately before execution of any of the following instructions in the processor's instruction stream:

- The next floating-point instruction, unless it is one of the non-waiting instructions (FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV, and FNSAVE).
- The next WAIT/FWAIT instruction.
- The next MMX instruction.

If the next floating-point instruction in the instruction stream is a non-waiting instruction, the x87 FPU executes the instruction without invoking the software exception handler.

### 8.7.2 MS-DOS\* Compatibility Sub-mode

If CR0.NE[bit 5] is 0, the MS-DOS compatibility mode for handling floating-point exceptions is selected. In this mode, the software exception handler for floating-point exceptions is invoked externally using the processor's FERR#, INTR, and IGNNE# pins. This method of reporting floating-point errors and invoking an exception handler is provided to support the floating-point exception handling mechanism used in PC systems that are running the MS-DOS or Windows\* 95 operating system.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems, this approach also limits newer processors to operate with one logical processor active.

The MS-DOS compatibility mode is typically used as follows to invoke the floating-point exception handler:

1. If the x87 FPU detects an unmasked floating-point exception, it sets the flag for the exception and the ES flag in the x87 FPU status word.
2. If the IGNNE# pin is deasserted, the x87 FPU then asserts the FERR# pin either immediately, or else delayed (deferred) until just before the execution of the next waiting floating-point instruction or MMX instruction. Whether the FERR# pin is asserted immediately or delayed depends on the type of processor, the instruction, and the type of exception.
3. If a preceding floating-point instruction has set the exception flag for an unmasked x87 FPU exception, the processor freezes just before executing the **next** WAIT instruction, waiting floating-point instruction, or MMX instruction. Whether the FERR# pin was asserted at the preceding floating-point instruction or is just now being asserted, the freezing of the processor assures that the x87 FPU exception handler will be invoked before the new floating-point (or MMX) instruction gets executed.
4. The FERR# pin is connected through external hardware to IRQ13 of a cascaded, programmable interrupt controller (PIC). When the FERR# pin is asserted, the PIC is programmed to generate an interrupt 75H.
5. The PIC asserts the INTR pin on the processor to signal the interrupt 75H.
6. The BIOS for the PC system handles the interrupt 75H by branching to the interrupt 02H (NMI) interrupt handler.
7. The interrupt 02H handler determines if the interrupt is the result of an NMI interrupt or a floating-point exception.

8. If a floating-point exception is detected, the interrupt 02H handler branches to the floating-point exception handler.

If the `IGNNE#` pin is asserted, the processor ignores floating-point error conditions. This pin is provided to inhibit floating-point exceptions from being generated while the floating-point exception handler is servicing a previously signaled floating-point exception.

Appendix D, "Guidelines for Writing SIMD Floating-Point Exception Handlers," describes the MS-DOS compatibility mode in much greater detail. This mode is somewhat more complicated in the Intel486 and Pentium processor implementations, as described in Appendix D.

### 8.7.3 Handling x87 FPU Exceptions in Software

Section 4.9.3, "Typical Actions of a Floating-Point Exception Handler," shows actions that may be carried out by a floating-point exception handler. The state of the x87 FPU can be saved with the `FSTENV/FNSTENV` or `FSAVE/FNSAVE` instructions (see Section 8.1.10, "Saving the x87 FPU's State with `FSTENV/FNSTENV` and `FSAVE/FNSAVE`").

If the faulting floating-point instruction is followed by one or more non-floating-point instructions, it may not be useful to re-execute the faulting instruction. See Section 8.6, "x87 FPU Exception Synchronization," for more information on synchronizing floating-point exceptions.

In cases where the handler needs to restart program execution with the faulting instruction, the `IRET` instruction cannot be used directly. The reason for this is that because the exception is not generated until the next floating-point or `WAIT/FWAIT` instruction following the faulting floating-point instruction, the return instruction pointer on the stack may not point to the faulting instruction. To restart program execution at the faulting instruction, the exception handler must obtain a pointer to the instruction from the saved x87 FPU state information, load it into the return instruction pointer location on the stack, and then execute the `IRET` instruction.



#### 4. Updates to Chapter 13, Volume 1

Change bars and green text show changes to Chapter 13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

---

Changes to this chapter: Added a clarifying paragraph to section 13.8.3, "XRSTOR and the Init and Modified Optimizations" and section 13.12, "Operation of XRSTORS".

# CHAPTER 13

## MANAGING STATE USING THE XSAVE FEATURE SET

---

The XSAVE feature set extends the functionality of the FXSAVE and FXRSTOR instructions (see Section 10.5, “FXSAVE and FXRSTOR Instructions”) by supporting the saving and restoring of processor state in addition to the x87 execution environment (**x87 state**) and the registers used by the streaming SIMD extensions (**SSE state**).

The **XSAVE feature set** comprises eight instructions. XGETBV and XSETBV allow software to read and write the extended control register XCR0, which controls the operation of the XSAVE feature set. XSAVE, XSAVEOPT, XSAVEC, and XSAVES are four instructions that save processor state to memory; XRSTOR and XRSTORS are corresponding instructions that load processor state from memory. XGETBV, XSAVE, XSAVEOPT, XSAVEC, and XRSTOR can be executed at any privilege level; XSETBV, XSAVES, and XRSTORS can be executed only if CPL = 0. In addition to XCR0, the XSAVES and XRSTORS instructions are controlled also by the IA32\_XSS MSR (index DA0H).

The XSAVE feature set organizes the state that manages into **state components**. Operation of the instructions is based on **state-component bitmaps** that have the same format as XCR0 and as the IA32\_XSS MSR: each bit corresponds to a state component. Section 13.1 discusses these state components and bitmaps in more detail.

Section 13.2 describes how the processor enumerates support for the XSAVE feature set and for **XSAVE-enabled features** (those features that require use of the XSAVE feature set for their enabling). Section 13.3 explains how software can enable the XSAVE feature set and XSAVE-enabled features.

The XSAVE feature set allows saving and loading processor state from a region of memory called an **XSAVE area**. Section 13.4 presents details of the XSAVE area and its organization. Each XSAVE-managed state component is associated with a section of the XSAVE area. Section 13.5 describes in detail each of the XSAVE-managed state components.

Section 13.7 through Section 13.12 describe the operation of XSAVE, XRSTOR, XSAVEOPT, XSAVEC, XSAVES, and XRSTORS, respectively.

### 13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS

The XSAVE feature set supports the saving and restoring of **state components**, each of which is a discrete set of processor registers (or parts of registers). In general, each such state component corresponds to a particular CPU feature. Such a feature is **XSAVE-supported**. Some XSAVE-supported features use registers in multiple XSAVE-managed state components.

The XSAVE feature set organizes the state components of the XSAVE-supported features using **state-component bitmaps**. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. The following bits are defined in state-component bitmaps:

- Bit 0 corresponds to the state component used for the x87 FPU execution environment (**x87 state**). See Section 13.5.1.
- Bit 1 corresponds to the state component used for registers used by the streaming SIMD extensions (**SSE state**). See Section 13.5.2.
- Bit 2 corresponds to the state component used for the additional register state used by the Intel® Advanced Vector Extensions (**AVX state**). See Section 13.5.3.
- Bits 4:3 correspond to the two state components used for the additional register state used by Intel® Memory Protection Extensions (**MPX state**):
  - State component 3 is used for the 4 128-bit bounds registers BND0–BND3 (**BNDREGS state**).
  - State component 4 is used for the 64-bit user-mode MPX configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (**BNDCSR state**).
- Bits 7:5 correspond to the three state components used for the additional register state used by Intel® Advanced Vector Extensions 512 (**AVX-512 state**):
  - State component 5 is used for the 8 64-bit opmask registers k0–k7 (**opmask state**).

- State component 6 is used for the upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0\_H–ZMM15\_H (**ZMM\_Hi256 state**).
- State component 7 is used for the 16 512-bit registers ZMM16–ZMM31 (**Hi16\_ZMM state**).
- Bit 8 corresponds to the state component used for the Intel Processor Trace MSRs (**PT state**).
- Bit 9 corresponds to the state component used for the protection-key feature’s register PKRU (**PKRU state**). See Section 13.5.7.
- Bits 12:11 correspond to the two state components used for the additional register state used by Control-Flow Enforcement Technology (**CET state**):
  - State component 11 is used for the 2 MSRs controlling user-mode functionality for CET (**CET\_U state**).
  - State component 12 is used for the 3 MSRs containing shadow-stack pointers for privilege levels 0–2 (**CET\_S state**).
- Bit 13 corresponds to the state component used for an MSR used to control hardware duty cycling (**HDC state**). See Section 13.5.9.
- Bit 16 corresponds to the state component used for an MSR used to control hardware P-states (**HWP state**). See Section 13.5.10.

Bit 10, bits 15:14, and bits in the range 62:17 are not currently defined in state-component bitmaps and are reserved for future expansion. As individual state components are defined using those bits, additional sub-sections will be updated within Section 13.5 over time. Bit 63 is used for special functionality in some bitmaps and does not correspond to any state component.

The state component corresponding to bit *i* of state-component bitmaps is called **state component *i***. Thus, x87 state is state component 0; SSE state is state component 1; AVX state is state component 2; MPX state comprises state components 3–4; AVX-512 state comprises state components 5–7; PT state is state component 8; PKRU state is state component 9; CET state comprises state components 11–12; HDC state is state component 13; and HWP state is state component 16.

The XSAVE feature set uses state-component bitmaps in multiple ways. Most of the instructions use an implicit operand (in EDX:EAX), called the **instruction mask**, which is the state-component bitmap that specifies the state components on which the instruction operates.

Some state components are **user state components**, and they can be managed by the entire XSAVE feature set. Other state components are **supervisor state components**, and they can be managed only by XSAVES and XRSTORS. The state components corresponding to bit 9 and to bits in the range 7:0 are user state components; those corresponding to bit 8, to bits in the range 13:11, and to bit 16 are supervisor state components.

Extended control register XCR0 contains a state-component bitmap that specifies the user state components that software has enabled the XSAVE feature set to manage. If the bit corresponding to a state component is clear in XCR0, instructions in the XSAVE feature set will not operate on that state component, regardless of the value of the instruction mask.

The IA32\_XSS MSR (index DA0H) contains a state-component bitmap that specifies the supervisor state components that software has enabled XSAVES and XRSTORS to manage (XSAVE, XSAVEC, XSAVEOPT, and XRSTOR cannot manage supervisor state components). If the bit corresponding to a state component is clear in the IA32\_XSS MSR, XSAVES and XRSTORS will not operate on that state component, regardless of the value of the instruction mask.

Some XSAVE-supported features can be used only if XCR0 has been configured so that the features’ state components can be managed by the XSAVE feature set. (This applies only to features with user state components.) Such state components and features are **XSAVE-enabled**. In general, the processor will not modify (or allow modification of) the registers of a state component of an XSAVE-enabled feature if the bit corresponding to that state component is clear in XCR0. (If software clears such a bit in XCR0, the processor preserves the corresponding state component.) If an XSAVE-enabled feature has not been fully enabled in XCR0, execution of any instruction defined for that feature causes an invalid-opcode exception (#UD).

As will be explained in Section 13.3, the XSAVE feature set is enabled only if CR4.OSXSAVE[bit 18] = 1. If CR4.OSXSAVE = 0, the processor treats XSAVE-enabled state features and their state components as if all bits in XCR0 were clear; the state components cannot be modified and the features’ instructions cannot be executed.

The state components for x87 state, for SSE state, for PT state, for PKRU state, for CET state, for HDC state, and for HWP state are XSAVE-managed but the corresponding features are not XSAVE-enabled. Processors allow modi-

fication of this state, as well as execution of x87 FPU instructions and SSE instructions and use of Intel Processor Trace, protection keys, CET, hardware duty cycling, and hardware P-states regardless of the value of CR4.OSXSAVE and XCR0.

## 13.2 ENUMERATION OF CPU SUPPORT FOR XSAVE INSTRUCTIONS AND XSAVE-SUPPORTED FEATURES

A processor enumerates support for the XSAVE feature set and for features supported by that feature set using the CPUID instruction. The following items provide specific details:

- CPUID.1:ECX.XSAVE[bit 26] enumerates general support for the XSAVE feature set:
  - If this bit is 0, the processor does not support any of the following instructions: XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV; the processor provides no further enumeration through CPUID function 0DH (see below).
  - If this bit is 1, the processor supports the following instructions: XGETBV, XRSTOR, XSAVE, and XSETBV.<sup>1</sup> Further enumeration is provided through CPUID function 0DH.
- CR4.OSXSAVE can be set to 1 if and only if CPUID.1:ECX.XSAVE[bit 26] is enumerated as 1.
- CPUID function 0DH enumerates details of CPU support through a set of sub-functions. Software selects a specific sub-function by the value placed in the ECX register. The following items provide specific details:
  - CPUID function 0DH, sub-function 0.
    - EDX:EAX is a bitmap of all the user state components that can be managed using the XSAVE feature set. A bit can be set in XCR0 if and only if the corresponding bit is set in this bitmap. Every processor that supports the XSAVE feature set will set EAX[0] (x87 state) and EAX[1] (SSE state).  
If  $EAX[i] = 1$  (for  $1 < i < 32$ ) or  $EDX[i-32] = 1$  (for  $32 \leq i < 63$ ), sub-function  $i$  enumerates details for state component  $i$  (see below).
    - ECX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components supported by this processor.
    - EBX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components corresponding to bits currently set in XCR0.
  - CPUID function 0DH, sub-function 1.
    - EAX[0] enumerates support for the XSAVEOPT instruction. The instruction is supported if and only if this bit is 1. If  $EAX[0] = 0$ , execution of XSAVEOPT causes an invalid-opcode exception (#UD).
    - EAX[1] enumerates support for **compaction extensions** to the XSAVE feature set. The following are supported if this bit is 1:
      - The compacted format of the extended region of XSAVE areas (see Section 13.4.3).
      - The XSAVEC instruction. If  $EAX[1] = 0$ , execution of XSAVEC causes a #UD.
      - Execution of the compacted form of XRSTOR (see Section 13.8).
    - EAX[2] enumerates support for execution of XGETBV with  $ECX = 1$ . This allows software to determine the state of the init optimization. See Section 13.6.
    - EAX[3] enumerates support for XSAVES, XRSTORS, and the IA32\_XSS MSR. If  $EAX[3] = 0$ , execution of XSAVES or XRSTORS causes a #UD; an attempt to access the IA32\_XSS MSR using RDMSR or WRMSR causes a general-protection exception (#GP). Every processor that supports a supervisor state component sets EAX[3]. Every processor that sets EAX[3] (XSAVES, XRSTORS, IA32\_XSS) will also set EAX[1] (the compaction extensions).
    - EAX[31:4] are reserved.

1. If CPUID.1:ECX.XSAVE[bit 26] = 1, XGETBV and XSETBV may be executed with  $ECX = 0$  (to read and write XCR0). Any support for execution of these instructions with other values of ECX is enumerated separately.

- EBX enumerates the size (in bytes) required by the XSAVES instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0 | IA32\_XSS.
- EDX:ECX is a bitmap of all the supervisor state components that can be managed by XSAVES and XRSTORS. A bit can be set in the IA32\_XSS MSR if and only if the corresponding bit is set in this bitmap.

### NOTE

In summary, the XSAVE feature set supports state component  $i$  ( $0 \leq i < 63$ ) if one of the following is true: (1)  $i < 32$  and  $\text{CPUID}(\text{EAX}=0\text{DH}, \text{ECX}=0): \text{EAX}[i] = 1$ ; (2)  $i \geq 32$  and  $\text{CPUID}(\text{EAX}=0\text{DH}, \text{ECX}=0): \text{EAX}[i-32] = 1$ ; (3)  $i < 32$  and  $\text{CPUID}(\text{EAX}=0\text{DH}, \text{ECX}=1): \text{ECX}[i] = 1$ ; or (4)  $i \geq 32$  and  $\text{CPUID}(\text{EAX}=0\text{DH}, \text{ECX}=1): \text{EDX}[i-32] = 1$ . The XSAVE feature set supports user state component  $i$  if (1) or (2) holds; if (3) or (4) holds, state component  $i$  is a supervisor state component and support is limited to XSAVES and XRSTORS.

- CPUID function 0DH, sub-function  $i$  ( $i > 1$ ). This sub-function enumerates details for state component  $i$ . If the XSAVE feature set supports state component  $i$  (see note above), the following items provide specific details:
  - EAX enumerates the size (in bytes) required for state component  $i$ .
  - If state component  $i$  is a user state component, EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section used for state component  $i$ . (This offset applies only when the standard format for the extended region of the XSAVE area is being used; see Section 13.4.3.)
  - If state component  $i$  is a supervisor state component, EBX returns 0.
  - If state component  $i$  is a user state component, ECX[0] return 0; if state component  $i$  is a supervisor state component, ECX[0] returns 1.
  - The value returned by ECX[1] indicates the alignment of state component  $i$  when the compacted format of the extended region of an XSAVE area is used (see Section 13.4.3). If ECX[1] returns 0, state component  $i$  is located immediately following the preceding state component; if ECX[1] returns 1, state component  $i$  is located on the next 64-byte boundary following the preceding state component.
  - ECX[31:2] and EDX return 0.

If the XSAVE feature set does not support state component  $i$ , sub-function  $i$  returns 0 in EAX, EBX, ECX, and EDX.

## 13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES

Software enables the XSAVE feature set by setting CR4.OSXSAVE[bit 18] to 1 (e.g., with the MOV to CR4 instruction). If this bit is 0, execution of any of XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV causes an invalid-opcode exception (#UD).

When CR4.OSXSAVE = 1 and CPL = 0, executing the XSETBV instruction with ECX = 0 writes the 64-bit value in EDX:EAX to XCR0 (EAX is written to XCR0[31:0] and EDX to XCR0[63:32]). (Execution of the XSETBV instruction causes a general-protection fault — #GP — if CPL > 0.) The following items provide details regarding individual bits in XCR0:

- XCR0[0] is associated with x87 state (see Section 13.5.1). XCR0[0] is always 1. It has that value coming out of RESET. Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[0] is 0.
- XCR0[1] is associated with SSE state (see Section 13.5.2). Software can use the XSAVE feature set to manage SSE state only if XCR0[1] = 1. The value of XCR0[1] in no way determines whether software can execute SSE instructions (these instructions can be executed even if XCR0[1] = 0).  
XCR0[1] is 0 coming out of RESET. As noted in Section 13.2, every processor that supports the XSAVE feature set allows software to set XCR0[1].
- XCR0[2] is associated with AVX state (see Section 13.5.3). Software can use the XSAVE feature set to manage AVX state only if XCR0[2] = 1. In addition, software can execute AVX instructions only if CR4.OSXSAVE = XCR0[2] = 1. Otherwise, any execution of an AVX instruction causes an invalid-opcode exception (#UD).



XCR0[2] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[2] if and only if  $\text{CPUID}(\text{EAX}=0\text{DH},\text{ECX}=0):\text{EAX}[2] = 1$ . In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if  $\text{ECX} = 0$  and  $\text{EAX}[2:1]$  has the value 10b; that is, software cannot enable the XSAVE feature set for AVX state but not for SSE state.

As noted in Section 13.1, the processor will preserve AVX state unmodified if software clears XCR0[2]. However, clearing XCR0[2] while AVX state is not in its initial configuration may cause SSE instructions to incur a power and performance penalty. See Section 13.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State Components" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[4:3] are associated with MPX state (see Section 13.5.4). Software can use the XSAVE feature set to manage MPX state only if  $\text{XCR0}[4:3] = 11\text{b}$ . In addition, MPX instructions operate as defined only if  $\text{CR4.OSXSAVE} = 1$  and  $\text{XCR0}[4:3] = 11\text{b}$ . Otherwise, execution of an MPX instruction causes no operation (as a NOP instruction); in addition, executions of CALL, RET, JMP, and Jcc do not initialize the bounds registers, and they ignore any F2H (BND) prefix.<sup>1</sup>

XCR0[4:3] have value 00b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[4:3] to 11b if and only if  $\text{CPUID}(\text{EAX}=0\text{DH},\text{ECX}=0):\text{EAX}[4:3] = 11\text{b}$ . In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if  $\text{ECX} = 0$ ,  $\text{EAX}[4:3]$  is neither 00b nor 11b; that is, software can enable the XSAVE feature set for MPX state only if it does so for both state components.

As noted in Section 13.1, the processor will preserve MPX state unmodified if software clears XCR0[4:3].

- XCR0[7:5] are associated with AVX-512 state (see Section 13.5.5). Software can use the XSAVE feature set to manage AVX-512 state only if  $\text{XCR0}[7:5] = 111\text{b}$ . In addition, software can execute AVX-512 instructions only if  $\text{CR4.OSXSAVE} = 1$  and  $\text{XCR0}[7:5] = 111\text{b}$ . Otherwise, any execution of an AVX-512 instruction causes an invalid-opcode exception (#UD).

XCR0[7:5] have value 000b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[7:5] to 111b if and only if  $\text{CPUID}(\text{EAX}=0\text{DH},\text{ECX}=0):\text{EAX}[7:5] = 111\text{b}$ . In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if  $\text{ECX} = 0$ ,  $\text{EAX}[7:5]$  is not 000b, and any bit is clear in  $\text{EAX}[2:1]$  or  $\text{EAX}[7:5]$ ; that is, software can enable the XSAVE feature set for AVX-512 state only if it does so for all three state components, and only if it also does so for AVX state and SSE state. This implies that the value of XCR0[7:5] is always either 000b or 111b.

As noted in Section 13.1, the processor will preserve AVX-512 state unmodified if software clears XCR0[7:5]. However, clearing XCR0[7:5] while AVX-512 state is not in its initial configuration may cause SSE and AVX instructions to incur a power and performance penalty. See Section 13.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State Components" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[9] is associated with PKRU state (see Section 13.5.7). Software can use the XSAVE feature set to manage PKRU state only if  $\text{XCR0}[9] = 1$ . The value of XCR0[9] in no way determines whether software can use protection keys or execute other instructions that access PKRU state (these instructions can be executed even if  $\text{XCR0}[9] = 0$ ).

XCR0[9] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[9] if and only if  $\text{CPUID}(\text{EAX}=0\text{DH},\text{ECX}=0):\text{EAX}[9] = 1$ .

- XCR0[63:10] and XCR0[8] are reserved.<sup>2</sup> Executing the XSETBV instruction causes a general-protection fault (#GP) if  $\text{ECX} = 0$  and any corresponding bit in  $\text{EDX}:\text{EAX}$  is not 0. These bits in XCR0 are all 0 coming out of RESET.

Software operating with  $\text{CPL} > 0$  may need to determine whether the XSAVE feature set and certain XSAVE-enabled features have been enabled. If  $\text{CPL} > 0$ , execution of the MOV from CR4 instruction causes a general-protection fault (#GP). The following alternative mechanisms allow software to discover the enabling of the XSAVE feature set regardless of CPL:

- 
1. Prior to the introduction of MPX, the opcodes defining MPX instructions operated as NOP, and the CALL, RET, JMP, and Jcc instructions ignored any F2H prefix.
  2. Bit 8 and bits 13:11 correspond to supervisor state components. Since bits can be set in XCR0 only for user state components, those bits of XCR0 must be 0.

- The value of CR4.OSXSAVE is returned in CPUID.1:ECX.OSXSAVE[bit 27]. If software determines that CPUID.1:ECX.OSXSAVE = 1, the processor supports the XSAVE feature set and the feature set has been enabled in CR4.
- Executing the XGETBV instruction with ECX = 0 returns the value of XCR0 in EDX:EAX. XGETBV can be executed if CR4.OSXSAVE = 1 (if CPUID.1:ECX.OSXSAVE = 1), regardless of CPL.

Thus, software can use the following algorithm to determine the support and enabling for the XSAVE feature set:

1. Use CPUID to discover the value of CPUID.1:ECX.OSXSAVE.
  - If the bit is 0, either the XSAVE feature set is not supported by the processor or has not been enabled by software. Either way, the XSAVE feature set is not available, nor are XSAVE-enabled features such as AVX.
  - If the bit is 1, the processor supports the XSAVE feature set — including the XGETBV instruction — and it has been enabled by software. The XSAVE feature set can be used to manage x87 state (because XCR0[0] is always 1). Software requiring more detailed information can go on to the next step.
2. Execute XGETBV with ECX = 0 to discover the value of XCR0. If XCR0[1] = 1, the XSAVE feature set can be used to manage SSE state. If XCR0[2] = 1, the XSAVE feature set can be used to manage AVX state and software can execute AVX instructions. If XCR0[4:3] is 11b, the XSAVE feature set can be used to manage MPX state and software can execute MPX instructions. If XCR0[7:5] is 111b, the XSAVE feature set can be used to manage AVX-512 state and software can execute AVX-512 instructions. If XCR0[9] = 1, the XSAVE feature set can be used to manage PKRU state.

The IA32\_XSS MSR (with MSR index DA0H) is zero coming out of RESET. If CR4.OSXSAVE = 1, CPUID.(EAX=0DH,ECX=1):EAX[3] = 1, and CPL = 0, executing the WRMSR instruction with ECX = DA0H writes the 64-bit value in EDX:EAX to the IA32\_XSS MSR (EAX is written to IA32\_XSS[31:0] and EDX to IA32\_XSS[63:32]). The following items provide details regarding individual bits in the IA32\_XSS MSR:

- IA32\_XSS[8] is associated with PT state (see Section 13.5.6). Software can use XSAVES and XRSTORS to manage PT state only if IA32\_XSS[8] = 1. The value of IA32\_XSS[8] does not determine whether software can use Intel Processor Trace (the feature can be used even if IA32\_XSS[8] = 0).
- IA32\_XSS[12:11] are associated with CET state (see Section 13.5.8), IA32\_XSS[11] with CET\_U state and IA32\_XSS[12] with CET\_S state. Software can use the XSAVES and XRSTORS to manage CET\_U state (respectively, CET\_S state) only if IA32\_XSS[11] = 1 (respectively, IA32\_XSS[12] = 1). The value of IA32\_XSS[12:11] does not determine whether software can use CET (the feature can be used even if either of IA32\_XSS[12:11] is 0).
- IA32\_XSS[13] is associated with HDC state (see Section 13.5.9). Software can use XSAVES and XRSTORS to manage HDC state only if IA32\_XSS[13] = 1. The value of IA32\_XSS[13] does not determine whether software can use hardware duty cycling (the feature can be used even if IA32\_XSS[13] = 0).
- IA32\_XSS[16] is associated with HWP state (see Section 13.5.10). Software can use XSAVES and XRSTORS to manage HWP state only if IA32\_XSS[16] = 1. The value of IA32\_XSS[16] does not determine whether software can use hardware P-states (the feature can be used even if IA32\_XSS[16] = 0).
- IA32\_XSS[63:17], IA32\_XSS[15:14], IA32\_XSS[10:9] and IA32\_XSS[7:0] are reserved.<sup>1</sup> Executing the WRMSR instruction causes a general-protection fault (#GP) if ECX = DA0H and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

The IA32\_XSS MSR is 0 coming out of RESET.

There is no mechanism by which software operating with CPL > 0 can discover the value of the IA32\_XSS MSR.

## 13.4 XSAVE AREA

The XSAVE feature set includes instructions that save and restore the XSAVE-managed state components to and from memory: XSAVE, XSAVEOPT, XSAVEC, and XSAVES (for saving); and XRSTOR and XRSTORS (for restoring). The processor organizes the state components in a region of memory called an **XSAVE area**. Each of the save and

---

1. Bit 9 and bits 7:0 correspond to user state components. Since bits can be set in the IA32\_XSS MSR only for supervisor state components, those bits of the MSR must be 0.

restore instructions takes a memory operand that specifies the 64-byte aligned base address of the XSAVE area on which it operates.

Every XSAVE area has the following format:

- The **legacy region**. The legacy region of an XSAVE area comprises the 512 bytes starting at the area's base address. It is used to manage the state components for x87 state and SSE state. The legacy region is described in more detail in Section 13.4.1.
- The **XSAVE header**. The XSAVE header of an XSAVE area comprises the 64 bytes starting at an offset of 512 bytes from the area's base address. The XSAVE header is described in more detail in Section 13.4.2.
- The **extended region**. The extended region of an XSAVE area starts at an offset of 576 bytes from the area's base address. It is used to manage the state components other than those for x87 state and SSE state. The extended region is described in more detail in Section 13.4.3. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 and IA32\_XSS (see Section 13.3).

### 13.4.1 Legacy Region of an XSAVE Area

The legacy region of an XSAVE area comprises the 512 bytes starting at the area's base address. It has the same format as the FXSAVE area (see Section 10.5.1). The XSAVE feature set uses the legacy area for x87 state (state component 0) and SSE state (state component 1). Table 13-1 illustrates the format of the first 416 bytes of the legacy region of an XSAVE area.

**Table 13-1. Format of the Legacy Region of an XSAVE Area**

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
FIP[63:48] or reserved	FCS or FIP[47:32]	FIP[31:0]		FOP	Rsvd.	FTW	FSW	FCW	<b>0</b>
MXCSR_MASK		MXCSR		FDP[63:48] or reserved	FDS or FDP[47:32]		FDP[31:0]		<b>16</b>
Reserved			ST0/MM0						<b>32</b>
Reserved			ST1/MM1						<b>48</b>
Reserved			ST2/MM2						<b>64</b>
Reserved			ST3/MM3						<b>80</b>
Reserved			ST4/MM4						<b>96</b>
Reserved			ST5/MM5						<b>112</b>
Reserved			ST6/MM6						<b>128</b>
Reserved			ST7/MM7						<b>144</b>
			XMM0						<b>160</b>
			XMM1						<b>176</b>
			XMM2						<b>192</b>
			XMM3						<b>208</b>
			XMM4						<b>224</b>
			XMM5						<b>240</b>
			XMM6						<b>256</b>
			XMM7						<b>272</b>
			XMM8						<b>288</b>
			XMM9						<b>304</b>
			XMM10						<b>320</b>

**Table 13-1. Format of the Legacy Region of an XSAVE Area (Contd.) (Contd.)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
XMM11															<b>336</b>	
XMM12															<b>352</b>	
XMM13															<b>368</b>	
XMM14															<b>384</b>	
XMM15															<b>400</b>	

The x87 state component comprises bytes 23:0 and bytes 159:32. The SSE state component comprises bytes 31:24 and bytes 415:160. The XSAVE feature set does not use bytes 511:416; bytes 463:416 are reserved. Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the legacy region of an XSAVE area.

### 13.4.2 XSAVE Header

The XSAVE header of an XSAVE area comprises the 64 bytes starting at offset 512 from the area’s base address:

- Bytes 7:0 of the XSAVE header is a state-component bitmap (see Section 13.1) called **XSTATE\_BV**. It identifies the state components in the XSAVE area.
- Bytes 15:8 of the XSAVE header is a state-component bitmap called **XCOMP\_BV**. It is used as follows:
  - XCOMP\_BV[63] indicates the format of the extended region of the XSAVE area (see Section 13.4.3). If it is clear, the standard format is used. If it is set, the compacted format is used; XCOMP\_BV[62:0] provide format specifics as specified in Section 13.4.3.
  - XCOMP\_BV[63] determines which form of the XRSTOR instruction is used. If the bit is set, the compacted form is used; otherwise, the standard form is used. See Section 13.8.
  - All bits in XCOMP\_BV should be 0 if the processor does not support the compaction extensions to the XSAVE feature set.
- Bytes 63:16 of the XSAVE header are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the XSAVE header of an XSAVE area.

### 13.4.3 Extended Region of an XSAVE Area

The extended region of an XSAVE area starts at byte offset 576 from the area’s base address. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 | IA32\_XSS (see Section 13.3).

The XSAVE feature set uses the extended area for each state component *i*, where *i* ≥ 2. The following state components are currently supported in the extended area: state component 2 contains AVX state; state components 5–7 contain AVX-512 state; and state component 9 contains PKRU state.

The extended region of the an XSAVE area may have one of two formats. The **standard format** is supported by all processors that support the XSAVE feature set; the **compacted format** is supported by those processors that support the compaction extensions to the XSAVE feature set (see Section 13.2). Bit 63 of the XCOMP\_BV field in the XSAVE header (see Section 13.4.2) indicates which format is used.

The following items describe the two possible formats of the extended region:

- **Standard format.** Each state component *i* (*i* ≥ 2) is located at the byte offset from the base address of the XSAVE area enumerated in CPUID.(EAX=0DH,ECX=*i*):EBX. (CPUID.(EAX=0DH,ECX=*i*):EAX enumerates the number of bytes required for state component *i*).
- **Compacted format.** Each state component *i* (*i* ≥ 2) is located at a byte offset from the base address of the XSAVE area based on the XCOMP\_BV field in the XSAVE header:

- If  $XCOMP\_BV[i] = 0$ , state component  $i$  is not in the XSAVE area.
- If  $XCOMP\_BV[i] = 1$ , state component  $i$  is located at a byte offset  $location_I$  from the base address of the XSAVE area, where  $location_I$  is determined by the following items:
  - If  $XCOMP\_BV[j] = 0$  for every  $j$ ,  $2 \leq j < i$ ,  $location_I$  is 576. (This item applies if  $i$  is the first bit set in bits 62:2 of the  $XCOMP\_BV$ ; it implies that state component  $i$  is located at the beginning of the extended region.)
  - Otherwise, let  $j$ ,  $2 \leq j < i$ , be the greatest value such that  $XCOMP\_BV[j] = 1$ . Then  $location_I$  is determined by the following values:  $location_j$ ;  $size_j$ , as enumerated in  $CPUID.(EAX=0DH,ECX=j):EAX$ ; and the value of  $align_I$ , as enumerated in  $CPUID.(EAX=0DH,ECX=i):ECX[1]$ :
    - If  $align_I = 0$ ,  $location_I = location_j + size_j$ . (This item implies that state component  $i$  is located immediately following the preceding state component whose bit is set in  $XCOMP\_BV$ .)
    - If  $align_I = 1$ ,  $location_I = \text{ceiling}(location_j + size_j, 64)$ . (This item implies that state component  $i$  is located on the next 64-byte boundary following the preceding state component whose bit is set in  $XCOMP\_BV$ .)

## 13.5 XSAVE-MANAGED STATE

The section provides details regarding how the XSAVE feature set interacts with the various XSAVE-managed state components.

Unless otherwise state, the state pertaining to a particular state component is saved beginning at byte 0 of the section of the XSAVE are corresponding to that state component.

### 13.5.1 x87 State

Instructions in the XSAVE feature set can manage the same state of the x87 FPU execution environment (**x87 state**) that can be managed using the FXSAVE and FXRSTOR instructions. They organize all x87 state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the x87 state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 1:0, 3:2, 7:6. These are used for the x87 FPU Control Word (FCW), the x87 FPU Status Word (FSW), and the x87 FPU Opcode (FOP), respectively.
- Byte 4 is used for an abridged version of the x87 FPU Tag Word (FTW). The following items describe its usage:
  - For each  $j$ ,  $0 \leq j \leq 7$ , XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 0 into bit  $j$  of byte 4 if x87 FPU data register  $STj$  has an empty tag; otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 1 into bit  $j$  of byte 4.
  - For each  $j$ ,  $0 \leq j \leq 7$ , XRSTOR and XRSTORS establish the tag value for x87 FPU data register  $STj$  as follows. If bit  $j$  of byte 4 is 0, the tag for  $STj$  in the tag register for that data register is marked empty (11B); otherwise, the x87 FPU sets the tag for  $STj$  based on the value being loaded into that register (see below).
- Bytes 15:8 are used as follows:
  - If the instruction has no REX prefix, or if  $REX.W = 0$ :
    - Bytes 11:8 are used for bits 31:0 of the x87 FPU Instruction Pointer Offset (FIP).
    - If  $CPUID.(EAX=07H,ECX=0H):EBX[\text{bit } 13] = 0$ , bytes 13:12 are used for x87 FPU Instruction Pointer Selector (FCS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H, and XRSTOR and XRSTORS ignore them.
    - Bytes 15:14 are not used.
  - If the instruction has a REX prefix with  $REX.W = 1$ , bytes 15:8 are used for the full 64 bits of FIP.
- Bytes 23:16 are used as follows:
  - If the instruction has no REX prefix, or if  $REX.W = 0$ :
    - Bytes 19:16 are used for bits 31:0 of the x87 FPU Data Pointer Offset (FDP).

- If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 0, bytes 21:20 are used for x87 FPU Data Pointer Selector (FDS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H; and XRSTOR and XRSTORS ignore them.
- Bytes 23:22 are not used.
  - If the instruction has a REX prefix with REX.W = 1, bytes 23:16 are used for the full 64 bits of FDP.
- Bytes 31:24 are used for SSE state (see Section 13.5.2).
- Bytes 159:32 are used for the registers ST0–ST7 (MM0–MM7). Each of the 8 register is allocated a 128-bit region, with the low 80 bits used for the register and the upper 48 bits unused.

x87 state is XSAVE-managed but the x87 FPU feature is not XSAVE-enabled. The XSAVE feature set can operate on x87 state only if the feature set is enabled (CR4.OSXSAVE = 1).<sup>1</sup> Software can otherwise use x87 state even if the XSAVE feature set is not enabled.

### 13.5.2 SSE State

Instructions in the XSAVE feature set can manage the registers used by the streaming SIMD extensions (**SSE state**) just as the FXSAVE and FXRSTOR instructions do. They organize all SSE state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the SSE state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 23:0 are used for x87 state (see Section 13.5.1).
- Bytes 27:24 are used for the MXCSR register. XRSTOR and XRSTORS generate general-protection faults (#GP) in response to attempts to set any of the reserved bits of the MXCSR register.<sup>2</sup>
- Bytes 31:28 are used for the MXCSR\_MASK value. XRSTOR and XRSTORS ignore this field.
- Bytes 159:32 are used for x87 state.
- Bytes 287:160 are used for the registers XMM0–XMM7.
- Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify these bytes; executions of XRSTOR and XRSTORS outside 64-bit mode do not update XMM8–XMM15. See Section 13.13.

SSE state is XSAVE-managed but the SSE feature is not XSAVE-enabled. The XSAVE feature set can operate on SSE state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage SSE state (XCR0[1] = 1). Software can otherwise use SSE state even if the XSAVE feature set is not enabled or has not been configured to manage SSE state.

### 13.5.3 AVX State

The register state used by the Intel<sup>®</sup> Advanced Vector Extensions (AVX) comprises the MXCSR register and 16 256-bit vector registers called YMM0–YMM15. The low 128 bits of each register YMM<sub>*i*</sub> is identical to the SSE register XMM<sub>*i*</sub>. Thus, the new state register state added by AVX comprises the upper 128 bits of the registers YMM0–YMM15. These 16 128-bit values are denoted YMM0\_H–YMM15\_H and are collectively called **AVX state**.

As noted in Section 13.1, the XSAVE feature set manages AVX state as user state component 2. Thus, AVX state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=2):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for AVX state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=2):EAX enumerates the size (in bytes) required for AVX state.

The XSAVE feature set partitions YMM0\_H–YMM15\_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 127:0 of the AVX-state section are used for YMM0\_H–YMM7\_H. Bytes 255:128 are used for

1. The processor ensures that XCR0[0] is always 1.

2. While MXCSR and MXCSR\_MASK are part of SSE state, their treatment by the XSAVE feature set is not the same as that of the XMM registers. See Section 13.7 through Section 13.11 for details.



YMM8\_H–YMM15\_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 255:128; executions of XRSTOR and XRSTORS outside 64-bit mode do not update YMM8\_H–YMM15\_H. See Section 13.13. In general, bytes  $16i+15:16i$  are used for YMM $i$ \_H (for  $0 \leq i \leq 15$ ).

AVX state is XSAVE-managed and the AVX feature is XSAVE-enabled. The XSAVE feature set can operate on AVX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX state (XCR0[2] = 1). AVX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX state.

### 13.5.4 MPX State

The register state used by the Intel® Memory Protection Extensions (MPX) comprises the 4 128-bit bounds registers BND0–BND3 (**BNDREGS state**); and the 64-bit user-mode configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (collectively, **BNDCSR state**). Together, these two user state components compose **MPX state**.

As noted in Section 13.1, the XSAVE feature set manages MPX state as state components 3–4. Thus, MPX state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **BNDREGS state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=3):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for BNDREGS state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=3):EAX enumerates the size (in bytes) required for BNDREGS state. The BNDREGS section is used for the 4 128-bit bound registers BND0–BND3, with bytes  $16i+15:16i$  being used for BND $i$ .
- **BNDCSR state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=4):EBX enumerates the offset of the section of the extended region of the XSAVE area used for BNDCSR state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=4):EAX enumerates the size (in bytes) required for BNDCSR state. In the BNDCSR section, bytes 7:0 are used for BNDCFGU and bytes 15:8 are used for BNDSTATUS.

Both components of MPX state are XSAVE-managed and the MPX feature is XSAVE-enabled. The XSAVE feature set can operate on MPX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage MPX state (XCR0[4:3] = 11b). MPX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage MPX state.

### 13.5.5 AVX-512 State

The register state used by the Intel® Advanced Vector Extensions 512 (AVX-512) comprises the MXCSR register, the 8 64-bit opmask registers k0–k7, and 32 512-bit vector registers called ZMM0–ZMM31. For each  $i$ ,  $0 \leq i \leq 15$ , the low 256 bits of register ZMM $i$  is identical to the AVX register YMM $i$ . Thus, the new state register state added by AVX comprises the following user state components:

- The opmask registers, collectively called **opmask state**.
- The upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0\_H–ZMM15\_H and are collectively called **ZMM\_Hi256 state**.
- The 16 512-bit registers ZMM16–ZMM31, collectively called **Hi16\_ZMM state**.

Together, these three state components compose **AVX-512 state**.

As noted in Section 13.1, the XSAVE feature set manages AVX-512 state as state components 5–7. Thus, AVX-512 state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **Opmask state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=5):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for opmask state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes)

required for opmask state. The opmask section is used for the 8 64-bit opmask registers k0–k7, with bytes  $8i+7:8i$  being used for  $ki$ .

- **ZMM\_Hi256 state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=6):EBX enumerates the offset of the section of the extended region of the XSAVE area used for ZMM\_Hi256 state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for ZMM\_Hi256 state.

The XSAVE feature set partitions ZMM0\_H–ZMM15\_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 255:0 of the ZMM\_Hi256-state section are used for ZMM0\_H–ZMM7\_H.

Bytes 511:256 are used for ZMM8\_H–ZMM15\_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 511:256; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM8\_H–ZMM15\_H. See Section 13.13. In general, bytes  $32i+31:32i$  are used for ZMM $i$ \_H (for  $0 \leq i \leq 15$ ).

- **Hi16\_ZMM state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=7):EBX enumerates the offset of the section of the extended region of the XSAVE area used for Hi16\_ZMM state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=7):EAX enumerates the size (in bytes) required for Hi16\_ZMM state.

The XSAVE feature set accesses Hi16\_ZMM state only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify the Hi16\_ZMM section; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM16–ZMM31. See Section 13.13. In general, bytes  $64(i-16)+63:64(i-16)$  are used for ZMM $i$  (for  $16 \leq i \leq 31$ ).

All three components of AVX-512 state are XSAVE-managed and the AVX-512 feature is XSAVE-enabled. The XSAVE feature set can operate on AVX-512 state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX-512 state (XCRO[7:5] = 111b). AVX-512 instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX-512 state.

## 13.5.6 PT State

The register state used by Intel Processor Trace (**PT state**) comprises the following 9 MSRs: IA32\_RTIT\_CTL, IA32\_RTIT\_OUTPUT\_BASE, IA32\_RTIT\_OUTPUT\_MASK\_PTRS, IA32\_RTIT\_STATUS, IA32\_RTIT\_CR3\_MATCH, IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B, IA32\_RTIT\_ADDR1\_A, and IA32\_RTIT\_ADDR1\_B.<sup>1</sup>

As noted in Section 13.1, the XSAVE feature set manages PT state as supervisor state component 8. Thus, PT state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=8):EAX enumerates the size (in bytes) required for PT state. The MSRs are each allocated 8 bytes in the state component in the order given above. Thus, IA32\_RTIT\_CTL is at byte offset 0, IA32\_RTIT\_OUTPUT\_BASE at byte offset 8, etc. Any locations in the state component at or beyond byte offset 72 are reserved.

PT state is XSAVE-managed but Intel Processor Trace is not XSAVE-enabled. The XSAVE feature set can operate on PT state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PT state (IA32\_XSS[8] = 1). Software can otherwise use Intel Processor Trace and access its MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage PT state.

The following items describe special treatment of PT state by the XSAVES and XRSTORS instructions:

- If XSAVES saves PT state, the instruction clears IA32\_RTIT\_CTL.TraceEn (bit 0) after saving the value of the IA32\_RTIT\_CTL MSR and before saving any other PT state. If XSAVES causes a fault or a VM exit, it restores IA32\_RTIT\_CTL.TraceEn to its original value.
- If XSAVES saves PT state, the instruction saves zeroes in the reserved portions of the state component.
- If XRSTORS would restore (or initialize) PT state and IA32\_RTIT\_CTL.TraceEn = 1, the instruction causes a general-protection exception (#GP) before modifying PT state.
- If XRSTORS causes an exception or a VM exit, it does so before any modification to IA32\_RTIT\_CTL.TraceEn (even if it has loaded other PT state).

1. These MSRs might not be supported by every processor that supports Intel Processor Trace. Software can use the CPUID instruction to discover which are supported; see Section 31.3.1, “Detection of Intel Processor Trace and Capability Enumeration,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.



### 13.5.7 PKRU State

The register state used by the protection-key feature (**PKRU state**) is the 32-bit PKRU register. As noted in Section 13.1, the XSAVE feature set manages PKRU state as user state component 9. Thus, PKRU state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=9):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for PKRU state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=9):EAX enumerates the size (in bytes) required for PKRU state. The XSAVE feature set uses bytes 3:0 of the PK-state section for the PKRU register.

PKRU state is XSAVE-managed but the protection-key feature is not XSAVE-enabled. The XSAVE feature set can operate on PKRU state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PKRU state (XCR0[9] = 1). Software can otherwise use protection keys and access PKRU state even if the XSAVE feature set is not enabled or has not been configured to manage PKRU state.

The value of the PKRU register determines the access rights for user-mode linear addresses. (See Section 4.6, "Access Rights," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.) The access rights that pertain to an execution of the XRSTOR and XRSTORS instructions are determined by the value of the register before the execution and not by any value that the execution might load into the PKRU register.

### 13.5.8 CET State

The register state used by Control-Flow Enforcement Technology (CET) comprises the two 64-bit MSRs (IA32\_U\_CET and IA32\_PL3\_SSP) that manage CET when CPL = 3 (**CET\_U state**); and the three 64-bit MSRs (IA32\_PL0\_SSP–IA32\_PL2\_SSP) that manage CET when CPL < 3 (**CET\_S state**). Together, these two user state components compose **CET state**.<sup>1</sup>

As noted in Section 13.1, the XSAVE feature set manages CET state as supervisor state components 11–23. Thus, CET state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **CET\_U state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=11):EAX enumerates the size (in bytes) required for CET\_U state. The CET\_U section is used for the 64-bit MSRs IA32\_U\_CET and IA32\_PL3\_SSP, with bytes 7:0 being used for IA32\_U\_CET and bytes 15:8 being used for IA32\_PL3\_SSP.
- **CET\_S state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=12):EAX enumerates the size (in bytes) required for CET\_S state. The CET\_S section is used for the three 64-bit MSRs IA32\_PL0\_SSP–IA32\_PL2\_SSP, with bytes  $8i+7:8i$  being used for IA32\_PL<sub>i</sub>\_SSP.

The two components of CET state are XSAVE-managed and CET is not XSAVE-enabled. The XSAVE feature set can operate on CET\_U state (respectively, CET\_S state) only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage CET\_U state (respectively, CET\_S state) by setting IA32\_XSS[11] (respectively, IA32\_XSS[12]). Software can otherwise use CET and access the CET MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage CET state.

### 13.5.9 HDC State

The register state used by hardware duty cycling (**HDC state**) comprises the IA32\_PM\_CTL1 MSR.

As noted in Section 13.1, the XSAVE feature set manages HDC state as supervisor state component 13. Thus, HDC state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=13):EAX enumerates the size (in bytes) required for HDC state. The IA32\_PM\_CTL1 MSR is allocated 8 bytes at byte offset 0 in the state component.

HDC state is XSAVE-managed but hardware duty cycling is not XSAVE-enabled. The XSAVE feature set can operate on HDC state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HDC state

1. The IA32\_S\_CET and IA32\_INTERRUPT\_SSP\_TABLE\_ADDR MSRs also control CET when CPL < 3. However, they are not managed by the XSAVE feature set and are thus not considered in this chapter.

(IA32\_XSS[13] = 1). Software can otherwise use hardware duty cycling and access the IA32\_PM\_CTL1 MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HDC state.

### 13.5.10 HWP State

The register state used by hardware P-states (**HWP state**) comprises the IA32\_HWP\_REQUEST MSR.

As noted in Section 13.1, the XSAVE feature set manages HWP state as supervisor state component 16. Thus, HWP state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=16):EAX enumerates the size (in bytes) required for HWP state. The IA32\_HWP\_REQUEST MSR is allocated 8 bytes at byte offset 0 in the state component.

HWP state is XSAVE-managed but the hardware P-states feature is not XSAVE-enabled. The XSAVE feature set can operate on HWP state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HWP state (IA32\_XSS[16] = 1). Software can otherwise use hardware P-states and access the IA32\_HWP\_REQUEST MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HWP state.

## 13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE

The XSAVEOPT, XSAVEC, and XSAVES instructions use two optimizations to reduce the amount of data that they write to memory. They avoid writing data for any state component known to be in its initial configuration (the **init optimization**). In addition, if either XSAVEOPT or XSAVES is using the same XSAVE area as that used by the most recent execution of XRSTOR or XRSTORS, it may avoid writing data for any state component whose configuration is known not to have been modified since then (the **modified optimization**). (XSAVE does not use these optimizations, and XSAVEC does not use the modified optimization.) The operation of XSAVEOPT, XSAVEC, and XSAVES are described in more detail in Section 13.9 through Section 13.11.

A processor can support the init and modified optimizations with special hardware that tracks the state components that might benefit from those optimizations. Other implementations might not include such hardware; such a processor would always consider each such state component as not in its initial configuration and as modified since the last execution of XRSTOR or XRSTORS.

The following notation describes the state of the init and modified optimizations:

- XINUSE denotes the state-component bitmap corresponding to the init optimization. If  $XINUSE[i] = 0$ , state component  $i$  is known to be in its initial configuration; otherwise  $XINUSE[i] = 1$ . It is possible for  $XINUSE[i]$  to be 1 even when state component  $i$  is in its initial configuration. On a processor that does not support the init optimization,  $XINUSE[i]$  is always 1 for every value of  $i$ .

Executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. Such an execution of XGETBV always sets EAX[1] to 1 if XCR0[1] = 1 and MXCSR does not have its RESET value of 1F80H. Section 13.2 explains how software can determine whether a processor supports this use of XGETBV.

- XMODIFIED denotes the state-component bitmap corresponding to the modified optimization. If  $XMODIFIED[i] = 0$ , state component  $i$  is known not to have been modified since the most recent execution of XRSTOR or XRSTORS; otherwise  $XMODIFIED[i] = 1$ . It is possible for  $XMODIFIED[i]$  to be 1 even when state component  $i$  has not been modified since the most recent execution of XRSTOR or XRSTORS. On a processor that does not support the modified optimization,  $XMODIFIED[i]$  is always 1 for every value of  $i$ .

A processor that implements the modified optimization saves information about the most recent execution of XRSTOR or XRSTORS in a quantity called **XRSTOR\_INFO**, a 4-tuple containing the following: (1) the CPL; (2) whether the logical processor was in VMX non-root operation; (3) the linear address of the XSAVE area; and (4) the XCOMP\_BV field in the XSAVE area. An execution of XSAVEOPT or XSAVES uses the modified optimization only if that execution corresponds to XRSTOR\_INFO on these four parameters.

This mechanism implies that, depending on details of the operating system, the processor might determine that an execution of XSAVEOPT by one user application corresponds to an earlier execution of XRSTOR by a different application. For this reason, Intel recommends the application software not use the XSAVEOPT instruction.

The following items specify the initial configuration each state component (for the purposes of defining the XINUSE bitmap):

- **x87 state.** x87 state is in its initial configuration if the following all hold: FCW is 037FH; FSW is 0000H; FTW is FFFFH; FCS and FDS are each 0000H; FIP and FDP are each 00000000\_00000000H; each of ST0–ST7 is 0000\_00000000\_00000000H.
- **SSE state.** In 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM15 is 0. Outside 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM7 is 0. XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update XMM8–XMM15. (See Section 13.13.)
- **AVX state.** In 64-bit mode, AVX state is in its initial configuration if each of YMM0\_H–YMM15\_H is 0. Outside 64-bit mode, AVX state is in its initial configuration if each of YMM0\_H–YMM7\_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update YMM8\_H–YMM15\_H. (See Section 13.13.)
- **BNDREGS state.** BNDREGS state is in its initial configuration if the value of each of BND0–BND3 is 0.
- **BNDCSR state.** BNDCSR state is in its initial configuration if BNDCFGU and BNDCSR each has value 0.
- **Opmask state.** Opmask state is in its initial configuration if each of the opmask registers k0–k7 is 0.
- **ZMM\_Hi256 state.** In 64-bit mode, ZMM\_Hi256 state is in its initial configuration if each of ZMM0\_H–ZMM15\_H is 0. Outside 64-bit mode, ZMM\_Hi256 state is in its initial configuration if each of ZMM0\_H–ZMM7\_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM8\_H–ZMM15\_H. (See Section 13.13.)
- **Hi16\_ZMM state.** In 64-bit mode, Hi16\_ZMM state is in its initial configuration if each of ZMM16–ZMM31 is 0. Outside 64-bit mode, Hi16\_ZMM state is always in its initial configuration. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM31–ZMM31. (See Section 13.13.)
- **PT state.** PT state is in its initial configuration if each of the 9 MSRs is 0.
- **PKRU state.** PKRU state is in its initial configuration if the value of the PKRU is 0.
- **PT state.** PT state is in its initial configuration if each of the 9 MSRs is 0.
- **CET\_U state.** CET\_U state is in its initial configuration if both of the MSRs are 0.
- **CET\_S state.** CET\_S state is in its initial configuration if each of the three MSRs is 0.
- **HDC state.** HDC state is in its initial configuration if the value of the IA32\_PM\_CTL1 MSR is 1.
- **HWP state.** HWP state is in its initial configuration if the value of the IA32\_HWP\_REQUEST MSR is 8000FF01H.

## 13.7 OPERATION OF XSAVE

The XSAVE instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be saved.

The following conditions cause execution of the XSAVE instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVE reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting XSTATE\_BV[*i*] ( $0 \leq i \leq 63$ ) as follows:

- If RFBM[*i*] = 0, XSTATE\_BV[*i*] is not changed.
- If RFBM[*i*] = 1, XSTATE\_BV[*i*] is set to the value of XINUSE[*i*]. Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.

- If state component  $i$  is in its initial configuration,  $XINUSE[i]$  may be either 0 or 1, and  $XSTATE\_BV[i]$  may be written with either 0 or 1.  
 $XINUSE[1]$  pertains only to the state of the XMM registers and not to MXCSR. Thus,  $XSTATE\_BV[1]$  may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
- If state component  $i$  is not in its initial configuration,  $XINUSE[i] = 1$  and  $XSTATE\_BV[i]$  is written with 1.  
 (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVE instruction does not write any part of the XSAVE header other than the XSTATE\_BV field; in particular, it does **not** write to the XCOMP\_BV field.

Execution of XSAVE saves into the XSAVE area those state components corresponding to bits that are set in RFBM. State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVE instruction always uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register and MXCSR\_MASK are part of SSE state (see Section 13.5.2) and are thus associated with RFBM[1]. However, the XSAVE instruction also saves these values when RFBM[2] = 1 (even if RFBM[1] = 0).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

## 13.8 OPERATION OF XRSTOR

The XRSTOR instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.
- If  $CR0.TS[\text{bit } 3]$  is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

After checking for these faults, the XRSTOR instruction reads the XCOMP\_BV field in the XSAVE area's XSAVE header (see Section 13.4.2). If  $XCOMP\_BV[63] = 0$ , the **standard form of XRSTOR** is executed (see Section 13.8.1); otherwise, the **compacted form of XRSTOR** is executed (see Section 13.8.2).<sup>2</sup>

See Section 13.2 for details of how to determine whether the compacted form of XRSTOR is supported.

### 13.8.1 Standard Form of XRSTOR

The standard form of XRSTOR performs additional fault checking. Either of the following conditions causes a general-protection exception (#GP):

- The XSTATE\_BV field of the XSAVE header sets a bit that is not set in XCR0.
- Bytes 23:8 of the XSAVE header are not all 0 (this implies that all bits in XCOMP\_BV are 0).<sup>3</sup>

---

1. If  $CR0.AM = 1$ ,  $CPL = 3$ , and  $EFLAGS.AC = 1$ , an alignment-check exception (#AC) may occur instead of #GP.  
 2. If the processor does not support the compacted form of XRSTOR, it may execute the standard form of XRSTOR without first reading the XCOMP\_BV field. A processor supports the compacted form of XRSTOR only if it enumerates  $CPUID.(EAX=0DH,ECX=1):EAX[1]$  as 1.  
 3. Bytes 63:24 of the XSAVE header are also reserved. Software should ensure that bytes 63:16 of the XSAVE header are all 0 in any XSAVE area. (Bytes 15:8 should also be 0 if the XSAVE area is to be used on a processor that does not support the compaction extensions to the XSAVE feature set.)

If none of these conditions cause a fault, the processor updates each state component  $i$  for which  $RFBM[i] = 1$ . XRSTOR updates state component  $i$  based on the value of bit  $i$  in the XSTATE\_BV field of the XSAVE header:

- If  $XSTATE\_BV[i] = 0$ , the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.  
The initial configuration of state component 1 pertains only to the XMM registers and not to MXCSR. See below for the treatment of MXCSR.
- If  $XSTATE\_BV[i] = 1$ , the state component is loaded with data from the XSAVE area. See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.  
State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the standard form of XRSTOR uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register is part of state component 1, SSE state (see Section 13.5.2). However, the standard form of XRSTOR loads the MXCSR register from memory whenever the  $RFBM[1]$  (SSE) or  $RFBM[2]$  (AVX) is set, regardless of the values of  $XSTATE\_BV[1]$  and  $XSTATE\_BV[2]$ . The standard form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

## 13.8.2 Compacted Form of XRSTOR

The compacted form of XRSTOR performs additional fault checking. Any of the following conditions causes a #GP:

- The XCOMP\_BV field of the XSAVE header sets a bit in the range 62:0 that is not set in XCR0.
- The XSTATE\_BV field of the XSAVE header sets a bit (including bit 63) that is not set in XCOMP\_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component  $i$  for which  $RFBM[i] = 1$ . XRSTOR updates state component  $i$  based on the value of bit  $i$  in the XSTATE\_BV field of the XSAVE header:

- If  $XSTATE\_BV[i] = 0$ , the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.  
If  $XSTATE\_BV[1] = 0$ , the compacted form XRSTOR initializes MXCSR to 1F80H. (This differs from the standard form of XRSTOR, which loads MXCSR from the XSAVE area whenever either  $RFBM[1]$  or  $RFBM[2]$  is set.)  
State component  $i$  is set to its initial configuration as indicated above if  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 0$  — **even if XCOMP\_BV[i] = 0**. This is true for all values of  $i$ , including 0 (x87 state) and 1 (SSE state).
- If  $XSTATE\_BV[i] = 1$ , the state component is loaded with data from the XSAVE area.<sup>1</sup> See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.  
State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the compacted form of the XRSTOR instruction uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if  $RFBM[1] = XSTATE\_BV[1] = 1$ . The compacted form of XRSTOR does not consider  $RFBM[2]$  (AVX) when determining whether to update MXCSR. (This is a difference from the standard form of XRSTOR.) The compacted form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

## 13.8.3 XRSTOR and the Init and Modified Optimizations

Execution of the XRSTOR instruction causes the processor to update its tracking for the init and modified optimizations (see Section 13.6). The following items provide details:

- The processor updates its tracking for the init optimization as follows:

---

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and  $XSTATE\_BV[i]$  is 1, then  $XCOMP\_BV[i]$  is also 1.



- If  $RFBM[i] = 0$ ,  $XINUSE[i]$  is not changed.
- If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 0$ , state component  $i$  may be tracked as init;  $XINUSE[i]$  may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the init optimization for state component  $i$ ; a processor that does not do so implicitly maintains  $XINUSE[i] = 1$  at all times.)
- If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ , state component  $i$  is tracked as not init;  $XINUSE[i]$  is set to 1.
- The processor updates its tracking for the modified optimization and records information about the XRSTOR execution for future interaction with the XSAVEOPT and XSAVES instructions (see Section 13.9 and Section 13.11) as follows:
  - If  $RFBM[i] = 0$ , state component  $i$  is tracked as modified;  $XMODIFIED[i]$  is set to 1.
  - If  $RFBM[i] = 1$ , state component  $i$  may be tracked as unmodified;  $XMODIFIED[i]$  may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the modified optimization for state component  $i$ ; a processor that does not do so implicitly maintains  $XMODIFIED[i] = 1$  at all times.)
  - $XRSTOR\_INFO$  is set to the 4-tuple  $\langle w, x, y, z \rangle$ , where  $w$  is the CPL (0);  $x$  is 1 if the logical processor is in VMX non-root operation and 0 otherwise;  $y$  is the linear address of the XSAVE area; and  $z$  is  $XCOMP\_BV$ . In particular, the standard form of XRSTOR always sets  $z$  to all zeroes, while the compacted form of XRSTORS never does so (because it sets at least bit 63 to 1).

Note that, if RFBM is entirely zero (e.g., because the instruction mask in EDX:EAX is zero), no state components are modified, the XINUSE bitmap is not modified, and all bits are set in the XMODIFIED bitmap. Thus, if EDX:EAX was zero for the most recent execution of XRSTOR, an execution of XSAVEOPT or XSAVES will identify all state components as modified and will thus not use the modified optimization.

## 13.9 OPERATION OF XSAVEOPT

The operation of XSAVEOPT is similar to that of XSAVE. Unlike XSAVE, XSAVEOPT uses the init optimization (by which it may omit saving state components that are in their initial configuration) and the modified optimization (by which it may omit saving state components that have not been modified since the last execution of XRSTOR); see Section 13.6. See Section 13.2 for details of how to determine whether XSAVEOPT is supported.

The XSAVEOPT instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical (bitwise) AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be saved.

The following conditions cause execution of the XSAVEOPT instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.
- If  $CR0.TS[\text{bit } 3]$  is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVEOPT reads the  $XSTATE\_BV$  field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting  $XSTATE\_BV[i]$  ( $0 \leq i \leq 63$ ) as follows:

- If  $RFBM[i] = 0$ ,  $XSTATE\_BV[i]$  is not changed.
- If  $RFBM[i] = 1$ ,  $XSTATE\_BV[i]$  is set to the value of  $XINUSE[i]$ . Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If the state component is in its initial configuration,  $XINUSE[i]$  may be either 0 or 1, and  $XSTATE\_BV[i]$  may be written with either 0 or 1.  
 $XINUSE[1]$  pertains only to the state of the XMM registers and not to MXCSR. Thus,  $XSTATE\_BV[1]$  may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If the state component is not in its initial configuration,  $XSTATE\_BV[i]$  is written with 1.

1. If  $CR0.AM = 1$ ,  $CPL = 3$ , and  $EFLAGS.AC = 1$ , an alignment-check exception (#AC) may occur instead of #GP.

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEOPT instruction does not write any part of the XSAVE header other than the XSTATE\_BV field; in particular, it does not write to the XCOMP\_BV field.

Execution of XSAVEOPT saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVEOPT instruction always uses the standard format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEOPT performs two optimizations that reduce the amount of data written to memory:

- **Init optimization.**

If  $XINUSE[i] = 0$ , state component  $i$  is not saved to the XSAVE area (even if  $RFBM[i] = 1$ ). (See below for exceptions made for MXCSR.)

- **Modified optimization.**

Each execution of XRSTOR and XRSTORS establishes XRSTOR\_INFO as a 4-tuple  $\langle w, x, y, z \rangle$  (see Section 13.8.3 and Section 13.12). Execution of XSAVEOPT uses the modified optimization only if the following all hold for the current value of XRSTOR\_INFO:

- $w = \text{CPL}$ ;
- $x = 1$  if and only if the logical processor is in VMX non-root operation;
- $y$  is the linear address of the XSAVE area being used by XSAVEOPT; and
- $z$  is 00000000\_00000000H. (This last item implies that XSAVEOPT does not use the modified optimization if the last execution of XRSTOR used the compacted form, or if an execution of XRSTORS followed the last execution of XRSTOR.)

If XSAVEOPT uses the modified optimization and  $XMODIFIED[i] = 0$  (see Section 13.6), state component  $i$  is not saved to the XSAVE area.

(In practice, the benefit of the modified optimization for state component  $i$  depends on how the processor is tracking state component  $i$ ; see Section 13.6. Limitations on the tracking ability may result in state component  $i$  being saved even though is in the same configuration that was loaded by the previous execution of XRSTOR.)

Depending on details of the operating system, an execution of XSAVEOPT by a user application might use the modified optimization when the most recent execution of XRSTOR was by a different application. Because of this, Intel recommends the application software not use the XSAVEOPT instruction.

The MXCSR register and MXCSR\_MASK are part of SSE state (see Section 13.5.2) and are thus associated with bit 1 of RFBM. However, the XSAVEOPT instruction also saves these values when  $RFBM[2] = 1$  (even if  $RFBM[1] = 0$ ). The init and modified optimizations do not apply to the MXCSR register and MXCSR\_MASK.

## 13.10 OPERATION OF XSAVEC

The operation of XSAVEC is similar to that of XSAVE. Two main differences are (1) XSAVEC uses the compacted format for the extended region of the XSAVE area; and (2) XSAVEC uses the init optimization (see Section 13.6). Unlike XSAVEOPT, XSAVEC does not use the modified optimization. See Section 13.2 for details of how to determine whether XSAVEC is supported.

The XSAVEC instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical (bitwise) AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be saved.

The following conditions cause execution of the XSAVEC instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.
- If  $CR0.TS[\text{bit } 3]$  is 1, a device-not-available exception (#NM) occurs.

- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVEC writes the XSTATE\_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE\_BV[*i*] ( $0 \leq i \leq 63$ ) as follows:<sup>2</sup>

- If RFBM[*i*] = 0, XSTATE\_BV[*i*] is written as 0.
- If RFBM[*i*] = 1, XSTATE\_BV[*i*] is set to the value of XINUSE[*i*] (see below for an exception made for XSTATE\_BV[1]). Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component *i* is in its initial configuration, XSTATE\_BV[*i*] may be written with either 0 or 1.
  - If state component *i* is not in its initial configuration, XSTATE\_BV[*i*] is written with 1.

XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC writes XSTATE\_BV[1] as 1 even if XINUSE[1] = 0.

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEC instruction sets bit 63 of the XCOMP\_BV field of the XSAVE header while writing RFBM[62:0] to XCOMP\_BV[62:0]. The XSAVEC instruction does not write any part of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.

Execution of XSAVEC saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the init optimization described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component *i*,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVEC instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEC performs the init optimization to reduce the amount of data written to memory. If XINUSE[*i*] = 0, state component *i* is not saved to the XSAVE area (even if RFBM[*i*] = 1). However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC saves all of state component 1 (SSE — including the XMM registers) even if XINUSE[1] = 0. Unlike the XSAVE instruction, RFBM[2] does not determine whether XSAVEC saves MXCSR and MXCSR\_MASK.

## 13.11 OPERATION OF XSAVES

The operation of XSAVES is similar to that of XSAVEC. The main differences are (1) XSAVES can be executed only if CPL = 0; (2) XSAVES can operate on the state components whose bits are set in XCR0 | IA32\_XSS and can thus operate on supervisor state components; and (3) XSAVES uses the modified optimization (see Section 13.6). See Section 13.2 for details of how to determine whether XSAVES is supported.

The XSAVES instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. EDX:EAX & (XCR0 | IA32\_XSS) (the logical AND the instruction mask with the logical OR of XCR0 and IA32\_XSS) is the **requested-feature bitmap (RFBM)** of the state components to be saved.

The following conditions cause execution of the XSAVES instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If CPL > 0 or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.

If none of these conditions cause a fault, execution of XSAVES writes the XSTATE\_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE\_BV[*i*] ( $0 \leq i \leq 63$ ) as follows:

- 
1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.
  2. Unlike the XSAVE and XSAVEOPT instructions, the XSAVEC instruction does **not** read the XSTATE\_BV field of the XSAVE header.



- If  $RFBM[i] = 0$ ,  $XSTATE\_BV[i]$  is written as 0.
- If  $RFBM[i] = 1$ ,  $XSTATE\_BV[i]$  is set to the value of  $XINUSE[i]$  (see below for an exception made for  $XSTATE\_BV[1]$ ). Section 13.6 defines  $XINUSE$  to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component  $i$  is in its initial configuration,  $XSTATE\_BV[i]$  may be written with either 0 or 1.
  - If state component  $i$  is not in its initial configuration,  $XSTATE\_BV[i]$  is written with 1.

$XINUSE[1]$  pertains only to the state of the XMM registers and not to MXCSR. However, if  $RFBM[1] = 1$  and MXCSR does not have the value 1F80H, XSAVES writes  $XSTATE\_BV[1]$  as 1 even if  $XINUSE[1] = 0$ .

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVES instruction sets bit 63 of the XCOMP\_BV field of the XSAVE header while writing  $RFBM[62:0]$  to  $XCOMP\_BV[62:0]$ . The XSAVES instruction does not write any part of the XSAVE header other than the  $XSTATE\_BV$  and  $XCOMP\_BV$  fields.

Execution of XSAVES saves into the XSAVE area those state components corresponding to bits that are set in  $RFBM$  (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVES instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XSAVES. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVES performs the init optimization to reduce the amount of data written to memory. If  $XINUSE[i] = 0$ , state component  $i$  is not saved to the XSAVE area (even if  $RFBM[i] = 1$ ). However, if  $RFBM[1] = 1$  and MXCSR does not have the value 1F80H, XSAVES saves all of state component 1 (SSE — including the XMM registers) even if  $XINUSE[1] = 0$ .

Like XSAVEOPT, XSAVES may perform the modified optimization. Each execution of XRSTOR and XRSTORS establishes  $XRSTOR\_INFO$  as a 4-tuple  $\langle w, x, y, z \rangle$  (see Section 13.8.3 and Section 13.12). Execution of XSAVES uses the modified optimization only if the following all hold:

- $w = CPL$ ;
- $x = 1$  if and only if the logical processor is in VMX non-root operation;
- $y$  is the linear address of the XSAVE area being used by XSAVEOPT; and
- $z[63]$  is 1 and  $z[62:0] = RFBM[62:0]$ . (This last item implies that XSAVES does not use the modified optimization if the last execution of XRSTOR used the standard form and followed the last execution of XRSTORS.)

If XSAVES uses the modified optimization and  $XMODIFIED[i] = 0$  (see Section 13.6), state component  $i$  is not saved to the XSAVE area.

## 13.12 OPERATION OF XRSTORS

The operation of XRSTORS is similar to that of XRSTOR. Three main differences are (1) XRSTORS can be executed only if  $CPL = 0$ ; (2) XRSTORS can operate on the state components whose bits are set in  $XCR0 \mid IA32\_XSS$  and can thus operate on supervisor state components; and (3) XRSTORS has only a compacted form (no standard form; see Section 13.8). See Section 13.2 for details of how to determine whether XRSTORS is supported.

The XRSTORS instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair  $EDX:EAX$  is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**.  $EDX:EAX \& (XCR0 \mid IA32\_XSS)$  (the logical AND of the instruction mask with the logical OR of  $XCR0$  and  $IA32\_XSS$ ) is the **requested-feature bitmap (RFBM)** of the state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.
- If  $CR0.TS[\text{bit } 3]$  is 1, a device-not-available exception (#NM) occurs.

- If CPL > 0 or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.

After checking for these faults, the XRSTORS instruction reads the first 64 bytes of the XSAVE header, including the XSTATE\_BV and XCOMP\_BV fields (see Section 13.4.2). A #GP occurs if any of the following conditions hold for the values read:

- XCOMP\_BV[63] = 0.
- XCOMP\_BV sets a bit in the range 62:0 that is not set in XCR0 | IA32\_XSS.
- XSTATE\_BV sets a bit (including bit 63) that is not set in XCOMP\_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component  $i$  for which RFBM[ $i$ ] = 1. XRSTORS updates state component  $i$  based on the value of bit  $i$  in the XSTATE\_BV field of the XSAVE header:

- If XSTATE\_BV[ $i$ ] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component. If XSTATE\_BV[1] = 0, XRSTORS initializes MXCSR to 1F80H.  
State component  $i$  is set to its initial configuration as indicated above if RFBM[ $i$ ] = 1 and XSTATE\_BV[ $i$ ] = 0 — **even if XCOMP\_BV[ $i$ ] = 0**. This is true for all values of  $i$ , including 0 (x87 state) and 1 (SSE state).
- If XSTATE\_BV[ $i$ ] = 1, the state component is loaded with data from the XSAVE area.<sup>1</sup> See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XRSTORS. See Section 13.13 for details regarding faults caused by memory accesses.

If XRSTORS is restoring a supervisor state component, the instruction causes a general-protection exception (#GP) if it would load any element of that component with an unsupported value (e.g., by setting a reserved bit in an MSR) or if a bit is set in any reserved portion of the state component in the XSAVE area.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; XRSTORS uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if RFBM[1] = XSTATE\_BV[1] = 1. XRSTORS causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

If an execution of XRSTORS causes an exception or a VM exit during or after restoring a supervisor state component, each element of that state component may have the value it held before the XRSTORS execution, the value loaded from the XSAVE area, or the element's initial value (as defined in Section 13.6). See Section 13.5.6 for some special treatment of PT state for the case in which XRSTORS causes an exception or a VM exit.

Like XRSTOR, execution of XRSTORS causes the processor to update its tracking for the init and modified optimizations (see Section 13.6 and Section 13.8.3). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  - If RFBM[ $i$ ] = 0, XINUSE[ $i$ ] is not changed.
  - If RFBM[ $i$ ] = 1 and XSTATE\_BV[ $i$ ] = 0, state component  $i$  may be tracked as init; XINUSE[ $i$ ] may be set to 0 or 1.
  - If RFBM[ $i$ ] = 1 and XSTATE\_BV[ $i$ ] = 1, state component  $i$  is tracked as not init; XINUSE[ $i$ ] is set to 1.
- The processor updates its tracking for the modified optimization and records information about the XRSTORS execution for future interaction with the XSAVEOPT and XSAVES instructions as follows:
  - If RFBM[ $i$ ] = 0, state component  $i$  is tracked as modified; XMODIFIED[ $i$ ] is set to 1.
  - If RFBM[ $i$ ] = 1, state component  $i$  may be tracked as unmodified; XMODIFIED[ $i$ ] may be set to 0 or 1.
  - XRSTOR\_INFO is set to the 4-tuple  $\langle w, x, y, z \rangle$ , where  $w$  is the CPL;  $x$  is 1 if the logical processor is in VMX non-root operation and 0 otherwise;  $y$  is the linear address of the XSAVE area; and  $z$  is XCOMP\_BV (this implies that  $z[63] = 1$ ).

---

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and XSTATE\_BV[ $i$ ] is 1, then XCOMP\_BV[ $i$ ] is also 1.

Note that, if RFBM is entirely zero (e.g., because the instruction mask in EDX:EAX is zero), no state components are modified, the XINUSE bitmap is not modified, and all bits are set in the XMODIFIED bitmap. Thus, if EDX:EAX was zero for the most recent execution of XRSTORS, an execution of XSAVEOPT or XSAVES will identify all state components as modified and will thus not use the modified optimization.

## 13.13 MEMORY ACCESSES BY THE XSAVE FEATURE SET

Each instruction in the XSAVE feature set operates on a set of XSAVE-managed state components. The specific set of components on which an instruction operates is determined by the values of XCR0, the IA32\_XSS MSR, EDX:EAX, and (for XRSTOR and XRSTORS) the XSAVE header.

Section 13.4 provides the details necessary to determine the location of each state component for any execution of an instruction in the XSAVE feature set. An execution of an instruction in the XSAVE feature set may access any byte of any state component on which that execution operates.

Section 13.5 provides details of the different XSAVE-managed state components. Some portions of some of these components are accessible only in 64-bit mode. Executions of XRSTOR and XRSTORS outside 64-bit mode will not update those portions; executions of XSAVE, XSAVEC, XSAVEOPT, and XSAVES will not modify the corresponding locations in memory.

Despite this fact, any execution of these instructions outside 64-bit mode may access any byte in any state component on which that execution operates — even those at addresses corresponding to registers that are accessible only in 64-bit mode. As result, such an execution may incur a fault due to an attempt to access such an address.

For example, an execution of XSAVE outside 64-bit mode may incur a page fault if paging does not map as read/write the section of the XSAVE area containing state component 7 (Hi16\_ZMM state) — despite the fact that state component 7 can be accessed only in 64-bit mode.



## 5. Updates to Chapter 18, Volume 1

Change bars and green text show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

---

Changes to this chapter: Updates to section 18.2.3, "Supervisor Shadow Stack Token", section 18.3.3.1, "Control Transfers between CPL 3 and CPL < 3", and section 18.3.6, "Legacy Compatibility Treatment".

### 18.1 INTRODUCTION

Return-oriented programming (ROP), and similarly CALL/JMP-oriented programming (COP/JOP), have been the prevalent attack methodologies for stealth exploit writers targeting vulnerabilities in programs. These attack methodologies have the common elements:

- A code module with execution privilege and contain small snippets of code sequence with the characteristic: at least one instruction in the sequence being a control transfer instruction that depends on data either in the return stack or in a register for the target address.
- Diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target (via modification in the data stack or in the register).

Control-Flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/COP/JOP style control-flow subversion attacks:

- Shadow stack: Return address protection to defend against ROP.
- Indirect branch tracking: Free branch protection to defend against COP/JOP.

Both capabilities introduce new instruction set extensions, and are described in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

Control-Flow Enforcement Technology introduces a new exception (#CP) with interrupt vector 21.

#### 18.1.1 Shadow Stack

A shadow stack is a second stack for the program that is used exclusively for control transfer operations. This stack is separate from the data stack and can be enabled for operation individually in user mode or supervisor mode. When shadow stacks are enabled, the CALL instruction pushes the return address on both the data and shadow stack. The RET instruction pops the return address from both stacks and compares them. If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP). Note that the shadow stack only holds the return addresses and not parameters passed to the call instruction.

The shadow stack is protected from tamper through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table protections are extended to support an additional attribute for pages to mark them as "Shadow Stack" pages. When shadow stacks are enabled, control transfer instructions/flows like near call, far call, call to interrupt/exception handlers, etc. store return addresses to the shadow stack and the access will fault if the underlying page is not marked as a "Shadow Stack" page. However stores from instructions like MOV, XSAVE, etc. will not be allowed. Likewise control transfer instructions like near RET, far RET, IRET, etc. when they attempt to read from the shadow stack the access will fault if the underlying page is not marked as a "Shadow Stack" page. This paging protection detects and prevents conditions that cause an overflow or underflow of the shadow stack when the shadow stack is delimited by non-shadow stack guard pages, or any malicious attempts to redirect the processor to consume data from addresses that are not shadow stack addresses.

#### 18.1.2 Indirect Branch Tracking

The ENDBRANCH instruction is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. This instruction opcode is selected to be one that is a NOP on legacy machines such that programs compiled with ENDBRANCH new instruction continue to function on old machines without the CET enforcement. On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the processor pipeline to detect control flow violations. The CPU implements a state machine that tracks indirect JMP and CALL instructions. When one of these instructions is executed, the state machine moves from IDLE to WAIT\_FOR\_ENDBRANCH state. In WAIT\_FOR\_ENDBRANCH state the next instruction in the program

stream must be an ENDBRANCH. If the next instruction is not an ENDBRANCH, the processor causes a control protection exception (#CP); otherwise, the state machine moves back to IDLE state.

### 18.1.3 Speculative Behavior when CET is Enabled

Speculative execution of near indirect JMP/CALL/RET indirect branches may be able to create an active side channel vulnerability that reveals the contents of data.

There are two basic methods that an attacker may be able to use to control indirect branch speculation in order to speculatively execute code that causes a side channel:

1. Attacker controlled prediction.
2. Attacker controlled jump redirection.

With attacker controlled prediction, the attacker trains indirect branch predictors such that the desired victim indirect branch goes to the attacker desired location. Examples include Branch Target Injection (also called "Variant 2" and "Spectre") and RSB wrap on underflow (also called "ret2spec").

With attacker controlled jump redirection, the attacker controls a speculative-only value used as input to the indirect branch so that the branch mispredicts to the attacker desired location. Examples of this include Bound Check Bypass Store (where a speculative store containing an attacker controlled value may overwrite the indirect branch target before the load of the target) and Speculative Store Bypass (where a load of the indirect branch target may bypass the most recent store of the target value and thus speculatively read an older attacker controlled value at the same memory location).

In addition to the existing mitigation features like IBRS, STIBP and IBPB, processors supporting CET will have a variety of additional features to constrain control flow speculation in order to mitigate such attacks. For details on these features, see Section 18.2.6, "Constraining Execution at Targets of RET" and Section 18.3.8, "Constraining Speculation after Missing ENDBRANCH".

## 18.2 SHADOW STACKS

A shadow stack is a second expand down stack used exclusively for control transfer operations. This stack is separate from the data stack. The shadow stack is not used to store data and hence is not explicitly writeable by software. Writes to the shadow stack are restricted to control transfer instructions and shadow stack management instructions. The shadow stack feature can be enabled separately in user mode (CPL == 3) or supervisor mode (CPL < 3).

Shadow stacks operate only in protected mode. Shadow stacks cannot be enabled in virtual 8086 mode.

It is recommended to not configure the shadow stack in the linear address range 0 to 64 KB or adjacent to the canonical address boundary.

### 18.2.1 Shadow Stack Pointer and its Operand and Address Size Attributes

When CET is enabled the processor supports a new architectural register, shadow stack pointer (SSP), when the processor supports the shadow stack feature. The SSP cannot be directly encoded as a source, destination or memory operand in instructions. The SSP points to the current top of the shadow stack.

The width of the shadow stack is 32-bit in 32-bit/compatibility mode and is 64-bit in 64-bit mode. The address-size attribute of the shadow stack is likewise 32-bit in 32-bit/compatibility mode and 64-bit in 64-bit mode.

### 18.2.2 Terminology

When shadow stacks are enabled, certain control transfer instructions/flows and shadow stack management instructions do loads/stores to the shadow stack. Such load/stores from control transfer instructions and shadow stack management instructions are termed as shadow\_stack\_load and shadow\_stack\_store to distinguish them from a load/store performed by other instructions like MOV, XSAVES, etc.

The pseudocode for the instruction operations use the notation `ShadowStackEnabled(CPL)` as a test of whether shadow stacks are enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

```
ShadowStackEnabled(CPL):
  IF CR4.CET = 1 AND CRO.PE = 1 AND EFLAGS.VM = 0
    IF CPL = 3
      THEN
        (* Obtain the shadow stack enable from IA32_U_CET MSR (MSR address 6A0H) used to enable
           feature for CPL = 3 *)
        SHADOW_STACK_ENABLED = IA32_U_CET.SH_STK_EN;
      ELSE
        (* Obtain the shadow stack enable from IA32_S_CET MSR (MSR address 6A2H) used to enable
           feature for CPL < 3 *)
        SHADOW_STACK_ENABLED = IA32_S_CET.SH_STK_EN;
    FI;
    IF SHADOW_STACK_ENABLED = 1
      THEN
        return TRUE;
      ELSE
        return FALSE;
    FI;
  ELSE
    (* Shadow stacks not enabled in real mode and virtual-8086 mode or if the master CET feature
       enable in CR4 is disabled *)
    return FALSE;
  ENDIF
```

Additionally, the following terms are used.

- `ShadowStackPush4B`: Decrements the shadow stack pointer (SSP) by 4 bytes and copies the 4 byte source operand to the top of the shadow stack.
- `ShadowStackPush8B`: Decrements the shadow stack pointer (SSP) by 8 bytes and copies the 8 byte source operand to the top of the shadow stack.
- `ShadowStackPop4B`: Copies 4 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 4 bytes to point to the new top of stack.
- `ShadowStackPop8B`: Copies 8 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 8 bytes to point to the new top of stack.
- `shadow_stack_lock_cmpxchg8B(address, new_value, expected_value)`: this function executes atomically and compares the `expected_value` to the 8 byte read from memory specified by the `address` operand using a locked `shadow_stack_load`. If the two values are equal, the `new_value` is written to memory specified by the `address` operand using a locked `shadow_stack_store`. If the two values are not equal, then the value read by the locked `shadow_stack_load` is written back. The memory specified by the `address` operand receives a write cycle without regard to the result of the comparison. The function returns the value read from the memory specified by the `address` operand.

### 18.2.3 Supervisor Shadow Stack Token

On an inter-privilege far CALL or when calling an interrupt/exception handler at a higher privilege level, a stack switch occurs; if shadow stacks are enabled at the new privilege level, then a shadow stack switch occurs. Shadow stacks that can be switched to by hardware as part of a privilege change are required to have a supervisor shadow stack token set up by the supervisor to provide the address of the new SSP register. The supervisor shadow stack tokens also serve the purpose of enforcing that a shadow stack can be made active on only one logical processor when switched to by the processor. The supervisor shadow stack token must be set up only on shadow stacks

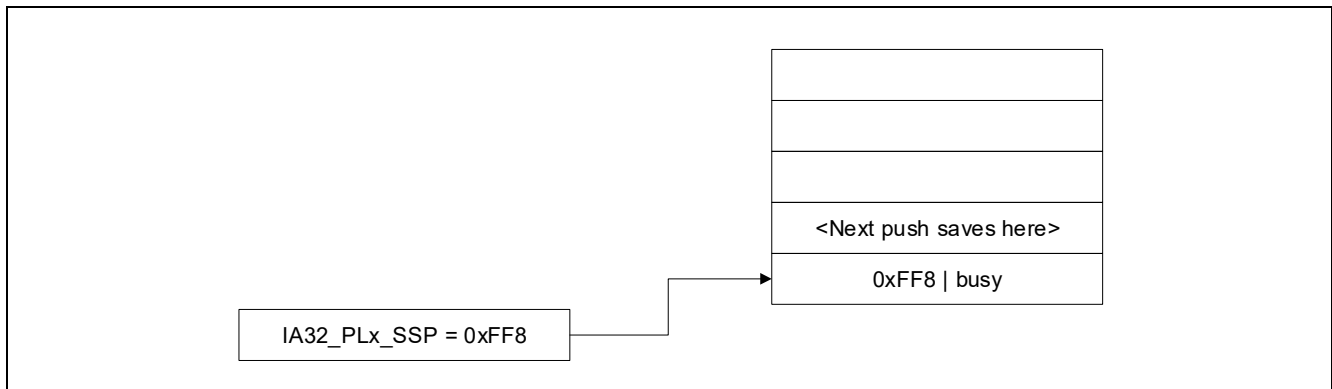


intended to be used on these transfers. The address of the supervisor shadow stack token is programmed into the IA32\_PLx\_SSP MSR (where  $0 \leq x \leq 2$ ). The WRMSR and XRSTORS instructions require the address specified in the IA32\_PLx\_SSP MSR (where  $0 \leq x \leq 2$ ) to be 4 byte aligned; otherwise, the instruction causes a general protection exception (#GP(0)).

The supervisor shadow stack token is a 64-bit value formulated as follows.

- Bit 63:3: Bits 63:3 of the linear address of the supervisor shadow stack token.
- Bit 2: Reserved. Must be zero.
- Bit 1: Reserved. Must be zero.
- Bit 0: Busy bit. If 0, indicates this shadow stack is not active on any logical processor. If 1, indicates this shadow stack is currently active on one of the logical processors.

The following figure illustrates a supervisor shadow stack with a supervisor shadow stack token located at its base.



**Figure 18-1. Supervisor Shadow Stack with a Supervisor Shadow Stack Token**

The 8-byte supervisor shadow stack token and the 24-byte stack frame pushed on the shadow stack after the token is acquired must be fully contained within a 32-byte region that is aligned to 32-bytes on the shadow stack. If they are not, a general-protection exception (#GP(0)) occurs. While the accesses to the token may cause faults, VM exits, or data breakpoints, the subsequent pushes of the remaining 24 bytes should not.

The processor does the following checks prior to switching to a supervisor shadow stack programmed into the IA32\_PLx\_SSP MSR. These steps are performed atomically.

1. Load the supervisor shadow stack token from the address specified in the IA32\_PLx\_SSP MSR using a `shadow_stack_load`.
2. Check if the busy bit in the token is 0; reserved bits must be 0.
3. Check if the address programmed in the MSR matches the address in the supervisor shadow stack token; reserved bits must be 0.
4. If checks 2 and 3 are successful, then set the busy bit in the token using a `shadow_stack_store` and switch the SSP to the value specified in the IA32\_PLx\_SSP MSR.
5. If checks 2 or 3 fail, then the busy bit is not set and a #GP(0) exception is raised.

On a far RET to a lesser privilege level or on an IRET that switches shadow stack, the instruction clears the busy bit in the shadow stack token as follows. These steps are also performed atomically.

1. Load the supervisor shadow stack token from the SSP using a `shadow_stack_load`.
2. Check if the busy bit in the token is 1; reserved bits must be 0.
3. Check if the address programmed in supervisor shadow stack token matches SSP; reserved bits must be 0.
4. If checks 2 and 3 are successful, then clear the busy bit in the token using a `shadow_stack_store`; else continue without modifying the contents of the shadow stack pointed to by SSP.

## 18.2.4 Shadow Stack Usage on Task Switch

A task switch (see Chapter 7, “Task Management” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*) may be invoked by:

- JMP or CALL instruction to a TSS descriptor in the GDT.
- JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.

With shadow stack enabled, the new task must be associated with a 32-bit TSS and must not be in virtual-8086 mode. The 32-bit SSP for the new task is located at offset 104 in the 32-bit TSS. Thus the TSS of the new task must be at least 108 bytes. This SSP is required to be 8 byte aligned, and required to point to a “supervisor shadow stack” token (though the task may be at CPL3).

On a task switch initiated by a CALL instruction, an interrupt, or exception, the SSP of the old task is pushed onto the shadow stack of the new task along with the CS and LIP of the old task. This is true even for a nested task switch initiated by a CALL instruction. Likewise, on a task switch initiated by IRET, the SSP of the new task is restored from the shadow stack of old task. The CS and LIP on the shadow stack of the old task are matched against the return address determined by the CS and EIP of the new task. If the match fails, a #CP(FAR-RET/IRET) exception is reported.

## 18.2.5 Switching Shadow Stacks

The architecture provides a mechanism to switch shadow stacks using a pair of instructions; RSTORSSP and SAVEPREVSSP. The RSTORSSP instruction verifies a shadow-stack-restore token located at the top of the new shadow stack and referenced by the memory operand of this instruction. After RSTORSSP determines the validity of the restore point on the new shadow stack, it switches the SSP to point to the token. The shadow-stack-restore token is a 64-bit value formatted as follows.

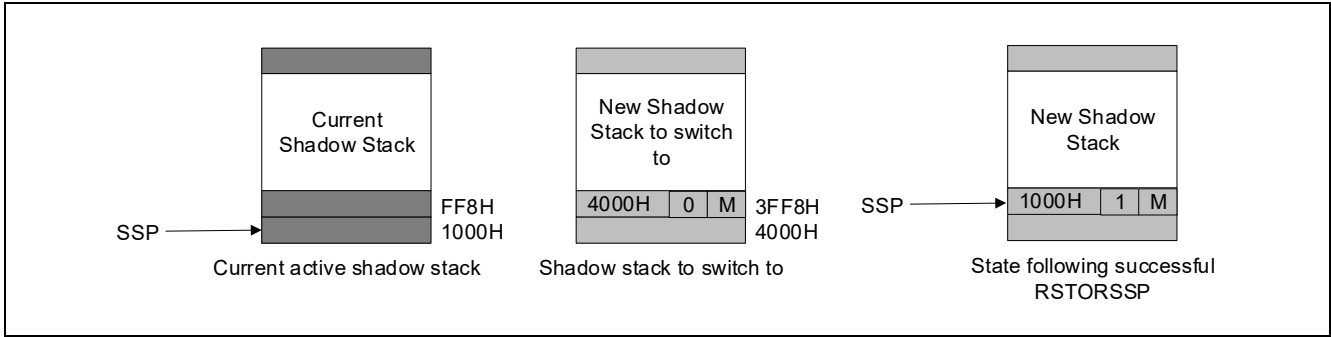
- Bit 63:2: Value of shadow stack pointer when this restore point was created.
- Bit 1: Reserved. Must be zero.
- Bit 0: Mode bit. If 0, the token is a compatibility/legacy mode shadow-stack-restore token. If 1, then this shadow stack restore token can be used with a RSTORSSP instruction in 64-bit mode.

The shadow-stack-restore token is created by the SAVEPREVSSP instruction. The operating system may also create a restore point on a shadow stack by creating a shadow-stack-restore token.

Once the shadow stack has been switched to a new shadow stack by the RSTORSSP instruction, software can create a restore point on the old shadow stack by executing the SAVEPREVSSP instruction. In order to allow the SAVEPREVSSP instruction to determine the address where to save the shadow-stack-restore token, the RSTORSSP instruction replaces the shadow-stack-restore token with a previous-ssp token that holds the value of the SSP at the time the RSTORSSP instruction was invoked. The previous-ssp token is formatted as follows.

- Bit 63:2: Shadow stack pointer when the RSTORSSP instruction was invoked, i.e., the SSP of the old shadow stack.
- Bit 1: Set to 1.
- Bit 0: Mode bit. If 0, then this previous-ssp token can be used with a SAVEPREVSSP instruction in compatibility/legacy mode. If 1, then this previous-ssp token can be used with a SAVEPREVSSP instruction in 64-bit mode.

The following figure illustrates the RSTORSSP instruction operation during a shadow stack switching sequence.

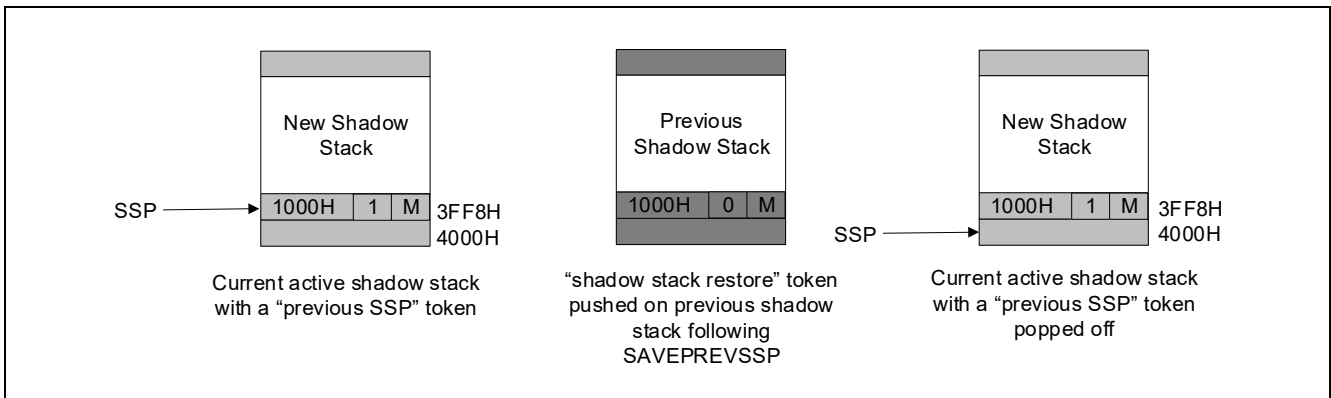


**Figure 18-2. RSTORSSP to Switch to New Shadow Stack**

In this example, the initial SSP is 1000H and the shadow-stack-restore token is on a new shadow stack at address 3FF8H. The token at address 3FF8H holds the SSP when this restore point was created; in this example it is 4000H.

In order to switch to the new shadow stack, the RSTORSSP instruction is invoked with the memory operand pointing set to 3FF8H. When the RSTORSSP instruction completes, the SSP is set to 3FF8H and the shadow-stack-restore token at 3FF8H is replaced by a previous-ssp token that holds the address 1000H, i.e., the old SSP.

The following figure illustrates the SAVEPREVSSP instruction operation during a shadow stack switching sequence.



**Figure 18-3. SAVEPREVSSP to Save a Restore Point**

To allow switching back to this old shadow stack, a SAVEPREVSSP instruction is now invoked. The SAVEPREVSSP instruction does not take any memory operand and expects to find a previous-ssp token at the top of the shadow stack, i.e., at address 3FF8H. The SAVEPREVSSP instruction then saves a shadow-stack-restore token on the old shadow stack at address FF8H, and the token itself holds the address 1000H which is the address recorded in the previous-ssp token. The SAVEPREVSSP instruction also pops the previous-ssp token off the current shadow stack and thus the SSP following SAVEPREVSSP is 4000H.

Subsequently to switch back to the old shadow stack, a RSTORSSP instruction may be invoked with memory operand set to FF8H.

If, following a switch to a new shadow stack, it is not required to create a restore point on the old shadow stack, then the previous-ssp token created by the RSTORSSP instruction can be popped off the shadow stack by using the INCSSP instruction.

See the SAVEPREVSSP and RSTORSSP instruction operations for the detailed algorithm.

## 18.2.6 Constraining Execution at Targets of RET

Instructions at the target of a RET instruction will not execute, even speculatively, if the RET addresses (either from normal stack or shadow stack) are speculative-only or do not match, unless the target of the RET is also predicted (e.g., by a Return Stack Buffer prediction), when CET shadow stack is enabled. A RET address would be speculative-only if it was modified by an older speculative-only store, or was an older value than the most recent value stored to that address on the logical processor.

## 18.3 INDIRECT BRANCH TRACKING

When the indirect branch tracking feature is active, the indirect JMP/CALL instruction behavior changes as follows.

- **JMP:** If the next instruction retired after an indirect JMP is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, then a #CP fault is generated. Below JMP instructions are tracked to enforce an ENDBRANCH. Note that Jcc, RIP relative, and far direct JMP are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
  - JMP r/m16, JMP r/m32, JMP r/m64
  - JMP m16:16, JMP m16:32, JMP m16:64
- **CALL:** If the next instruction retired after an indirect CALL is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 in 64-bit mode, then a #CP fault is generated. Below CALL instructions are tracked to enforce an ENDBRANCH. Note that relative and zero displacement forms of CALL instructions are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
  - CALL r/m16, CALL r/m32, CALL r/m64
  - CALL m16:16, CALL m16:32, CALL m16:64

The ENDBR32 and ENDBR64 instructions will have the same effect as the NOP instruction on Intel 64 processors that do not support CET. On processors supporting CET, these instructions do not change register or flag state. This allows CET instrumented programs to execute on processors that do not support CET. Even when CET is supported and enabled, these NOP-like instructions do not affect the execution state of the program, do not cause any additional register pressure, and are minimally intrusive from power and performance perspectives.

The processor implements two dual-state machines to track indirect CALL/JMP for terminations. One state machine is maintained for user mode and one for supervisor mode. At reset the user and supervisor mode state machines are in IDLE state.

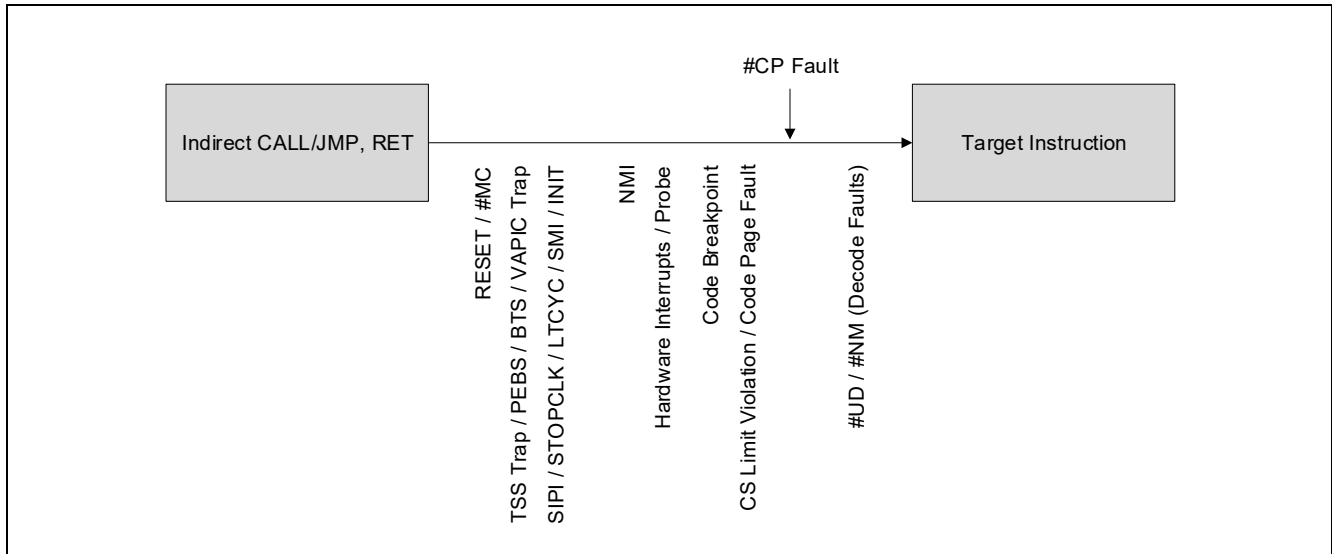
On instructions other than indirect CALL/JMP, the state machine stays in the IDLE state.

On an indirect CALL or JMP instruction, the state machine transitions to the WAIT\_FOR\_ENDBRANCH state.

In the WAIT\_FOR\_ENDBRANCH state, the indirect branch tracking state machine verifies the next instruction is an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, and either:

- Causes a #CP fault, or
- Allows the next instruction if legacy compatibility configuration allows (see Section 18.3.6).

The priority of the #CP(ENDBRANCH) exception relative to other events is as follows.



**Figure 18-4. Priority of Control Protection Exception on Missing ENDBRANCH**

Higher priority faults/traps/events that occur at the end of an indirect CALL/JMP are delivered ahead of any #CP(ENDBRANCH) fault. The CET state machine at the privilege level where the higher priority fault/trap/event occurred retains its state when the control transfers to the fault/trap/event handler. The instruction pointer pushed on the stack for a #CP(ENDBRANCH) fault is the address of the instruction at the target of the indirect CALL/JMP that caused the fault.

### 18.3.1 No-track Prefix for Near Indirect CALL/JMP

CET allows software to designate certain indirect CALL and JMP instructions as “non-tracked indirect control transfer instructions”. Software (e.g., compiler generated code for switch statements, jump tables, etc.) should use the no-track prefix only if they have generated code to validate the possible targets of this CALL/JMP to be legal targets. Software (e.g., compilers), when using the no-track prefix with CALL/JMP where an absolute offset is specified indirectly in a memory location, should ensure that such memory locations cannot be tampered. When enabled by setting the NO\_TRACK\_EN control in the IA32\_U\_CET/IA32\_S\_CET MSR, near indirect CALL and JMP instructions when prefixed with 3EH do not modify the CET indirect branch tracker. Far CALL and JMP instructions are always tracked and ignore the 3EH prefix. When this control is 0, near indirect CALL and JMP instructions are always tracked irrespective of the presence of the 3EH prefix.

In 64-bit mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix if there isn’t a 64H/65H prefix on the instruction.

In legacy/compatibility mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix when it is the last group 2 prefix on the instruction.

### 18.3.2 Terminology

The pseudocode for the instruction operations use a notation `EndbranchEnabled(CPL)` as a test of whether indirect branch tracking is enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

`EndbranchEnabled(CPL):`

```

IF CR4.CET = 1 AND CRO.PE = 1 AND EFLAGS.VM = 0
  IF CPL = 3
    THEN
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL = 3 *)
      ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
    ELSE
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL < 3 *)
      ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
  FI;
  IF ENDBR_ENABLED = 1
    THEN
      return TRUE;
    ELSE
      return FALSE;
  FI;
ELSE
  (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature
  enable in CR4 is disabled *)
  return FALSE;
ENDIF

```

Likewise the notation `EndbranchEnabledAndNotSuppressed` is defined as follows:

`EndbranchEnabledAndNotSuppressed(CPL):`

```

IF CR4.CET = 1 AND CRO.PE = 1 AND EFLAGS.VM = 0
  IF CPL = 3
    THEN
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL = 3 *)
      ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
      SUPPRESSED = IA32_U_CET.SUPPRESS;
    ELSE
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL < 3 *)
      ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
      SUPPRESSED = IA32_S_CET.SUPPRESS;
  FI;
  IF ENDBR_ENABLED = 1 AND SUPPRESSED = 0
    THEN
      return TRUE;
    ELSE
      return FALSE;
  FI;
ELSE
  (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature
  enable in CR4 is disabled *)
  return FALSE;
ENDIF

```

### 18.3.3 Indirect Branch Tracking

The hardware implements two CET indirect branch tracker state machines, one for user mode (CPL == 3) and one for supervisor mode (CPL < 3). At any time, which of the CET indirect branch trackers is in the active state depends on the CPL of the machine. When a user space program is executing, the CPL 3 CET indirect branch tracker is active. When supervisor mode software is executing, the CPL < 3 tracker is active. This section describes the various control transfer conditions and the tracker state on those transfers.

#### 18.3.3.1 Control Transfers between CPL 3 and CPL < 3

Some events and instructions can cause control transfer to occur from CPL 3 to CPL < 3, and vice versa. As part of the CPL change the hardware also switches the active CET indirect branch tracker. For example, when an interrupt occurs during execution of a user mode (CPL == 3) program and it causes the CPL to switch to supervisor mode (CPL < 3) then, as part of the CPL change, the user mode CET indirect branch tracker becomes inactive and the supervisor mode CET indirect branch tracker becomes active. A subsequent IRET is used by the interrupt handler to return to the interrupted user mode program. This IRET causes the processor to switch the CPL to user mode (CPL == 3) and, as part of the CPL change, the supervisor mode CET indirect branch tracker becomes inactive and the user mode CET indirect branch tracker becomes active.

The CPL where the event or instruction that caused the control transfer occurs is termed the source CPL, and the CET indirect branch tracker state at that CPL is referred here as the source CET indirect branch tracker state. The CPL reached at the end of the control transfer is termed the destination CPL, and the CET indirect branch tracker state at that CPL is referred to as the destination CET indirect branch tracker state.

This section describes various cases of control transfers that occur between user mode (CPL 3) and supervisor mode (CPL < 3).

In all these cases the source CET indirect branch tracker state becomes not active and retains its state (IDLE, WAIT\_FOR\_ENDBRANCH), and the target CET indirect branch tracker state becomes active if there was no fault during the transfer.

- Case 1: Far CALL/JMP, SYSCALL/SYSENTER
  - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT\_FOR\_ENDBRANCH. This enforces that the subroutine invoked by a far CALL/JMP must begin with an ENDBRANCH.
- Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks
  - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT\_FOR\_ENDBRANCH.
- Case 3: IRET/Far RET
  - If indirect branch tracking enabled, the target indirect branch tracker becomes active and keeps its state. If the user mode was interrupted by a higher priority event, like an interrupt at the end of the indirect CALL/JMP, then when an IRET or Far RET is used to return to the interrupted user mode program, the user mode indirect branch tracker retains its state and a #CP fault will occur if the next instruction decoded is not an ENDBR32/64 according to mode of machine.

#### 18.3.3.2 Control Transfers within CPL 3 or CPL < 3

Some events and instructions can cause control transfer to occur within CPL 3 or CPL < 3. For such transfers since the CPL class does not change, the same indirect branch tracker is used at the beginning and end of the control transfer.

- Case 1: Far CALL/JMP, Near indirect CALL/JMPCALL/JMP
  - Far CALL/JMP: If indirect branch tracking is enabled, active indirect branch tracker is unsuppressed and goes to WAIT\_FOR\_ENDBRANCH.
  - Near indirect CALL/JMPCALL/JMP: If indirect branch tracking is enabled and not suppressed, active indirect branch tracker goes to WAIT\_FOR\_ENDBRANCH.
- Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks

- If indirect branch tracking is enabled, the active indirect branch tracker is unsuppressed and goes to WAIT\_FOR\_ENDBRANCH.
- Case 3: IRET
  - If indirect branch tracking is enabled, the active indirect branch tracker keeps its state.

### 18.3.4 Indirect Branch Tracking State Machine

The state machine is described by Table 18-1.

**Table 18-1. Indirect Branch Tracking State Machine**

Current State	Trigger	Next State
TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1	Instructions other than indirect CALL/JMP or 3EH prefixed near indirect CALL/JMP and NO_TRACK_EN=1	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Indirect CALL/JMP without 3EH prefix Indirect CALL/JMP with 3EH prefix and NO_TRACK_EN=0 Far CALL/JMP	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1	INT3/INT1	TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	ENDBRANCH instruction	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Successful ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Instructions other than ENDBRANCH, successful ENCLU[ERESUME] or INT3 or INT1	If legacy compatibility treatment is not enabled or if not allowed by legacy code page bitmap: <ul style="list-style-type: none"> <li>▪ No state change and deliver #CP (ENDBRANCH)</li> </ul> If legacy compatibility treatment is enabled and transfer allowed by legacy code page bitmap: <ul style="list-style-type: none"> <li>▪ TRACKER=IDLE, SUPPRESS=ISUPPRESS_DIS, ENDBR_EN=1</li> </ul>
TRACKER=x, SUPPRESS=x, ENDBR_EN=0	All instructions	TRACKER=x, SUPPRESS=x, ENDBR_EN=0
TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1	Far CALL/JMP, INTn/INT3/INTO	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	ENDBRANCH instruction Successful ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	All other instructions including indirect CALL/JMP	TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1
TRACKER=1, SUPPRESS=1, ENDBR_EN=1 (This state cannot be reached by hardware and is disallowed as a valid state by WRMSR/XRSTORS/VM entry/VM exit)	NA	NA



### 18.3.5 INT3 Treatment

INT3 are treated special in the WAIT\_FOR\_ENDBRANCH state. Occurrence of INT3 do not move the tracker to IDLE but instead the #BP trap from the INT3 instructions respectively is delivered as a higher priority event than the #CP exception due to missing ENDBRANCH.

Inside an enclave, INT3 delivers a fault-class exception and thus does not require the CPL to be less than DPL in the IDT gate 3. Following opt-out entry, the instruction delivers #UD. Following opt-in entry, INT3 delivers #BP. The special treatment of INT3 in WAIT\_FOR\_ENDBRANCH state does not apply in enclave mode following opt-out entry.

### 18.3.6 Legacy Compatibility Treatment

ENDBRANCH legacy compatibility treatment allows a CET enabled program to be used with legacy software that was not compiled / instrumented with ENDBRANCH. A CET enabled program enters legacy compatibility treatment when all of the below conditions are met.

1. Legacy compatibility configuration is enabled in this CPL class by setting the LEG\_IW\_EN bit in IA32\_U\_CET/IA32\_S\_CET.
2. Control transfer is performed using an indirect CALL/JMP without no-track prefix to an instruction other than ENDBRANCH.
3. The legacy code page bitmap is setup to indicate that the target of the control transfer is a legacy code page.

The legacy code page bitmap is a data structure in program memory that is used by the hardware to determine if the code page to which a legacy transfer is being performed is allowed. **The access rights for accessing the legacy code page bitmap is determined by the current privilege level (CPL). The legacy code page bitmap is expected to be setup as a read-only data structure.**

When a matching ENDBRANCH instruction is not decoded at the target of an indirect CALL/JMP when required, the processor performs the below actions.

CET indirect branch tracking state machine violation event handler:

If LEG\_IW\_EN == 1

LA = LIP;

IF ENCLAVE\_MODE == 1

LA = LA - SECS.BASEADDR;

ENDIF

(\* Load byte from bitmap. Address-size attribute for this load is 64 bits if IA32\_EFER.LMA is 1 and is 32 bits when IA32\_EFER.LMA is 0 \*)

IF (IA32\_EFER.LMA & CS.L) == 0

BITMAP\_BYTE = load 1 byte from address (BITMAP\_BASE + LA[31:15])

ELSE IF (CR4.LA57 == 0)

BITMAP\_BYTE = load 1 byte from address (BITMAP\_BASE + LA[47:15])

ELSE

BITMAP\_BYTE = load 1 byte from address (BITMAP\_BASE + LA[56:15])

FI;

IF BITMAP\_BYTE & (1 << LA[14:12]) == 0 then Deliver #CP(ENDBRANCH) fault

IF CPL = 3

IA32\_U\_CET.TRACKER = IDLE

IA32\_U\_CET.SUPPRESS = IA32\_U\_CET.SUPPRESS\_DIS == 0 ? 1 : 0

ELSE

IA32\_S\_CET.TRACKER = IDLE

IA32\_S\_CET.SUPPRESS = IA32\_S\_CET.SUPPRESS\_DIS == 0 ? 1 : 0

ENDIF

Restart the instruction (handle all arch. consistency around MOV SS state machines, STI etc.) without opening up interrupt/trap window.

ELSE

Deliver #CP(ENDBRANCH) Fault

ENDIF

Faults/traps in pseudocode are delivered normally (e.g., #PF, EPT violation). On a fault, the active tracker holds the last value (WAIT\_FOR\_ENDBRANCH) and the address saved on the stack is the current IP (instruction that wasn't the ENDBRANCH).

The CET indirect branch tracking state machine is suppressed in legacy compatibility mode if the SUPPRESS\_DIS control bit is 0.

Once the CET indirect branch tracking state machine has been suppressed, subsequent indirect CALL/JMP are not tracked for termination instruction.

Once CET indirect branch tracking has been suppressed, subsequent execution of ENDBRANCH instructions will do the following (see the ENDBR32 and ENDBR64 instructions in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for details).

```
IF EndbranchEnabled(CPL) == 0
    NOP
ELSE
    SUPPRESS = 0
    TRACKER = IDLE
ENDIF
```

### 18.3.6.1 Legacy Code Page Bitmap Format

The legacy code page bitmap is a flat bitmap whose linear address is pointed to by the EB\_LEG\_BITMAP\_BASE. Each bit in the bitmap represents a 4K page in linear memory. If the bit is 1 it indicates that the corresponding code page is a legacy code page; else it is a CET-enabled code page.

The processor uses the linear address of the instruction to which legacy transfer was attempted to lookup the bitmap. Bits of the linear address used as index in the bitmap are as follows.

- In legacy and compatibility mode: Bits 31:12.
- In 64-bit mode (EFER.LMA=1 and CS.L=1): Bits 47:12.

## 18.3.7 Other Considerations

### 18.3.7.1 Intel® Transactional Synchronization Extensions (Intel® TSX) Interactions

The XBEGIN instruction encodes the relative offset to the abort handler and hence the fallback to the abort handler can be considered as a "direct" branch and the abort handler does not need to have an ENDBRANCH.

CET continues to enforce indirect CALL/JMP tracking within a transaction. Legacy compatibility treatment inside a transaction functions normally. If a transaction abort occurs then the processor sets the state of the indirect branch tracker to IDLE and not-suppressed.

### 18.3.7.2 #CP(ENDBRANCH) Priority w.r.t #NM and #UD

#NM, #UD and #CP(ENDBRANCH) are in the same priority class. Both #NM and #UD are opcode based faults. The #CP(ENDBRANCH) is prioritized higher than #NM and #UD as CET architecturally requires an ENDBRANCH at target of indirect CALL/JMP.

### 18.3.7.3 #CP(ENDBRANCH) Priority w.r.t #BP and #DB

Debug Exceptions priority is as follows.

- Traps delivered before any #CP(ENDBRANCH) fault: Data breakpoint trap, IO breakpoint trap single step trap, task switch trap.
- Code Breakpoint fault detected before instruction decode and delivered before #CP(ENDBRANCH).

- General-detect (GD) exception condition fault: Lower priority than #CP(ENDBRANCH).
- On IRET back from #DB/#BP, the source indirect branch tracker becomes active if enabled and not suppressed.

INT3 does not cause #CP(ENDBRANCH) to support debugger usage of replacing bytes of ENDBRANCH with INT3 to set breakpoints. INT3 at target of a CALL-JMP(indirect) cause #BP(INT3) instead of #CP(ENDBRANCH), #CP(ENDBRANCH) fault is delayed. #BP caused by INT3 treated like other events that are higher priority than CET fault. On IRET back from #BP the source indirect tracker becomes active if enabled and not suppressed.

### 18.3.8 Constraining Speculation after Missing ENDBRANCH

When the CET tracker is in the WAIT\_FOR\_ENDBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH.

This means that when indirect branch tracking is enabled and not suppressed, the instructions at the target of a near indirect JMP/CALL without the no-track prefix will only speculatively execute if there is an ENDBRANCH at the target. This can constrain both attacker controlled prediction as well as attacker controlled jump redirection attacks on near indirect JMPs/CALLs by reducing the gadgets available to an attacker using these techniques. Early implementations of CET may limit the speculative execution to a small number of instructions (less than 8, with no more than 5 loads) past a missing ENDBRANCH, while later implementations will completely block the speculative execution of instructions after a missing ENDBRANCH.

This mechanism also limits or blocks speculation of the next sequential instructions after an indirect JMP or CALL, presuming the JMP/CALL puts the CET tracker into the WAIT\_FOR\_ENDBRANCH state and the next sequential instruction is not an ENDBRANCH.

## 18.4 INTEL® TRUSTED EXECUTION TECHNOLOGY (INTEL® TXT) INTERACTIONS

GETSEC[ENTERACCS] and GETSEC[SENDER] clear CR4.CET, and it is not restored when these instructions complete.

GETSEC[EXITAC] will cause #GP(0) fault if CR4.CET is set.

## 6. Updates to Appendix D (previously Appendix E), Volume 1

Change bars and green text show changes to Appendix D (previously Appendix E) of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

---

Changes to this chapter: Removed reference to previous Appendix D, which has been removed from the manual. Relabeled this appendix as Appendix D. The appendix removed is Appendix D, "Guidelines for Writing x87 FPU Exception Handlers". Removed link to outdated reference document.

# APPENDIX D

## GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

---

See Section 11.5, “SSE, SSE2, and SSE3 Exceptions,” for a detailed discussion of SIMD floating-point exceptions.

This appendix considers only SSE/SSE2/SSE3 instructions that can generate numeric (SIMD floating-point) exceptions, and gives an overview of the necessary support for handling such exceptions. This appendix does not address instructions that do not generate floating-point exceptions (such as RSQRTSS, RSQRTPS, RCPSS, or RCPPS), any x87 instructions, or any unlisted instruction.

For detailed information on which instructions generate numeric exceptions, and a listing of those exceptions, refer to Appendix C, “Floating-Point Exceptions Summary.” Non-numeric exceptions are handled in a way similar to that for the standard IA-32 instructions.

### D.1 TWO OPTIONS FOR HANDLING FLOATING-POINT EXCEPTIONS

Just as for x87 FPU floating-point exceptions, the processor takes one of two possible courses of action when an SSE/SSE2/SSE3 instruction raises a floating-point exception:

- If the exception being raised is masked (by setting the corresponding mask bit in the MXCSR to 1), then a default result is produced which is acceptable in most situations. No external indication of the exception is given, but the corresponding exception flags in the MXCSR are set and may be examined later. Note though that for packed operations, an exception flag that is set in the MXCSR will not tell which of the sub-operands caused the event to occur.
- If the exception being raised is not masked (by setting the corresponding mask bit in the MXCSR to 0), a software exception handler previously registered by the user with operating system support will be invoked through the SIMD floating-point exception (#XM, exception 19). This case is discussed below in Section D.2, “Software Exception Handling.”

### D.2 SOFTWARE EXCEPTION HANDLING

The #XM handler is usually part of the system software (the operating system kernel). Note that an interrupt descriptor table (IDT) entry must have been previously set up for exception 19 (refer to Chapter 6, “Interrupt and Exception Handling,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Some compilers use specific run-time libraries to assist in floating-point exception handling. If any x87 FPU floating-point operations are going to be performed that might raise floating-point exceptions, then the exception handling routine must either disable all floating-point exceptions (for example, loading a local control word with FLDCW), or it must be implemented as re-entrant. If this is not the case, the routine has to clear the status flags for x87 FPU exceptions or to mask all x87 FPU floating-point exceptions. For SIMD floating-point exceptions though, the exception flags in MXCSR do not have to be cleared, even if they remain unmasked (but they may still be cleared). Exceptions are in this case precise and occur immediately, and a SIMD floating-point exception status flag that is set when the corresponding exception is unmasked will not generate an exception.

Typical actions performed by this low-level exception handling routine are:

- Incrementing an exception counter for later display or printing
- Printing or displaying diagnostic information (e.g. the MXCSR and XMM registers)
- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it
- Storing information about the exception in a data structure that will be passed to a higher level user exception handler

In most cases (and this applies also to SSE/SSE2/SSE3 instructions), there will be three main components of a low-level floating-point exception handler: a prologue, a body, and an epilogue.

The prologue performs functions that must be protected from possible interruption by higher-priority sources - typically saving registers and transferring diagnostic information from the processor to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler (assuming that the interrupt handler was called through an interrupt gate, meaning that the processor cleared the interrupt enable (IF) flag in the EFLAGS register - refer to Section 6.5.1, "Call and Return Operation for Interrupt or Exception Handling Procedures").

The body of the exception handler examines the diagnostic information and makes a response that is application-dependent. It may range from halting execution, to displaying a message, to attempting to fix the problem and then proceeding with normal execution, to setting up a data structure, calling a higher-level user exception handler and continuing execution upon return from it. This latter case will be assumed in Section D.4, "SIMD Floating-Point Exceptions and the IEEE Standard 754" below.

Finally, the epilogue essentially reverses the actions of the prologue, restoring the processor state so that normal execution can be resumed.

The following example represents a typical exception handler. To link it with Example D-2 that will follow in Section D.4.3, "Example SIMD Floating-Point Emulation Implementation," assume that the body of the handler (not shown here in detail) passes the saved state to a routine that will examine in turn all the sub-operands of the excepting instruction, invoking a user floating-point exception handler if a particular set of sub-operands raises an unmasked (enabled) exception, or emulating the instruction otherwise.

**Example D-1. SIMD Floating-Point Exception Handler**

```
SIMD_FP_EXC_HANDLER PROC

;PROLOGUE
;SAVE REGISTERS THAT MIGHT BE USED BY THE EXCEPTION HANDLER
    PUSH EBP                ;SAVE EBP
    PUSH EAX                ;SAVE EAX
    ...
    MOV EBP, ESP            ;SAVE ESP in EBP
    SUB ESP, 512            ;ALLOCATE 512 BYTES
    AND ESP, 0FFFFFF0h     ;MAKE THE ADDRESS 16-BYTE ALIGNED
    FXSAVE [ESP]           ;SAVE FP, MMX, AND SIMD FP STATE
    PUSH [EBP+EFLAGS_OFFSET] ;COPY OLD EFLAGS TO STACK TOP
    POPFD                  ;RESTORE THE INTERRUPT ENABLE FLAG IF
                          ;TO VALUE BEFORE SIMD FP EXCEPTION

;BODY
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
    LDMXCSR LOCAL_MXCSR     ;LOAD LOCAL MXCSR VALUE IF NEEDED
    ...
;EPILOGUE
    FXRSTOR [ESP]          ;RESTORE MODIFIED STATE IMAGE
    MOV ESP, EBP           ;DE-ALLOCATE STACK SPACE
    ...
    POP EAX                ;RESTORE EAX
    POP EBP                ;RESTORE EBP
    IRET                   ;RETURN TO INTERRUPTED CALCULATION
SIMD_FP_EXC_HANDLER ENDP
```

## D.3 EXCEPTION SYNCHRONIZATION

An SSE/SSE2/SSE3 instruction can execute in parallel with other similar instructions, with integer instructions, and with floating-point or MMX instructions. Unlike for x87 instructions, special precaution for exception synchronization is not necessary in this case. This is because floating-point exceptions for SSE/SSE2/SSE3 instructions occur immediately and are not delayed until a subsequent floating-point instruction is executed. However, floating-point emulation may be necessary when unmasked floating-point exceptions are generated.

## D.4 SIMD FLOATING-POINT EXCEPTIONS AND THE IEEE STANDARD 754

SSE/SSE2/SSE3 extensions are 100% compatible with the IEEE Standard 754 for Binary Floating-Point Arithmetic, satisfying all of its mandatory requirements (when the flush-to-zero or denormals-are-zeros modes are not enabled). But a programming environment that includes SSE/SSE2/SSE3 instructions will comply with both the obligatory and the strongly recommended requirements of the IEEE Standard 754 regarding floating-point exception handling, only as a combination of hardware and software (which is acceptable). The standard states that a user should be able to request a trap on any of the five floating-point exceptions (note that the denormal exception is an IA-32 addition), and it also specifies the values (operands or result) to be delivered to the exception handler.

The main issue is that for SSE/SSE2/SSE3 instructions that raise post-computation exceptions (traps: overflow, underflow, or inexact), unlike for x87 FPU instructions, the processor does not provide the result recommended by IEEE Standard 754 to the user handler. If a user program needs the result of an instruction that generated a post-computation exception, it is the responsibility of the software to produce this result by emulating the faulting SSE/SSE2/SSE3 instruction. Another issue is that the standard does not specify explicitly how to handle multiple floating-point exceptions that occur simultaneously. For packed operations, a logical OR of the flags that would be set by each sub-operation is used to set the exception flags in the MXCSR. The following subsections present one possible way to solve these problems.

### D.4.1 Floating-Point Emulation

Every operating system must provide a kernel level floating-point exception handler (a template was presented in Section D.2, “Software Exception Handling” above). In the following discussion, assume that a user mode floating-point exception filter is supplied for SIMD floating-point exceptions (for example as part of a library of C functions), that a user program can invoke in order to handle unmasked exceptions. The user mode floating-point exception filter (not shown here) has to be able to emulate the subset of SSE/SSE2/SSE3 instructions that can generate numeric exceptions, and has to be able to invoke a user provided floating-point exception handler for floating-point exceptions. When a floating-point exception that is not masked is raised by an SSE/SSE2/SSE3 instruction, the low-level floating-point exception handler will be called. This low-level handler may in turn call the user mode floating-point exception filter. The filter function receives the original operands of the excepting instruction as no results are provided by the hardware, whether a pre-computation or a post-computation exception has occurred. The filter will unpack the operands into up to four sets of sub-operands, and will submit them one set at a time to an emulation function (See Example D-2 in Section D.4.3, “Example SIMD Floating-Point Emulation Implementation”). The emulation function will examine the sub-operands, and will possibly redo the necessary calculation.

Two cases are possible:

- If an unmasked (enabled) exception would occur in this process, the emulation function will return to its caller (the filter function) with the appropriate information. The filter will invoke a (previously registered) user floating-point exception handler for this set of sub-operands, and will record the result upon return from the user handler (provided the user handler allows continuation of the execution).
- If no unmasked (enabled) exception would occur, the emulation function will determine and will return to its caller the result of the operation for the current set of sub-operands (it has to be IEEE Standard 754 compliant). The filter function will record the result (plus any new flag settings).

The user level filter function will then call the emulation function for the next set of sub-operands (if any). When done with all the operand sets, the partial results will be packed (if the excepting instruction has a packed floating-point result, which is true for most SSE/SSE2/SSE3 numeric instructions) and the filter will return to the low-level exception handler, which in turn will return from the interruption, allowing execution to continue. Note that the

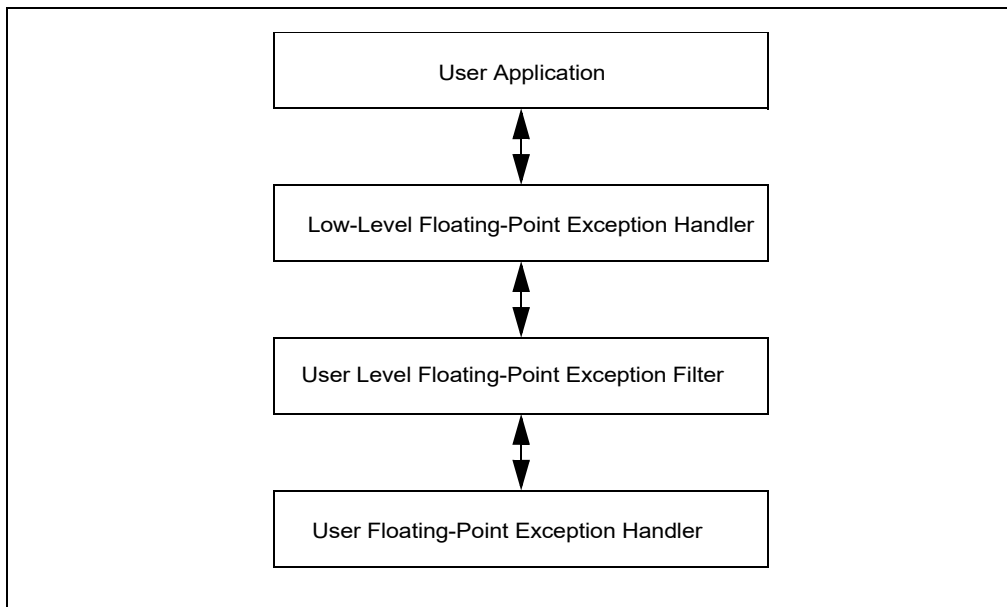
instruction pointer (EIP) has to be altered to point to the instruction following the excepting instruction, in order to continue execution correctly.

If a user mode floating-point exception filter is not provided, then all the work for decoding the excepting instruction, reading its operands, emulating the instruction for the components of the result that do not correspond to unmasked floating-point exceptions, and providing the compounded result will have to be performed by the user-provided floating-point exception handler.

Actual emulation might have to take place for one operand or pair of operands for scalar operations, and for all sub-operands or pairs of sub-operands for packed operations. The steps to perform are the following:

- The excepting instruction has to be decoded and the operands have to be read from the saved context.
- The instruction has to be emulated for each (pair of) sub-operand(s); if no floating-point exception occurs, the partial result has to be saved; if a masked floating-point exception occurs, the masked result has to be produced through emulation and saved, and the appropriate status flags have to be set; if an unmasked floating-point exception occurs, the result has to be generated by the user provided floating-point exception handler, and the appropriate status flags have to be set.
- The partial results have to be combined and written to the context that will be restored upon application program resumption.

A diagram of the control flow in handling an unmasked floating-point exception is presented below.



**Figure D-1. Control Flow for Handling Unmasked Floating-Point Exceptions**

From the user-level floating-point filter, Example D-2 in Section D.4.3, “Example SIMD Floating-Point Emulation Implementation,” will present only the floating-point emulation part. In order to understand the actions involved, the expected response to exceptions has to be known for all SSE/SSE2/SSE3 numeric instructions in two situations: with exceptions enabled (unmasked result), and with exceptions disabled (masked result). The latter can be found in Section 6.5, “Interrupts and Exceptions.” The response to NaN operands that do not raise an exception is specified in Section 4.8.3.4, “NaNs.” Operations on NaNs are explained in the same source. This response is also discussed in more detail in the next subsection, along with the unmasked and masked responses to floating-point exceptions.

## D.4.2 SSE/SSE2/SSE3 Response To Floating-Point Exceptions

This subsection specifies the unmasked response expected from the SSE/SSE2/SSE3 instructions that raise floating-point exceptions. The masked response is given in parallel, as it is necessary in the emulation process of



the instructions that raise unmasked floating-point exceptions. The response to NaN operands is also included in more detail than in Section 4.8.3.4, “NaNs.” For floating-point exception priority, refer to “Priority Among Simultaneous Exceptions and Interrupts” in Chapter 6, “Interrupt and Exception Handling,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### D.4.2.1 Numeric Exceptions

There are six classes of numeric (floating-point) exception conditions that can occur: Invalid operation (#I), Divide-by-Zero (#Z), Denormal Operand (#D), Numeric Overflow (#O), Numeric Underflow (#U), and Inexact Result (precision) (#P). #I, #Z, #D are pre-computation exceptions (floating-point faults), detected before the arithmetic operation. #O, #U, #P are post-computation exceptions (floating-point traps).

Users can control how the SSE/SSE2/SSE3 floating-point exceptions are handled by setting the mask/unmask bits in MXCSR. Masked exceptions are handled by the processor, or by software if they are combined with unmasked exceptions occurring in the same instruction. Unmasked exceptions are usually handled by the low-level exception handler, in conjunction with user-level software.

### D.4.2.2 Results of Operations with NaN Operands or a NaN Result for SSE/SSE2/SSE3 Numeric Instructions

The tables below (E-1 through E-10) specify the response of SSE/SSE2/SSE3 instructions to NaN inputs, or to other inputs that lead to NaN results.

These results will be referenced by subsequent tables (e.g., E-10). Most operations do not raise an invalid exception for quiet NaN operands, but even so, they will have higher precedence over raising floating-point exceptions other than invalid operation.

Note that the single precision QNaN Indefinite value is FFC00000H, the double precision QNaN Indefinite value is FFF8000000000000H, and the Integer Indefinite value is 80000000H (not a floating-point number, but it can be the result of a conversion instruction from floating-point to integer).

For an unmasked exception, no result will be provided by the hardware to the user handler. If a user registered floating-point exception handler is invoked, it may provide a result for the excepting instruction, that will be used if execution of the application code is continued after returning from the interruption.

In Tables D-1 through Table D-12, the specified operands cause an invalid exception, unless the unmasked result is marked with “not an exception”. In this latter case, the unmasked and masked results are the same.

**Table D-1. ADDPS, ADDSS, SUBPS, SUBSS, MULPS, MULSS, DIVPS, DIVSS, ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, ADDSUBPS, ADDSUBPD, HADDPS, HADDPD, HSUBPS, HSUBPD**

Source Operands	Masked Result	Unmasked Result
SNaN1 op <sup>1</sup> SNaN2	SNaN1   00400000H or SNaN1   0008000000000000H <sup>2</sup>	None
SNaN1 op QNaN2	SNaN1   00400000H or SNaN1   0008000000000000H <sup>2</sup>	None
QNaN1 op SNaN2	QNaN1	None
QNaN1 op QNaN2	QNaN1	QNaN1 (not an exception)
SNaN op real value	SNaN   00400000H or SNaN1   0008000000000000H <sup>2</sup>	None
Real value op SNaN	SNaN   00400000H or SNaN1   0008000000000000H <sup>2</sup>	None
QNaN op real value	QNaN	QNaN (not an exception)
Real value op QNaN	QNaN	QNaN (not an exception)

**Table D-1. ADDPS, ADDSS, SUBPS, SUBSS, MULPS, MULSS, DIVPS, DIVSS, ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, ADDSUBPS, ADDSUBPD, HADDPS, HADDPD, HSUBPS, HSUBPD (Contd.)**

Source Operands	Masked Result	Unmasked Result
Neither source operand is SNaN, but #I is signaled (e.g. for Inf - Inf, Inf * 0, Inf / Inf, 0/0)	Single precision or double precision QNaN Indefinite	None

**NOTES:**

1. For Tables E-1 to E-12: op denotes the operation to be performed.
2. SNaN | 00400000H is a quiet NaN in single precision format (if SNaN is in single precision) and SNaN | 0008000000000000H is a quiet NaN in double precision format (if SNaN is in double precision), obtained from the signaling NaN given as input.
3. Operations involving only quiet NaNs do not raise floating-point exceptions.

**Table D-2. CMPPS.EQ, CMPSS.EQ, CMPPS.ORD, CMPSS.ORD, CMPPD.EQ, CMPSD.EQ, CMPPD.ORD, CMPSD.ORD**

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	00000000H or 0000000000000000H <sup>1</sup>	00000000H or 0000000000000000H <sup>1</sup> (not an exception)
Opd1 op NaN (any Opd1)	00000000H or 0000000000000000H <sup>1</sup>	00000000H or 0000000000000000H <sup>1</sup> (not an exception)

**NOTE:**

1. 32-bit results are for single, and 64-bit results for double precision operations.

**Table D-3. CMPPS.NEQ, CMPSS.NEQ, CMPPS.UNORD, CMPSS.UNORD, CMPPD.NEQ, CMPSD.NEQ, CMPPD.UNORD, CMPSD.UNORD**

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	FFFFFFFFH or FFFFFFFFFFFFFFFFH <sup>1</sup>	FFFFFFFFH or FFFFFFFFFFFFFFFFH <sup>1</sup> (not an exception)
Opd1 op NaN (any Opd1)	FFFFFFFFH or FFFFFFFFFFFFFFFFH <sup>1</sup>	FFFFFFFFH or FFFFFFFFFFFFFFFFH <sup>1</sup> (not an exception)

**NOTE:**

1. 32-bit results are for single, and 64-bit results for double precision operations.

**Table D-4. CMPPS.LT, CMPSS.LT, CMPPS.LE, CMPSS.LE, CMPPD.LT, CMPSD.LT, CMPPD.LE, CMPSD.LE**

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	00000000H or 0000000000000000H <sup>1</sup>	None
Opd1 op NaN (any Opd1)	00000000H or 0000000000000000H <sup>1</sup>	None

**NOTE:**

1. 32-bit results are for single, and 64-bit results for double precision operations.

**Table D-5. CMPPS.NLT, CMPSS.NLT, CMPPS.NLE, CMPSS.NLE, CMPPD.NLT, CMPSD.NLT, CMPPD.NLE, CMPSD.NLE**

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	FFFFFFFFH or FFFFFFFFFFFFFFFFH <sup>1</sup>	None
Opd1 op NaN (any Opd1)	FFFFFFFFH or FFFFFFFFFFFFFFFFH <sup>1</sup>	None

**NOTE:**

1. 32-bit results are for single, and 64-bit results for double precision operations.

**Table D-6. COMISS, COMISD**

Source Operands	Masked Result	Unmasked Result
SNaN op Opd2 (any Opd2)	OF, SF, AF = 000 ZF, PF, CF = 111	None
Opd1 op SNaN (any Opd1)	OF, SF, AF = 000 ZF, PF, CF = 111	None
QNaN op Opd2 (any Opd2)	OF, SF, AF = 000 ZF, PF, CF = 111	None
Opd1 op QNaN (any Opd1)	OF, SF, AF = 000 ZF, PF, CF = 111	None

**Table D-7. UCOMISS, UCOMISD**

Source Operands	Masked Result	Unmasked Result
SNaN op Opd2 (any Opd2)	OF, SF, AF = 000 ZF, PF, CF = 111	None
Opd1 op SNaN (any Opd1)	OF, SF, AF = 000 ZF, PF, CF = 111	None
QNaN op Opd2 (any Opd2 ≠ SNaN)	OF, SF, AF = 000 ZF, PF, CF = 111	OF, SF, AF = 000 ZF, PF, CF = 111 (not an exception)
Opd1 op QNaN (any Opd1 ≠ SNaN)	OF, SF, AF = 000 ZF, PF, CF = 111	OF, SF, AF = 000 ZF, PF, CF = 111 (not an exception)

**Table D-8. CVTSP2PI, CVTSS2SI, CVTTPS2PI, CVTTSS2SI, CVTPD2PI, CVTSD2SI, CVTTPD2PI, CVTTSD2SI, CVTSP2DQ, CVTTPS2DQ, CVTPD2DQ, CVTTPD2DQ**

Source Operand	Masked Result	Unmasked Result
SNaN	80000000H or 8000000000000000 <sup>1</sup> (Integer Indefinite)	None
QNaN	80000000H or 8000000000000000 <sup>1</sup> (Integer Indefinite)	None

**NOTE:**

1. 32-bit results are for single, and 64-bit results for double precision operations.

**Table D-9. MAXPS, MAXSS, MINPS, MINSS, MAXPD, MAXSD, MINPD, MINSD**

Source Operands	Masked Result	Unmasked Result
Opd1 op NaN2 (any Opd1)	NaN2	None
NaN1 op Opd2 (any Opd2)	Opd2	None

**NOTE:**

1. SNaN and QNaN operands raise an Invalid Operation fault.

**Table D-10. SQRTPS, SQRTSS, SQRTPD, SQRTSD**

Source Operand	Masked Result	Unmasked Result
QNaN	QNaN	QNaN (not an exception)
SNaN	SNaN   00400000H or SNaN   0008000000000000H <sup>1</sup>	None
Source operand is not SNaN; but #I is signaled (e.g. for sqrt (-1.0))	Single precision or double precision QNaN Indefinite	None

**NOTE:**

1. SNaN | 00400000H is a quiet NaN in single precision format (if SNaN is in single precision) and SNaN | 0008000000000000H is a quiet NaN in double precision format (if SNaN is in double precision), obtained from the signaling NaN given as input.

**Table D-11. CVTPS2PD, CVTSS2SD**

Source Operands	Masked Result	Unmasked Result
QNaN	QNaN <sup>1</sup>	QNaN <sup>1</sup> (not an exception)
SNaN	QNaN <sup>2</sup>	None

**NOTES:**

1. The double precision output QNaN<sup>1</sup> is created from the single precision input QNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 11-bit exponent 7FFH, and the 24-bit significand is extended to a 53-bit significand by appending 29 bits equal to 0.
2. The double precision output QNaN<sup>1</sup> is created from the single precision input SNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 11-bit exponent 7FFH, and the 24-bit significand is extended to a 53-bit significand by pending 29 bits equal to 0. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

**Table D-12. CVTPD2PS, CVTSD2SS**

Source Operands	Masked Result	Unmasked Result
QNaN	QNaN <sup>1</sup>	QNaN <sup>1</sup> (not an exception)
SNaN	QNaN <sup>2</sup>	None

**NOTES:**

1. The single precision output QNaN<sup>1</sup> is created from the double precision input QNaN as follows: the sign bit is preserved, the 11-bit exponent 7FFH is replaced by the 8-bit exponent FFH, and the 53-bit significand is truncated to a 24-bit significand by removing its 29 least significant bits.
2. The single precision output QNaN<sup>1</sup> is created from the double precision input SNaN as follows: the sign bit is preserved, the 11-bit exponent 7FFH is replaced by the 8-bit exponent FFH, and the 53-bit significand is truncated to a 24-bit significand by removing its 29 least significant bits. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

### D.4.2.3 Condition Codes, Exception Flags, and Response for Masked and Unmasked Numeric Exceptions

In the following, the masked response is what the processor provides when a masked exception is raised by an SSE/SSE2/SSE3 numeric instruction. The same response is provided by the floating-point emulator for SSE/SSE2/SSE3 numeric instructions, when certain components of the quadruple input operands generate exceptions that are masked (the emulator also generates the correct answer, as specified by IEEE Standard 754 whenever applicable, in the case when no floating-point exception occurs). The unmasked response is what the emulator provides to the user handler for those components of the packed operands of SSE/SSE2/SSE3 instructions that raise unmasked exceptions. Note that for pre-computation exceptions (floating-point faults), no result is provided to the user handler. For post-computation exceptions (floating-point traps), a result is provided to the user handler, as specified below.

In the following tables, the result is denoted by 'res', with the understanding that for the actual instruction, the destination coincides with the first source operand (except for COMISS, UCOMISS, COMISD, and UCOMISD, whose destination is the EFLAGS register).

**Table D-13. #I - Invalid Operations**

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDPD ADDSS ADDSD HADDPS HADDPD	src1 or src2 <sup>1</sup> = SNaN	Refer to Table D-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
ADDSUBPS (the addition component) ADDSUBPD (the addition component)	src1 = +Inf, src2 = -Inf or src1 = -Inf, src2 = +Inf	res <sup>1</sup> = QNaN Indefinite, #IA = 1	
SUBPS SUBPD SUBSS SUBSD HSUBPS HSUBPD	src1 or src2 = SNaN	Refer to Table D-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
ADDSUBPS (the subtraction component) ADDSUBPD (the subtraction component)	src1 = +Inf, src2 = +Inf or src1 = -Inf, src2 = -Inf	res = QNaN Indefinite, #IA = 1	
MULPS MULPD	src1 or src2 = SNaN	Refer to Table D-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
MULSS MULSD	src1 = ±Inf, src2 = ±0 or src1 = ±0, src2 = ±Inf	res = QNaN Indefinite, #IA = 1	
DIVPS DIVPD	src1 or src2 = SNaN	Refer to Table D-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
DIVSS DIVSD	src1 = ±Inf, src2 = ±Inf or src1 = ±0, src2 = ±0	res = QNaN Indefinite, #IA = 1	
SQRTPS SQRTPD SQRTSS SQRTSD	src = SNaN	Refer to Table D-10 for NaN operands, #IA = 1	src unchanged, #IA = 1
	src < 0 (note that -0 < 0 is false)	res = QNaN Indefinite, #IA = 1	

**Table D-13. #I - Invalid Operations (Contd.)**

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
MAXPS MAXSS MAXPD MAXSD	src1 = NaN or src2 = NaN	res = src2, #IA = 1	src1, src2 unchanged; #IA = 1
MINPS MINSS MINPD MINSF	src1 = NaN or src2 = NaN	res = src2, #IA = 1	src1, src2 unchanged; #IA = 1
CMPPS.LT CMPPS.LE CMPPS.NLT CMPPS.NLE CMPSS.LT CMPSS.LE CMPSS.NLT CMPSS.NLE CMPPD.LT CMPPD.LE CMPPD.NLT CMPPD.NLE CMPSD.LT CMPSD.LE CMPSD.NLT CMPSD.NLE	src1 = NaN or src2 = NaN	Refer to Table D-4 and Table D-5 for NaN operands; #IA = 1	src1, src2 unchanged; #IA = 1
COMISS COMISD	src1 = NaN or src2 = NaN	Refer to Table D-6 for NaN operands	src1, src2, EFLAGS unchanged; #IA = 1
UCOMISS UCOMISD	src1 = SNaN or src2 = SNaN	Refer to Table D-7 for NaN operands	src1, src2, EFLAGS unchanged; #IA = 1
CVTTPS2PI CVTTSS2SI CVTPD2PI CVTSD2SI CVTTPS2DQ CVTPD2DQ	src = NaN, ±Inf, or   (src) <sub>rnd</sub>   > 7FFFFFFFH and (src) <sub>rnd</sub> ≠ 80000000H  See Note <sup>2</sup> for information on rnd.	res = Integer Indefinite, #IA = 1	src unchanged, #IA = 1
CVTTTPS2PI CVTTTSS2SI CVTTTPD2PI CVTTTSD2SI CVTTTPS2DQ CVTTTPD2DQ	src = NaN, ±Inf, or   (src) <sub>rz</sub>   > 7FFFFFFFH and (src) <sub>rz</sub> ≠ 80000000H  See Note <sup>2</sup> for information on rz.	res = Integer Indefinite, #IA = 1	src unchanged, #IA = 1

Table D-13. #I - Invalid Operations (Contd.)

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
CVTPS2PD CVTSS2SD	src = SNAN	Refer to Table D-11 for NaN operands	src unchanged, #IA = 1
CVTPD2PS CVTSD2SS	src = SNAN	Refer to Table D-12 for NaN operands	src unchanged, #IA = 1

**NOTES:**

- For Tables E-13 to E-18:
  - src denotes the single source operand of a unary operation.
  - src1, src2 denote the first and second source operand of a binary operation.
  - res denotes the numerical result of an operation.
- rnd signifies the user rounding mode from MXCSR, and rz signifies the rounding mode toward zero. (truncate), when rounding a floating-point value to an integer. For more information, refer to Table 4-8.
- For NAN encodings, see Table 4-3.

Table D-14. #Z - Divide-by-Zero

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
DIVPS DIVSS DIVPD DIVPS	src1 = finite non-zero (normal, or denormal) src2 = $\pm 0$	res = $\pm \text{Inf}$ , #ZE = 1	src1, src2 unchanged; #ZE = 1

**Table D-15. #D - Denormal Operand**

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDPD ADDSUBPS ADDSUBPD HADDPS HADDPD SUBPS SUBPD HSUBPS HSUBPD MULPS MULPD DIVPS DIVPD SQRTPS SQRTPD MAXPS MAXPD MINPS MINPD ADDSS ADDSD SUBSS SUBSD MULSS MULSD DIVSS DIVSD SQRTSS SQRTSD MAXSS MAXSD MINSS MINS CVTSS2SD CVTSS2SS CVTPD2PS CVTSD2SS	src1 = denormal <sup>1</sup> or src2 = denormal (and the DAZ bit in MXCSR is 0)	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked post-computation exception occurs; #DE = 1.	src1, src2 unchanged; #DE = 1  Note that SQRT, CVTSS2SD, CVTSS2SS, CVTPD2PS, CVTSD2SS have only 1 src.
CMPPS CMPPD CMPSS CMPSD	src1 = denormal <sup>1</sup> or src2 = denormal (and the DAZ bit in MXCSR is 0)	Comparison result, stored in the destination register; #DE = 1	src1, src2 unchanged; #DE = 1
COMISS COMISD UCOMISS UCOMISD	src1 = denormal <sup>1</sup> or src2 = denormal (and the DAZ bit in MXCSR is 0)	Comparison result, stored in the EFLAGS register; #DE = 1	src1, src2 unchanged; #DE = 1

**NOTE:**

1. For denormal encodings, see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers."



**Table D-16. #0 - Numeric Overflow**

Instruction	Condition	Masked Response			Unmasked Response and Exception Code
		Rounding	Sign	Result & Status Flags	
ADDPS ADDSUBPS HADDPS SUBPS HSUBPS MULPS DIVPS ADDSS SUBSS MULSS DIVSS CVTPD2PS CVTSD2SS	Rounded result > largest single precision finite normal value	To nearest	+ -	#OE = 1, #PE = 1 res = +∞ res = -∞	res = (result calculated with unbounded exponent and rounded to the destination precision) / 2 <sup>192</sup> #OE = 1 #PE = 1 if the result is inexact
		Toward -∞	+ -	#OE = 1, #PE = 1 res = 1.11...1 * 2 <sup>127</sup> res = -∞	
		Toward +∞	+ -	#OE = 1, #PE = 1 res = +∞ res = -1.11...1 * 2 <sup>127</sup>	
		Toward 0	+ -	#OE = 1, #PE = 1 res = 1.11...1 * 2 <sup>127</sup> res = -1.11...1 * 2 <sup>127</sup>	
ADDPD ADDSUBPD HADDPD SUBPD HSUBPD MULPD DIVPD ADDSD SUBSD MULSD DIVSD	Rounded result > largest double precision finite normal value	To nearest	+ -	#OE = 1, #PE = 1 res = +∞ res = -∞	res = (result calculated with unbounded exponent and rounded to the destination precision) / 2 <sup>1536</sup> ▪ #OE = 1 ▪ #PE = 1 if the result is inexact
		Toward -∞	+ -	#OE = 1, #PE = 1 res = 1.11...1 * 2 <sup>1023</sup> res = -∞	
		Toward +∞	+ -	#OE = 1, #PE = 1 res = +∞ res = -1.11...1 * 2 <sup>1023</sup>	
		Toward 0	+ -	#OE = 1, #PE = 1 res = 1.11...1 * 2 <sup>1023</sup> res = -1.11...1 * 2 <sup>1023</sup>	

**Table D-17. #U - Numeric Underflow**

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDSUBPS HADDPS SUBPS HSUBPS MULPS DIVPS ADDSS SUBSS MULSS DIVSS CVTPD2PS CVTSD2SS	Result calculated with unbounded exponent and rounded to the destination precision < smallest single precision finite normal value.	res = ±0, denormal, or normal  #UE = 1 and #PE = 1, but only if the result is inexact	res = (result calculated with unbounded exponent and rounded to the destination precision) * 2 <sup>192</sup> <ul style="list-style-type: none"> <li>▪ #UE = 1</li> <li>▪ #PE = 1 if the result is inexact</li> </ul>
ADDPD ADDSUBPD HADDPD SUBPD HSUBPD MULPD DIVPD ADDSD SUBSD MULSD DIVSD	Result calculated with unbounded exponent and rounded to the destination precision < smallest double precision finite normal value.	res = ±0, denormal or normal  #UE = 1 and #PE = 1, but only if the result is inexact	res = (result calculated with unbounded exponent and rounded to the destination precision) * 2 <sup>1536</sup> <ul style="list-style-type: none"> <li>▪ #UE = 1</li> <li>▪ #PE = 1 if the result is inexact</li> </ul>

Table D-18. #P - Inexact Result (Precision)

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDPD ADDSUBPS ADDSUBPD HADDPS HADDPD SUBPS SUBPD HSUBPS HSUBPD MULPS MULPD DIVPS DIVPD SQRTPS SQRTPD CVTDQ2PS CVTPI2PS CVTSS2PS CVTSS2DQ CVTPD2PI CVTPD2DQ CVTPD2PS CVTTPS2PI CVTTPD2PI CVTTPD2DQ CVTTPS2DQ ADDSS ADDSD SUBSS SUBSD MULSS MULSD DIVSS DIVSD SQRTSS SQRTSD CVTSS2SS CVTSS2SI CVTSD2SI CVTSD2SS CVTSS2SI CVTSD2SI	The result is not exactly representable in the destination format.	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked underflow or overflow conditions occur (this exception can occur in the presence of a masked underflow or overflow); #PE = 1.	Only if no underflow/overflow condition occurred, or if the corresponding exceptions are masked: <ul style="list-style-type: none"> <li>▪ Set #OE if masked overflow and set result as described above for masked overflow.</li> <li>▪ Set #UE if masked underflow and set result as described above for masked underflow.</li> </ul> If neither underflow nor overflow, res equals the result rounded to the destination precision and using the bounded exponent set #PE = 1.

### D.4.3 Example SIMD Floating-Point Emulation Implementation

The sample code listed below may be considered as being part of a user-level floating-point exception filter for the SSE/SSE2/SSE3 numeric instructions. It is assumed that the filter function is invoked by a low-level exception handler (invoked for exception 19 when an unmasked floating-point exception occurs), and that it operates as explained in Section D.4.1, "Floating-Point Emulation." The sample code does the emulation only for the SSE instructions for addition, subtraction, multiplication, and division. For this, it uses C code and x87 FPU operations. Operations corresponding to other SSE/SSE2/SSE3 numeric instructions can be emulated similarly. The example assumes that the emulation function receives a pointer to a data structure specifying a number of input parameters: the operation that caused the exception, a set of sub-operands (unpacked, of type float), the rounding mode

(the precision is always single), exception masks (having the same relative bit positions as in the MXCSR but starting from bit 0 in an unsigned integer), and flush-to-zero and denormals-are-zeros indicators.

The output parameters are a floating-point result (of type float), the cause of the exception (identified by constants not explicitly defined below), and the exception status flags. The corresponding C definition is:

```
typedef struct {
    unsigned int operation;           //SSE or SSE2 operation: ADDPS, ADDSS, ...
    unsigned int operand1_uint32; //first operand value
    unsigned int operand2_uint32; //second operand value (if any)
    float result_fval; // result value (if any)
    unsigned int rounding_mode; //rounding mode
    unsigned int exc_masks; //exception masks, in the order P,U,O,Z,D,I
    unsigned int exception_cause; //exception cause
    unsigned int status_flag_inexact; //inexact status flag
    unsigned int status_flag_underflow; //underflow status flag
    unsigned int status_flag_overflow; //overflow status flag
    unsigned int status_flag_divide_by_zero;
                                     //divide by zero status flag
    unsigned int status_flag_denormal_operand;
                                     //denormal operand status flag
    unsigned int status_flag_invalid_operation;
                                     //invalid operation status flag

    unsigned int ftz; // flush-to-zero flag
    unsigned int daz; // denormals-are-zeros flag
} EXC_ENV;
```

The arithmetic operations exemplified are emulated as follows:

1. If the denormals-are-zeros mode is enabled (the DAZ bit in MXCSR is set to 1), replace all the denormal inputs with zeroes of the same sign (the denormal flag is not affected by this change).
2. Perform the operation using x87 FPU instructions, with exceptions disabled, the original user rounding mode, and single precision. This reveals invalid, denormal, or divide-by-zero exceptions (if there are any) and stores the result in memory as a double precision value (whose exponent range is large enough to look like “unbounded” to the result of the single precision computation).
3. If no unmasked exceptions were detected, determine if the magnitude of the result is less than the smallest normal number that can be represented in single precision format, or greater than the largest normal number that can be represented in single precision format (huge). If an unmasked overflow or underflow occurs, calculate the scaled result that will be handed to the user exception handler, as specified by IEEE Standard 754.
4. If no exception was raised, calculate the result with a “bounded” exponent. If the result is tiny, it requires denormalization (shifting the significand right while incrementing the exponent to bring it into the admissible range of [-126,+127] for single precision floating-point numbers).

The result obtained in step 2 cannot be used because it might incur a double rounding error (it was rounded to 24 bits in step 2, and might have to be rounded again in the denormalization process). To overcome this is, calculate the result as a double precision value, and store it to memory in single precision format.

Rounding first to 53 bits in the significand, and then to 24 never causes a double rounding error (exact properties exist that state when double-rounding error occurs, but for the elementary arithmetic operations, the rule of thumb is that if an infinitely precise result is rounded to  $2p+1$  bits and then again to  $p$  bits, the result is the same as when rounding directly to  $p$  bits, which means that no double-rounding error occurs).

5. If the result is inexact and the inexact exceptions are unmasked, the calculated result will be delivered to the user floating-point exception handler.
6. The flush-to-zero case is dealt with if the result is tiny.

7. The emulation function returns RAISE\_EXCEPTION to the filter function if an exception has to be raised (the exception\_cause field indicates the cause). Otherwise, the emulation function returns DO\_NOT\_RAISE\_EXCEPTION. In the first case, the result is provided by the user exception handler called by the filter function. In the second case, it is provided by the emulation function. The filter function has to collect all the partial results, and to assemble the scalar or packed result that is used if execution is to continue.

### Example D-2. SIMD Floating-Point Emulation

```
// masks for individual status word bits
#define PRECISION_MASK 20H
#define UNDERFLOW_MASK 10H
#define OVERFLOW_MASK 08H
#define ZERODIVIDE_MASK 04H
#define DENORMAL_MASK 02H
#define INVALID_MASK 01H

// 32-bit constants
static unsigned ZERO_ARRAY[] = {00000000H};
#define ZERO *(float *) ZERO_ARRAY
    // +0.0
static unsigned NZERO_ARRAY[] = {80000000H};
#define NZERO *(float *) NZERO_ARRAY
    // -0.0
static unsigned POSINFF_ARRAY[] = {7f800000H};
#define POSINFF *(float *) POSINFF_ARRAY
    // +Inf
static unsigned NEGINFF_ARRAY[] = {ff800000H};
#define NEGINFF *(float *) NEGINFF_ARRAY
    // -Inf

// 64-bit constants
static unsigned MIN_SINGLE_NORMAL_ARRAY [] = {00000000H, 38100000H};
#define MIN_SINGLE_NORMAL *(double *)MIN_SINGLE_NORMAL_ARRAY
    // +1.0 * 2^-126
static unsigned MAX_SINGLE_NORMAL_ARRAY [] = {70000000H, 47efffffH};
#define MAX_SINGLE_NORMAL *(double *)MAX_SINGLE_NORMAL_ARRAY
    // +1.1...1*2^127
static unsigned TWO_TO_192_ARRAY[] = {00000000H, 4bf00000H};
#define TWO_TO_192 *(double *)TWO_TO_192_ARRAY
    // +1.0 * 2^192
static unsigned TWO_TO_M192_ARRAY[] = {00000000H, 33f00000H};
#define TWO_TO_M192 *(double *)TWO_TO_M192_ARRAY
    // +1.0 * 2^-192

// auxiliary functions
static int isnanf (unsigned int ); // returns 1 if f is a NaN, and 0 otherwise
static float quietf (unsigned int ); // converts a signaling NaN to a quiet
    // NaN, and leaves a quiet NaN unchanged
static unsigned int check_for_daz (unsigned int ); // converts denormals
    // to zeros of the same sign;
    // does not affect any status flags

// emulation of SSE and SSE2 instructions using
// C code and x87 FPU instructions

unsigned int
simd_fp_emulate (EXC_ENV *exc_env)
{
    int uiopd1; // first operand of the add, subtract, multiply, or divide
    int uiopd2; // second operand of the add, subtract, multiply, or divide
    float res; // result of the add, subtract, multiply, or divide
    double dbl_res24; // result with 24-bit significand, but "unbounded" exponent
```

## GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

```
// (needed to check tininess, to provide a scaled result to
// an underflow/overflow trap handler, and in flush-to-zero mode)
double dbl_res; // result in double precision format (needed to avoid a
// double rounding error when denormalizing)
unsigned int result_tiny;
unsigned int result_huge;
unsigned short int sw; // 16 bits
unsigned short int cw; // 16 bits

// have to check first for faults (V, D, Z), and then for traps (O, U, I)

// initialize x87 FPU (floating-point exceptions are masked)
_asm {
    fninit;
}

result_tiny = 0;
result_huge = 0;

switch (exc_env->operation) {

    case ADDPS:
    case ADDSS:
    case SUBPS:
    case SUBSS:
    case MULPS:
    case MULSS:
    case DIVPS:
    case DIVSS:

        uiopd1 = exc_env->operand1_uint32; // copy as unsigned int
        // do not copy as float to avoid conversion
        // of SNaN to QNaN by compiled code
        uiopd2 = exc_env->operand2_uint32;
        // do not copy as float to avoid conversion of SNaN
        // to QNaN by compiled code
        uiopd1 = check_for_daz (uiopd1); // operand1 = +0.0 * operand1 if it is
        // denormal and DAZ=1
        uiopd2 = check_for_daz (uiopd2); // operand2 = +0.0 * operand2 if it is
        // denormal and DAZ=1

        // execute the operation and check whether the invalid, denormal, or
        // divide by zero flags are set and the respective exceptions enabled

        // set control word with rounding mode set to exc_env->rounding_mode,
        // single precision, and all exceptions disabled
        switch (exc_env->rounding_mode) {
            case ROUND_TO_NEAREST:
                cw = 003fH; // round to nearest, single precision, exceptions masked
                break;
            case ROUND_DOWN:
                cw = 043fH; // round down, single precision, exceptions masked
                break;
            case ROUND_UP:
                cw = 083fH; // round up, single precision, exceptions masked
                break;
            case ROUND_TO_ZERO:
                cw = 0c3fH; // round to zero, single precision, exceptions masked
                break;
            default:
                ;
        }
        _asm {
            fldcw WORD PTR cw;
        }
    }
}
```

```

}

// compute result and round to the destination precision, with
// "unbounded" exponent (first IEEE rounding)
switch (exc_env->operation) {

case ADDPS:
case ADDSS:
    // perform the addition
    __asm {
        fnclex;
        // load input operands
        fld DWORD PTR uiopd1; // may set denormal or invalid status flags
        fld DWORD PTR uiopd2; // may set denormal or invalid status flags
        faddp st(1), st(0); // may set inexact or invalid status flags
        // store result
        fstp QWORD PTR dbl_res24; // exact
    }
    break;

case SUBPS:
case SUBSS:
    // perform the subtraction
    __asm {
        fnclex;
        // load input operands
        fld DWORD PTR uiopd1; // may set denormal or invalid status flags
        fld DWORD PTR uiopd2; // may set denormal or invalid status flags
        fsubp st(1), st(0); // may set the inexact or invalid status flags

        // store result
        fstp QWORD PTR dbl_res24; // exact
    }
    break;

case MULPS:
case MULSS:
    // perform the multiplication
    __asm {
        fnclex;
        // load input operands
        fld DWORD PTR uiopd1; // may set denormal or invalid status flags
        fld DWORD PTR uiopd2; // may set denormal or invalid status flags
        fmulp st(1), st(0); // may set inexact or invalid status flags

        // store result
        fstp QWORD PTR dbl_res24; // exact
    }
    break;

case DIVPS:
case DIVSS:
    // perform the division
    __asm {
        fnclex;
        // load input operands
        fld DWORD PTR uiopd1; // may set denormal or invalid status flags
        fld DWORD PTR uiopd2; // may set denormal or invalid status flags
        fdivp st(1), st(0); // may set the inexact, divide by zero, or
        // invalid status flags

        // store result
        fstp QWORD PTR dbl_res24; // exact
    }
    break;
}

```

## GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

```
        default:
            ; // will never occur

    }

    // read status word
    __asm {
        fstsw WORD PTR sw;
    }

    if (sw & ZERODIVIDE_MASK)
    sw = sw & ~DENORMAL_MASK; // clear D flag for (denormal / 0)

    // if invalid flag is set, and invalid exceptions are enabled, take trap
    if (!(exc_env->exc_masks & INVALID_MASK) && (sw & INVALID_MASK)) {
        exc_env->status_flag_invalid_operation = 1;
        exc_env->exception_cause = INVALID_OPERATION;
        return (RAISE_EXCEPTION);
    }

    // checking for NaN operands has priority over denormal exceptions;
    // also fix for the SSE and SSE2
    // differences in treating two NaN inputs between the
    // instructions and other IA-32 instructions
    if (isnanf (uiopd1) || isnanf (uiopd2)) {

        if (isnanf (uiopd1) && isnanf (uiopd2))
            exc_env->result_fval = quietf (uiopd1);
        else
            exc_env->result_fval = (float)dbl_res24; // exact

        if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
        return (DO_NOT_RAISE_EXCEPTION);
    }

    // if denormal flag set, and denormal exceptions are enabled, take trap
    if (!(exc_env->exc_masks & DENORMAL_MASK) && (sw & DENORMAL_MASK)) {
        exc_env->status_flag_denormal_operand = 1;
        exc_env->exception_cause = DENORMAL_OPERAND;
        return (RAISE_EXCEPTION);
    }

    // if divide by zero flag set, and divide by zero exceptions are
    // enabled, take trap (for divide only)
    if (!(exc_env->exc_masks & ZERODIVIDE_MASK) && (sw & ZERODIVIDE_MASK)) {
        exc_env->status_flag_divide_by_zero = 1;
        exc_env->exception_cause = DIVIDE_BY_ZERO;
        return (RAISE_EXCEPTION);
    }

    // done if the result is a NaN (QNaN Indefinite)
    res = (float)dbl_res24;
    if (isnanf (*(unsigned int *)&res)) {
        exc_env->result_fval = res; // exact
        exc_env->status_flag_invalid_operation = 1;
        return (DO_NOT_RAISE_EXCEPTION);
    }

    // dbl_res24 is not a NaN at this point

    if (sw & DENORMAL_MASK) exc_env->status_flag_denormal_operand = 1;

    // Note: (dbl_res24 == 0.0 && sw & PRECISION_MASK) cannot occur
    if (-MIN_SINGLE_NORMAL < dbl_res24 && dbl_res24 < 0.0 ||
        0.0 < dbl_res24 && dbl_res24 < MIN_SINGLE_NORMAL) {
```



```

    result_tiny = 1;
}

// check if the result is huge
if (NEGINFF < dbl_res24 && dbl_res24 < -MAX_SINGLE_NORMAL ||
    MAX_SINGLE_NORMAL < dbl_res24 && dbl_res24 < POSINFF) {
    result_huge = 1;
}

// at this point, there are no enabled I,D, or Z exceptions
// to take; the instr.
// might lead to an enabled underflow, enabled underflow and inexact,
// enabled overflow, enabled overflow and inexact, enabled inexact, or
// none of these; if there are no U or O enabled exceptions, re-execute
// the instruction using IA-32 double precision format, and the
// user's rounding mode; exceptions must have
// been disabled before calling
// this function; an inexact exception may be reported on the 53-bit
// fsubp, fmulp, or on both the 53-bit and 24-bit conversions, while an
// overflow or underflow (with traps disabled) may be reported on the
// conversion from dbl_res to res

// check whether there is an underflow, overflow,
// or inexact trap to be taken
// if the underflow traps are enabled and the result is
// tiny, take underflow trap

if (!(exc_env->exc_masks & UNDERFLOW_MASK) && result_tiny) {
    dbl_res24 = TWO_TO_192 * dbl_res24; // exact
    exc_env->status_flag_underflow = 1;
    exc_env->exception_cause = UNDERFLOW;
    exc_env->result_fval = (float)dbl_res24; // exact
    if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
    return (RAISE_EXCEPTION);
}

// if overflow traps are enabled and the result is huge, take
// overflow trap
if (!(exc_env->exc_masks & OVERFLOW_MASK) && result_huge) {
    dbl_res24 = TWO_TO_M192 * dbl_res24; // exact
    exc_env->status_flag_overflow = 1;
    exc_env->exception_cause = OVERFLOW;
    exc_env->result_fval = (float)dbl_res24; // exact
    if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
    return (RAISE_EXCEPTION);
}

// set control word with rounding mode set to exc_env->rounding_mode,
// double precision, and all exceptions disabled
cw = cw | 0200H; // set precision to double
__asm {
    fldcw WORD PTR cw;
}

switch (exc_env->operation) {

    case ADDPS:
    case ADDSS:
        // perform the addition
        __asm {
            // load input operands
            fld DWORD PTR uiopd1; // may set the denormal status flag
            fld DWORD PTR uiopd2; // may set the denormal status flag
            faddp st(1), st(0); // rounded to 53 bits, may set the inexact
                                // status flag
        }

```

## GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

```

    // store result
    fstp QWORD PTR dbl_res; // exact, will not set any flag
}
break;

case SUBPS:
case SUBSS:
    // perform the subtraction
    __asm {
        // load input operands
        fld DWORD PTR uiopd1; // may set the denormal status flag
        fld DWORD PTR uiopd2; // may set the denormal status flag
        fsubp st(1), st(0); // rounded to 53 bits, may set the inexact
                            // status flag

        // store result
        fstp QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

case MULPS:
case MULSS:
    // perform the multiplication
    __asm {
        // load input operands
        fld DWORD PTR uiopd1; // may set the denormal status flag
        fld DWORD PTR uiopd2; // may set the denormal status flag
        fmulp st(1), st(0); // rounded to 53 bits, exact

        // store result
        fstp QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

case DIVPS:
case DIVSS:
    // perform the division
    __asm {
        // load input operands
        fld DWORD PTR uiopd1; // may set the denormal status flag
        fld DWORD PTR uiopd2; // may set the denormal status flag
        fdivp st(1), st(0); // rounded to 53 bits, may set the inexact
                            // status flag

        // store result
        fstp QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

default:
    ; // will never occur
}

// calculate result for the case an inexact trap has to be taken, or
// when no trap occurs (second IEEE rounding)
res = (float)dbl_res;
    // may set P, U or O; may also involve denormalizing the result

// read status word
__asm {
    fstsw WORD PTR sw;
}

// if inexact traps are enabled and result is inexact, take inexact trap
if (!(exc_env->exc_masks & PRECISION_MASK) &&
    ((sw & PRECISION_MASK) || (exc_env->ftz && result_tiny))) {
    exc_env->status_flag_inexact = 1;
}

```

```

exc_env->exception_cause = INEXACT;
if (result_tiny) {
    exc_env->status_flag_underflow = 1;

    // if ftz = 1 and result is tiny, result = 0.0
    // (no need to check for underflow traps disabled: result tiny and
    // underflow traps enabled would have caused taking an underflow
    // trap above)
    if (exc_env->ftz) {
        if (res > 0.0)
            res = ZEROF;
        else if (res < 0.0)
            res = NZEROF;
        // else leave res unchanged
    }
}
if (result_huge) exc_env->status_flag_overflow = 1;
exc_env->result_fval = res;
return (RAISE_EXCEPTION);
}

// if it got here, then there is no trap to be taken; the following must
// hold: ((the MXCSR U exceptions are disabled or
//
// the MXCSR underflow exceptions are enabled and the underflow flag is
// clear and (the inexact flag is set or the inexact flag is clear and
// the 24-bit result with unbounded exponent is not tiny))
// and (the MXCSR overflow traps are disabled or the overflow flag is
// clear) and (the MXCSR inexact traps are disabled or the inexact flag
// is clear)
//
// in this case, the result has to be delivered (the status flags are
// sticky, so they are all set correctly already)

// read status word to see if result is inexact
__asm {
    fstsw WORD PTR sw;
}

if (sw & UNDERFLOW_MASK) exc_env->status_flag_underflow = 1;
if (sw & OVERFLOW_MASK) exc_env->status_flag_overflow = 1;
if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;

// if ftz = 1, and result is tiny (underflow traps must be disabled),
// result = 0.0
if (exc_env->ftz && result_tiny) {
    if (res > 0.0)
        res = ZEROF;
    else if (res < 0.0)
        res = NZEROF;
    // else leave res unchanged

    exc_env->status_flag_inexact = 1;
    exc_env->status_flag_underflow = 1;
}

exc_env->result_fval = res;
if (sw & ZERODIVIDE_MASK) exc_env->status_flag_divide_by_zero = 1;
if (sw & DENORMAL_MASK) exc_env->status_flag_denormal = 1;
if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
return (DO_NOT_RAISE_EXCEPTION);

break;

case CMPPS:

```

## GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

```
case CMPSS:
    ...
    break;

case COMISS:
case UCOMISS:
    ...
    break;

case CVTPI2PS:
case CVTSI2SS:
    ...
    break;

case CVTPS2PI:
case CVTSS2SI:
case CVTTPS2PI:
case CVTTSS2SI:
    ...
    break;

case MAXPS:
case MAXSS:
case MINPS:
case MINSS:
    ...
    break;

case SQRTPS:
case SQRTSS:
    ...
    break;
...
case UNSPEC:
    ...
    break;

default:
    ...
}
}
```

## 7. Updates to Chapter 1, Volume 2A

Change bars and green text show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*.

-----  
Changes to this chapter: Updated section 1.1 "Intel® 64 and IA-32 Processors Covered in this Manual". Removed link to outdated reference document.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569) are part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 253665).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

## 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series

## ABOUT THIS MANUAL

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family
- 8th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
- Intel® Xeon® E processors
- 9th generation Intel® Core™ processors
- 2nd generation Intel® Xeon® Processor Scalable Family

- 10th generation Intel® Core™ processors
- 11th generation Intel® Core™ processors
- 3rd generation Intel® Xeon® Processor Scalable Family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Airmont microarchitecture.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.



The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor C series, the Intel® Atom™ processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Processor Scalable Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Processor Scalable Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF VOLUME 2A, 2B, 2C AND 2D: INSTRUCTION SET REFERENCE

A description of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D* content follows:

**Chapter 1 — About This Manual.** Gives an overview of all seven volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — Instruction Format.** Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

**Chapter 3 — Instruction Set Reference, A-L.** Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3/SSSE3/SSE4 extensions, and system instructions are included.

**Chapter 4 — Instruction Set Reference, M-U.** Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

**Chapter 5 — Instruction Set Reference, V-Z.** Continues the description of Intel 64 and IA-32 instructions started in chapters 3 and 4. It provides the balance of the alphabetized list of instructions and starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

**Chapter 6 — Safer Mode Extensions Reference.** Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions. This chapter starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*.

**Chapter 7— Instruction Set Reference Unique to Intel® Xeon Phi™ Processors.** Describes the instruction set that is unique to Intel® Xeon Phi™ processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors.

**Appendix A — Opcode Map.** Gives an opcode map for the IA-32 instruction set.

**Appendix B — Instruction Formats and Encodings.** Gives the binary encoding of each form of each IA-32 instruction.

**Appendix C — Intel® C/C++ Compiler Intrinsic and Functional Equivalents.** Lists the Intel® C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

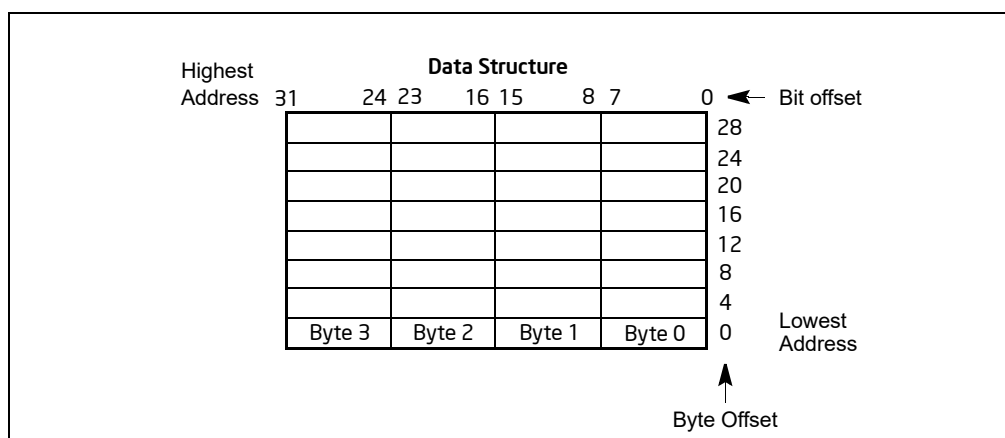


Figure 1-1. Bit and Byte Order

### 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

**NOTE**

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

**1.3.3 Instruction Operands**

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

**1.3.4 Hexadecimal and Binary Numbers**

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

**1.3.5 Segmented Addressing**

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

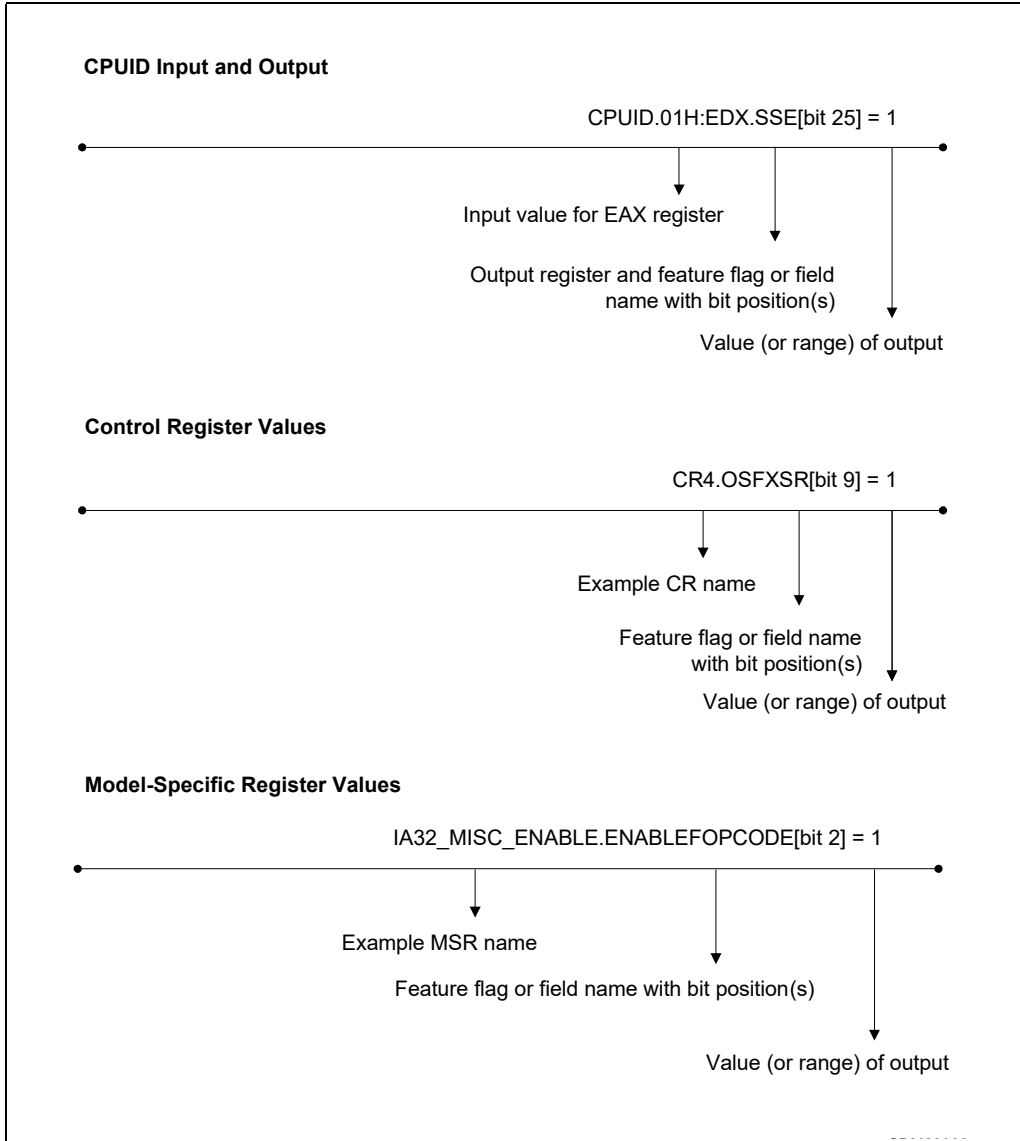


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

<https://software.intel.com/en-us/articles/intel-sdm>

See also:

- The latest security information on Intel® products:  
<https://www.intel.com/content/www/us/en/security-center/default.html>
- Software developer resources, guidance and insights for security advisories:  
<https://software.intel.com/security-software-guidance/>
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>

- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Software Development Tools:  
<https://software.intel.com/en-us/intel-sdp-home>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):  
<https://software.intel.com/en-us/articles/intel-sdm>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<https://software.intel.com/en-us/articles/intel-sdm#optimization>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Intel® Software Guard Extensions (Intel® SGX) Information  
<https://software.intel.com/en-us/isa-extensions/intel-sgx>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
<https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf>
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<https://software.intel.com/sites/default/files/22/30/25602>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference  
<https://software.intel.com/en-us/isa-extensions>

More relevant links are:

- Intel® Developer Zone:  
<https://software.intel.com/en-us>
- Developer centers:  
<http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>



## 8. Updates to Chapter 2, Volume 2A

Change bars and green text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*.

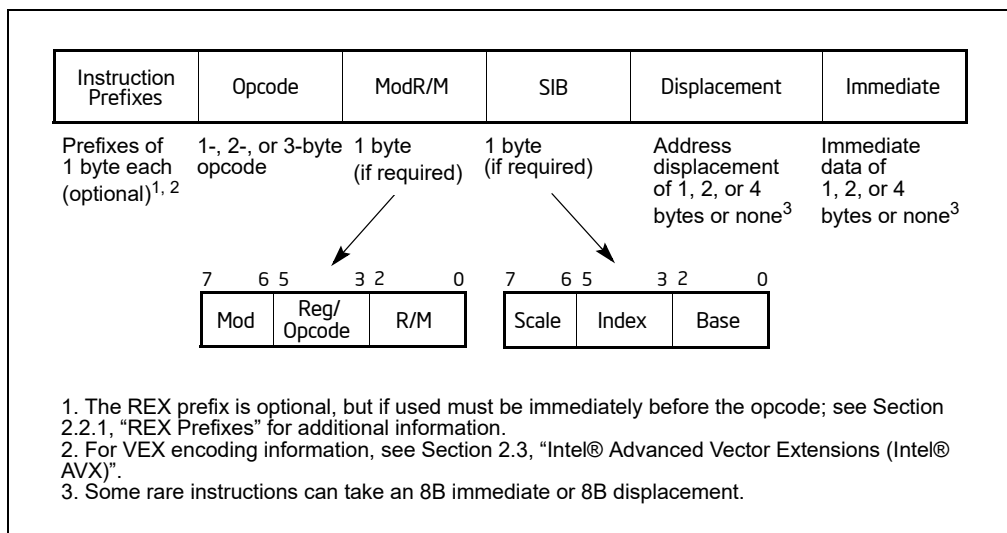
-----  
Changes to this chapter: Correction to Exception type E10NF.



This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

## 2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).



**Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format**

### 2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. F3H is also used as a mandatory prefix for POPCNT, LZCNT and ADOX instructions.

## INSTRUCTION FORMAT

- BND prefix is encoded using F2H if the following conditions are true:
  - CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.
  - BNDCFGU.EN and/or IA32\_BNDCFGS.EN is set.
  - When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Chapter 17, “Intel® MPX,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).
- Group 2
  - Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved).
    - 36H—SS segment override prefix (use with any branch instruction is reserved).
    - 3EH—DS segment override prefix (use with any branch instruction is reserved).
    - 26H—ES segment override prefix (use with any branch instruction is reserved).
    - 64H—FS segment override prefix (use with any branch instruction is reserved).
    - 65H—GS segment override prefix (use with any branch instruction is reserved).
  - Branch hints<sup>1</sup>:
    - 2EH—Branch not taken (used only with Jcc instructions).
    - 3EH—Branch taken (used only with Jcc instructions).
- Group 3
  - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
  - 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, “Instruction Set Reference, A-L,” for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H,F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (Jcc). Other use of branch hint prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

---

1. Some earlier microarchitectures used these as branch hints, but recent generations have not and they are reserved for future hint usage.

## 2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

## 2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the *r/m* field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the *reg/opcode* field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or it can be combined with the *mod* field to encode an addressing mode. Sometimes, certain combinations of the *mod* field and the *r/m* field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

## 2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

## 2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled "REG ="). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute. If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled "/digit (Opcode)"). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label "Value of ModR/M Byte (in Hexadecimal)") contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4 and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1 and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

	Mod	11	
	RM		000
/digit (Opcode);	REG =	<b>001</b>	
	C8H	11	<b>001000</b>

**Figure 2-2. Table Interpretation of ModR/M Byte (C8H)**

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP <sup>1</sup> EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 <sup>2</sup> [BX]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[BX+SI]+disp8 <sup>3</sup> [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

**NOTES:**

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =	AL AX EAX	CL CX ECX	DL DX EDX	BL BX EBX	AH SP ESP	CH BP EBP	DH SI ESI	BH DI EDI		
	MM0 XMM0	MM1 XMM1	MM2 XMM2	MM3 XMM3	MM4 XMM4	MM5 XMM5	MM6 XMM6	MM7 XMM7		
	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111		
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] [--][--] <sup>1</sup> disp32 <sup>2</sup> [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8 <sup>3</sup> [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

**NOTES:**

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte’s base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4 and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

**NOTES:**

- The [\*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [\*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits    Effective Address

- 00            [scaled index] + disp32  
01            [scaled index] + disp8 + [EBP]  
10            [scaled index] + disp32 + [EBP]

## 2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- 64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

## 2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.
- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Only one REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can be immediately preceding the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 2-3. Prefix Ordering in 64-bit Mode

### 2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

### 2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 through Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

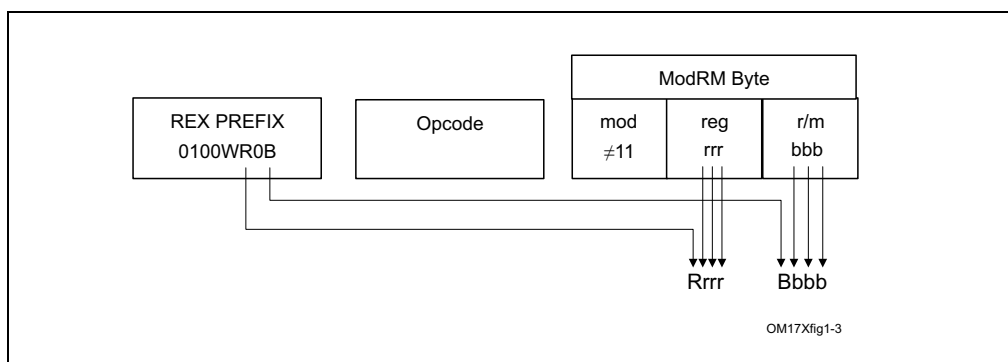
- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.



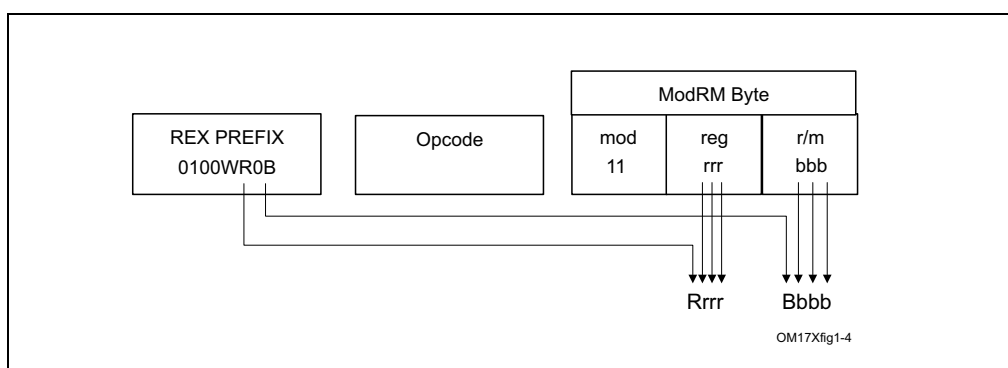
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

**Table 2-4. REX Prefix Fields [BITS: 0100WRXB]**

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D 1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field



**Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used**



**Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used**

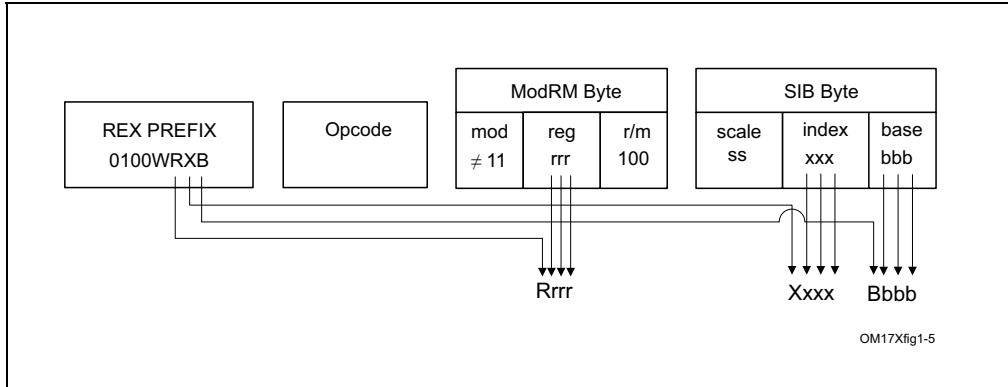


Figure 2-6. Memory Addressing With a SIB Byte

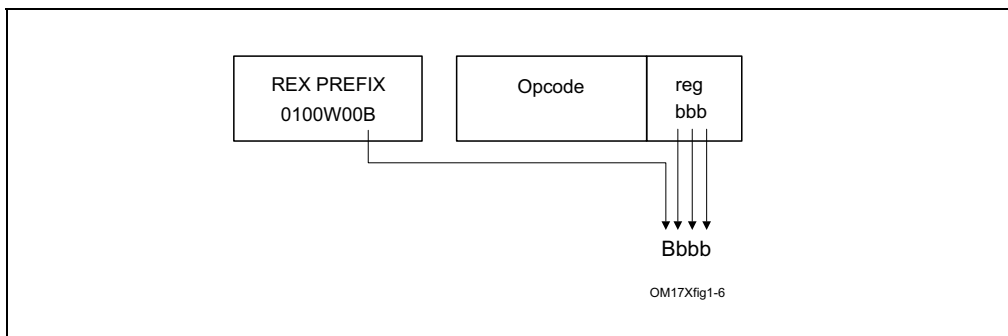


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte's reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations. Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod ≠ 11 r/m = b*100(ESP)	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0 r/m = b*101(EBP)	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

**NOTES:**

\* Don't care about value of REX.B

### 2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

### 2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

Table 2-6. Direct Memory Offset Form of MOV

Opcode	Instruction
A0	MOV AL, moffset
A1	MOV EAX, moffset
A2	MOV moffset, AL
A3	MOV moffset, EAX

### 2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8 8877665544332211 MOV RAX,1122334455667788H
```

### 2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of  $\pm 2\text{GB}$  from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

**Table 2-7. RIP-Relative Addressing**

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00	Disp32	RIP + Disp32	In 64-bit mode, if one wants to use a Disp32 without specifying a base register, one can use a SIB byte encoding (indicated by MODRM.r/m=100) as described in the next row.
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	If mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

### 2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

### 2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8-DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

## 2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

### 2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

- Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by VBLENDVPD is encoded using bits 7:4 of the immediate byte.
- Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers).
- Encoding support for instruction syntax of non-destructive source operands.
- Elimination of escape opcode byte (0FH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, “single-purpose” byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

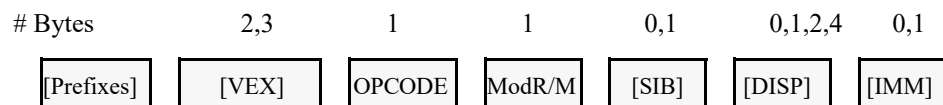


Figure 2-8. Instruction Encoding Format with VEX Prefix

### 2.3.2 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

### 2.3.3 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

### 2.3.4 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix preceding VEX will #UD.

### 2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- Non-destructive source register encoding (applicable to three and four operand syntax): This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1's complement form (inverted form), i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
- Vector length encoding: This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.
- REX prefix functionality: Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1's complement form, i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
  - Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1's complement encoding. This is represented as VEX.R.
  - Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1's complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
  - Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.

- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

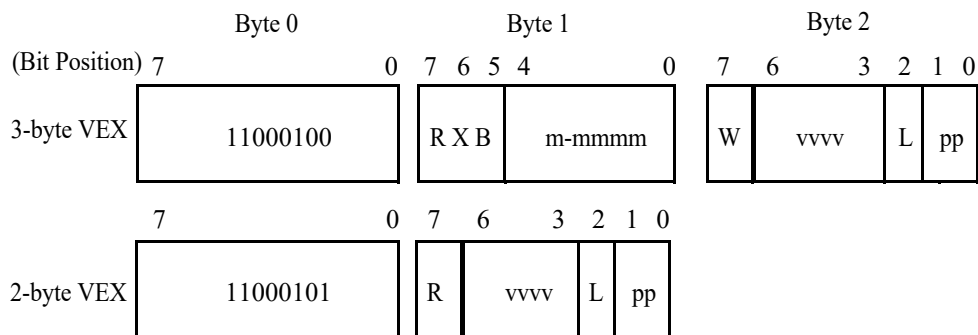
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.



R: REX.R in 1's complement (inverted) form  
 1: Same as REX.R=0 (must be 1 in 32-bit mode)  
 0: Same as REX.R=1 (64-bit mode only)

X: REX.X in 1's complement (inverted) form  
 1: Same as REX.X=0 (must be 1 in 32-bit mode)  
 0: Same as REX.X=1 (64-bit mode only)

B: REX.B in 1's complement (inverted) form  
 1: Same as REX.B=0 (Ignored in 32-bit mode).  
 0: Same as REX.B=1 (64-bit mode only)

W: opcode specific (use like REX.W, or used for opcode extension, or ignored, depending on the opcode byte)

m-mmmm:

00000: Reserved for future use (will #UD)  
 00001: implied 0F leading opcode byte  
 00010: implied 0F 38 leading opcode bytes  
 00011: implied 0F 3A leading opcode bytes  
 00100-11111: Reserved for future use (will #UD)

vvvv: a register specifier (in 1's complement form) or 1111 if unused.

L: Vector Length

0: scalar or 128-bit vector  
 1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix

00: None  
 01: 66  
 10: F3  
 11: F2

**Figure 2-9. VEX bit fields**

The following subsections describe the various fields in two or three-byte VEX prefix.

### 2.3.5.1 VEX Byte 0, bits[7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

### 2.3.5.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is present in both 2- and 3-byte VEX prefixes.

The usage of WRXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer's manual, Volume 2A.

This bit is stored in bit inverted format.

### 2.3.5.3 3-byte VEX byte 1, bit[6] - 'X'

Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

### 2.3.5.4 3-byte VEX byte 1, bit[5] - 'B'

Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

### 2.3.5.5 3-byte VEX byte 2, bit[7] - 'W'

Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.

- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes for these instructions, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

### 2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or Dest Register Specifier

In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits.

The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1's complement) form.

VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD.

In 64-bit mode all 4 bits may be used. See Table 2-8 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).



**Table 2-8. VEX.vvvv to register name mapping**

VEX.vvvv	Dest Register	General-Purpose Register (If Applicable) <sup>1</sup>	Valid in Legacy/Compatibility 32-bit modes? <sup>2</sup>
1111B	XMM0/YMM0	RAX/EAX	Valid
1110B	XMM1/YMM1	RCX/ECX	Valid
1101B	XMM2/YMM2	RDX/EDX	Valid
1100B	XMM3/YMM3	RBX/EBX	Valid
1011B	XMM4/YMM4	RSP/ESP	Valid
1010B	XMM5/YMM5	RBP/EBP	Valid
1001B	XMM6/YMM6	RSI/ESI	Valid
1000B	XMM7/YMM7	RDI/EDI	Valid
0111B	XMM8/YMM8	R8/R8D	Invalid
0110B	XMM9/YMM9	R9/R9D	Invalid
0101B	XMM10/YMM10	R10/R10D	Invalid
0100B	XMM11/YMM11	R11/R11D	Invalid
0011B	XMM12/YMM12	R12/R12D	Invalid
0010B	XMM13/YMM13	R13/R13D	Invalid
0001B	XMM14/YMM14	R14/R14D	Invalid
0000B	XMM15/YMM15	R15/R15D	Invalid

**NOTES:**

1. See Section 2.5, “VEX Encoding Support for GPR Instructions” for additional details.
2. Only the first eight General-Purpose Registers are accessible/encodable in 16/32b modes.

The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

### 2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g. VEX-encoded pack shift instructions support one source operand and one destination operand).

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax.

The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1’s complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1’s complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the “Opcode” column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

**Table 2-9. Instructions with a VEX.vvvv destination**

Opcode	Instruction mnemonic
VEX.128.66.0F 73 /7 ib	VPSLLDQ xmm1, xmm2, imm8
VEX.128.66.0F 73 /3 ib	VPSRLDQ xmm1, xmm2, imm8
VEX.128.66.0F 71 /2 ib	VPSRLW xmm1, xmm2, imm8
VEX.128.66.0F 72 /2 ib	VPSRLD xmm1, xmm2, imm8
VEX.128.66.0F 73 /2 ib	VPSRLQ xmm1, xmm2, imm8
VEX.128.66.0F 71 /4 ib	VPSRAW xmm1, xmm2, imm8

Opcode	Instruction mnemonic
VEX.128.66.0F 72 /4 ib	VPSRAD xmm1, xmm2, imm8
VEX.128.66.0F 71 /6 ib	VPSLLW xmm1, xmm2, imm8
VEX.128.66.0F 72 /6 ib	VPSLLD xmm1, xmm2, imm8
VEX.128.66.0F 73 /6 ib	VPSLLQ xmm1, xmm2, imm8

The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:

- Imm8[7:4] encodes the third source register operand.

### 2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

**Table 2-10. VEX.m-mmmm interpretation**

VEX.m-mmmm	Implied Leading Opcode Bytes
00000B	Reserved
00001B	0F
00010B	0F 38
00011B	0F 3A
00100-11111B	Reserved
(2-byte VEX)	0F

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

### 2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- “L”

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If “VEX.L = 1”, it indicates 256-bit vector operation. “VEX.L = 0” indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero’s bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

Table 2-11. VEX.L interpretation

VEX.L	Vector Length
0	128-bit (or 32/64-bit scalar)
1	256-bit

### 2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- “pp”

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.

Table 2-12. VEX.pp interpretation

pp	Implies this prefix after other prefixes but before VEX
00B	None
01B	66
10B	F3
11B	F2

## 2.3.7 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

## 2.3.8 The MODRM, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg\_field or rm\_field differs (see above).

## 2.3.9 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

## 2.3.10 AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

### 2.3.10.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255. (This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded

with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

### 2.3.11 AVX Instruction Length

The AVX instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

### 2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-3 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8D-R15D applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-3). In 32-bit mode, R8D-R15D does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e. XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte

r32 (In decimal) Base = (In binary) Base =				EAX/ R8D 0 000	ECX/ R9D 1 001	EDX/ R10D 2 010	EBX/ R11D 3 011	ESP/ R12D 4 100	EBP/ R13D <sup>1</sup> 5 101	ESI/ R14D 6 110	EDI/ R15D 7 111
Scaled Index		SS	Index	Value of SIB Byte (in Hexadecimal)							
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*1	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*2	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*4	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*8	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

**NOTES:**

1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13]+ disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

MOD	Effective Address
00b	[Scaled Vector Register] + Disp32
01b	[Scaled Vector Register] + Disp8 + [EBP/R13]
10b	[Scaled Vector Register] + Disp32 + [EBP/R13]

**2.3.12.1 64-bit Mode VSIB Memory Addressing**

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

**2.4 AVX AND SSE INSTRUCTION EXCEPTION SPECIFICATION**

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.5.1. For example, ADDPS contains the entry:

“See Exceptions Type 2”

## INSTRUCTION FORMAT

In this entry, "Type2" can be looked up in Table 2-14.

The instruction's corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

### NOTE

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 21.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

**Table 2-14. Exception Class Description**

Exception Class	Instruction set	Mem arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	None
Type 2	AVX, Legacy SSE	16/32 byte not explicitly aligned	Yes
Type 3	AVX, Legacy SSE	< 16 byte	Yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	No
Type 5	AVX, Legacy SSE	< 16 byte	No
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, Legacy SSE	None	None
Type 8	AVX	None	None
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	Yes
Type 12	AVX2 Gathers	Not explicitly aligned, no AC#	No

See Table 2-15 for lists of instructions in each exception class.



## INSTRUCTION FORMAT

- (\*\*) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no alignment checks are performed.
- (\*\*\*) - PCMPSTRM, PCMPSTRM, PCMPSTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-17 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

**Table 2-16. #UD Exception and VEX.W=1 Encoding**

Exception Class	#UD If VEX.W = 1 in all modes	#UD If VEX.W = 1 in non-64-bit modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLEND, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	
Type 12		



Table 2-17. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 1		VMOVNTDQA	
Type 2		VDPPD	VDPPD
Type 3			
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRW, VPMULHW/LW, VPMULLD, VPMULLDQ, VPMULDQ, VPOR, VPSADBW, VPSHUF/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/W/D/DQ, VPUNPCKHQDQ, VPUNPCKLBW/W/D/DQ, VPUNPCKLQDQ, VPXOR	VPCMP(E/I)STRI/M, PHMINPOSUW
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR	Same as column 3
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,		
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMASKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ	VMOVLHPS, VMOVHLPS
Type 8			
Type 11			
Type 12			

## 2.4.1 Exceptions Type 1 (Aligned Memory Reference)

Table 2-18. Type 1 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	VEX.256: Memory operand is not 32-byte aligned. VEX.128: Memory operand is not 16-byte aligned.
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

## 2.4.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

Table 2-19. Type 2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CRO.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.4.3 Exceptions Type 3 (<16 Byte Memory Argument)

Table 2-20. Type 3 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.4.4 Exceptions Type 4 (>=16 Byte Mem Arg, No Alignment, No Floating-point Exceptions)

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CRO.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned. <sup>1</sup>
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

### NOTES:

1. LDDQU, MOVUPD, MOVUPS, PCMPSTRI, PCMPSTRM, PCMPISTRI, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

## 2.4.5 Exceptions Type 5 (<16 Byte Mem Arg and No FP Exceptions)

**Table 2-22. Type 5 Class Exception Conditions**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CRO.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.4.6 Exceptions Type 6 (VEX-Encoded Instructions without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

**Table 2-23. Type 6 Class Exception Conditions**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.4.7 Exceptions Type 7 (No FP Exceptions, No Memory Arg)

Table 2-24. Type 7 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.

## 2.4.8 Exceptions Type 8 (AVX and No Memory Argument)

Table 2-25. Type 8 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual-8086 mode.
			X	X	If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv ≠ 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.



## 2.4.9 Exceptions Type 11 (VEX-only, Mem Arg, No AC, Floating-point Exceptions)

Table 2-26. Type 11 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.4.10 Exceptions Type 12 (VEX-only, VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-27. Type 12 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm ≠ '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X		For an illegal address in the SS segment.
Stack, #SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
General Protection, #GP(0)				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

## 2.5 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.

Any VEX-encoded GPR instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

Any VEX-encoded GPR instruction with a REX prefix proceeding VEX will #UD.

VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

## 2.5.1 Exceptions Type 13 (VEX-Encoded GPR Instructions)

The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GPR instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

**Table 2-28. VEX-Encoded GPR Instructions**

Exception Class	Instruction
Type 13	ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

(\*) - Additional exception restrictions are present - see the Instruction description for details.

**Table 2-29. Type 13 Class Exception Conditions**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
	X	X	X	X	If VEX.L = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, #SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.6 INTEL® AVX-512 ENCODING

The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.

## INSTRUCTION FORMAT

The significant feature differences between EVEX and VEX are summarized below.

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve the encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g., packed instruction with "load+op" semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality).

### 2.6.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10. Note that the values contained within brackets are optional.

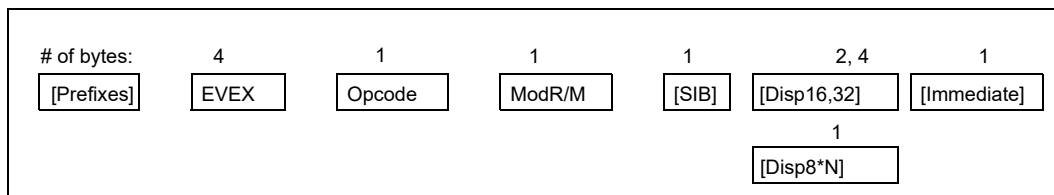


Figure 2-10. AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).

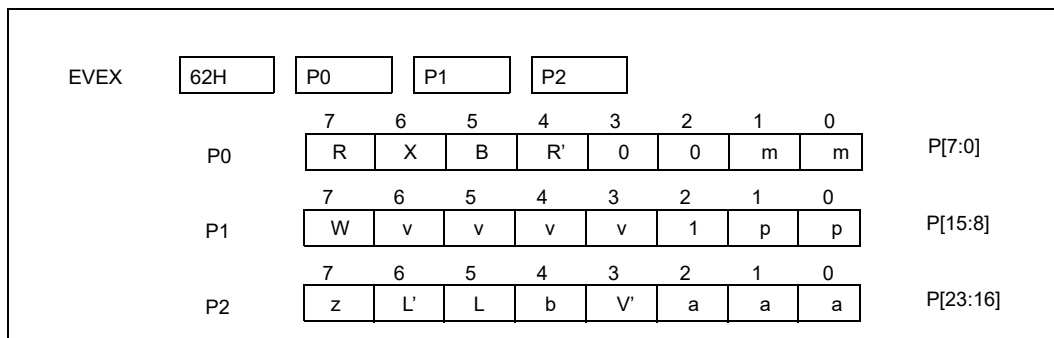


Figure 2-11. Bit Field Layout of the EVEX Prefix<sup>1</sup>

#### NOTES:

1. See Table 2-30 for additional details on bit fields.

Table 2-30. EVEX Prefix Bit Field Functional Grouping

Notation	Bit field Group	Position	Comment
--	Reserved	P[3 : 2]	Must be 0.
--	Fixed Value	P[10]	Must be 1.
EVEX.mm	Compressed legacy escape	P[1 : 0]	Identical to low two bits of VEX.mmmmm.
EVEX.pp	Compressed legacy prefix	P[9 : 8]	Identical to VEX.pp.
EVEX.RXB	Next-8 register specifier modifier	P[7 : 5]	Combine with ModR/M.reg, ModR/M.rm (base, index/vidx). This field is encoded in bit inverted format.
EVEX.R'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg. This bit is stored in inverted format.
EVEX.X	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent.
EVEX.vvvv	VVVV register specifier	P[14 : 11]	Same as VEX.vvvv. This field is encoded in bit inverted format.
EVEX.V'	High-16 VVVV/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present. This bit is stored in inverted format.
EVEX.aaa	Embedded opmask register specifier	P[18 : 16]	
EVEX.W	Osize promotion/Opcode extension	P[15]	
EVEX.z	Zeroing/Merging	P[23]	
EVEX.b	Broadcast/RC/SAE Context	P[20]	
EVEX.L'L	Vector length/RC	P[22 : 21]	

The bit fields in P[23:0] are divided into the following functional groups (Table 2-30 provides a tabular summary):

- Reserved bits: P[3:2] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
  - Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).

- Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
- For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.
- Vector length/rounding control specifier: P[22:21] can serve one of three options.
  - Vector length information for packed vector instructions.
  - Ignored for instructions operating on vector register content as a single data element.
  - Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

### 2.6.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-31. Opmask register encoding is described in Section 2.6.3.

**Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits**

	4 <sup>1</sup>	3	[2:0]	Reg. Type	Common Usages
<b>REG</b>	EVEX.R'	REX.R	modrm.reg	GPR, Vector	Destination or Source
<b>VVVV</b>	EVEX.V'	EVEX.vvvv		GPR, Vector	2nd Source or Destination
<b>RM</b>	EVEX.X	EVEX.B	modrm.r/m	GPR, Vector	1st Source or Destination
<b>BASE</b>	0	EVEX.B	modrm.r/m	GPR	memory addressing
<b>INDEX</b>	0	EVEX.X	sib.index	GPR	memory addressing
<b>VIDX</b>	EVEX.V'	EVEX.X	sib.index	Vector	VSIB memory addressing

**NOTES:**

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-32.

**Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode**

	[2:0]	Reg. Type	Common Usages
<b>REG</b>	modrm.reg	GPR, Vector	Destination or Source
<b>VVVV</b>	EVEX.vvv	GPR, Vector	2nd Source or Destination
<b>RM</b>	modrm.r/m	GPR, Vector	1st Source or Destination
<b>BASE</b>	modrm.r/m	GPR	Memory Addressing
<b>INDEX</b>	sib.index	GPR	Memory Addressing
<b>VIDX</b>	sib.index	Vector	VSIB Memory Addressing

### 2.6.3 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using

standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.

- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.6.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

**Table 2-33. Opmask Register Specifier Encoding**

	[2:0]	Register Access	Common Usages
REG	modrm.reg	k0-k7	Source
VVVV	VEX.vvvv	k0-k7	2nd Source
RM	modrm.r/m	k0-7	1st Source
{k1}	EVEX.aaa	k0 <sup>1</sup> -k7	Opmask

**NOTES:**

1. Instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

## 2.6.4 Masking Support in EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided into the following groups:

- Instructions which support “zeroing-masking”.
  - Also allow merging-masking.
- Instructions which require aaa = 000.
  - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking.
  - Require EVEX.z to be set to 0.
  - This group is mostly composed of instructions that write to memory.
- Instructions which require aaa <> 000 do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

## 2.6.5 Compressed Displacement (disp8\*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-34 and Table 2-35 below,

where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-34 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.6.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less than dword do not support broadcasting using EVEX.b. These are listed in Table 2-35. Table 2-35 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-35. Instruction classified in Table 2-35 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8\*N rules still apply when using 16b addressing.

**Table 2-34. Compressed Displacement (DISP8\*N) Affected by Embedded Broadcast**

TupleType	EVEX.b	InputSize	EVEX.W	Broadcast	N (VL=128)	N (VL=256)	N (VL= 512)	Comment
Full	0	32bit	0	none	16	32	64	Load+Op (Full Vector Dword/Qword)
	1	32bit	0	{1tox}	4	4	4	
	0	64bit	1	none	16	32	64	
	1	64bit	1	{1tox}	8	8	8	
Half	0	32bit	0	none	8	16	32	Load+Op (Half Vector)
	1	32bit	0	{1tox}	4	4	4	

**Table 2-35. EVEX DISP8\*N for Instructions Not Affected by Embedded Broadcast**

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Mem	N/A	N/A	16	32	64	Load/store or subDword full vector
Tuple1 Scalar	8bit	N/A	1	1	1	1 Tuple
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed	32bit	N/A	4	4	4	1 Tuple, memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple2	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8	32bit	0	NA	NA	32	Broadcast (8 elements)
Half Mem	N/A	N/A	8	16	32	SubQword Conversion
Quarter Mem	N/A	N/A	4	8	16	SubDword Conversion
Eighth Mem	N/A	N/A	2	4	8	SubWord Conversion
Mem128	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP	N/A	N/A	8	32	64	VMOVDDUP



## 2.6.6 EVEX Encoding of Broadcast/Rounding/SAE Support

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with “load+op” semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- “Suppress All exceptions” (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

## 2.6.7 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single-precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

## 2.6.8 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L'L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

## 2.6.9 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

## 2.6.10 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L'L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L'L bits. When EVEX is used to encode scalar instructions, L'L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 2-36.

**Table 2-36. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions**

Position	P2[4]	P2[6:5]	P2[6:5]
Broadcast/Rounding/SAE Context	EVEX.b	EVEX.L'L	EVEX.RC
Reg-reg, FP Instructions w/ rounding semantic or SAE	Enable static rounding control (SAE implied)	Vector length Implied (512 bit or scalar)	00b: SAE + RNE 01b: SAE + RD 10b: SAE + RU 11b: SAE + RZ
Load+op Instructions w/ memory source	Broadcast Control	00b: 128-bit 01b: 256-bit 10b: 512-bit 11b: Reserved (#UD)	NA
Other Instructions (Explicit Load/Store/Broadcast/Gather/Scatter)	Must be 0 (otherwise #UD)		NA

## 2.6.11 #UD Equations for EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

### 2.6.11.1 State Dependent #UD

In general, attempts to execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-37 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-37 will cause #UD.

**Table 2-37. OS XSAVE Enabling Requirements of Instruction Categories**

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 256-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	111xxx11b

### 2.6.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-38.

**Table 2-38. Opcode Independent, State Dependent EVEX Bit Fields**

Position	Notation	64-bit #UD	Non-64-bit #UD
P[3 : 2]	--	if > 0	if > 0
P[10]	--	if 0	if 0
P[1: 0]	EVEX.mm	if 00b	if 00b
P[7 : 6]	EVEX.RX	None (valid)	None (BOUND if EVEX.RX != 11b)

### 2.6.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-39 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

**Table 2-39. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields**

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.R	P[7]	ModRM.reg encodes k-reg	If EVEX.R = 0	None (BOUND if EVEX.RX != 11b)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes all other registers	None (valid)	
EVEX.X	P[6]	ModRM.r/m encodes ZMM/YMM/XMM	None (valid)	
		ModRM.r/m encodes k-reg or GPR	None (ignored)	
		ModRM.r/m without SIB/VSIB	None (ignored)	
		ModRM.r/m with SIB/VSIB	None (valid)	
EVEX.B	P[5]	ModRM.r/m encodes k-reg	None (ignored)	None (ignored)
		ModRM.r/m encodes other registers	None (valid)	
		ModRM.r/m base present	None (valid)	
		ModRM.r/m base not present	None (ignored)	
EVEX.R'	P[4]	ModRM.reg encodes k-reg or GPR	If 0	None (ignored)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes ZMM/YMM/XMM	None (valid)	
EVEX.vvvv	P[14 : 11]	vvvv encodes ZMM/YMM/XMM	None (valid)	None (valid) P[14] ignored
		Otherwise	If != 1111b	If != 1111b
EVEX.V'	P[19]	Encodes ZMM/YMM/XMM	None (valid)	If 0
		Otherwise	If 0	If 0

Table 2-40 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

**Table 2-40. #UD Conditions of Opmask Related Encoding Field**

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18 : 16]	Instructions do not use opmask for conditional processing <sup>1</sup> .	If aaa != 000b	If aaa != 000b
		Opmask used as conditional processing mask and updated at completion <sup>2</sup> .	If aaa = 000b	If aaa = 000b;
		Opmask used as conditional processing.	None (valid <sup>3</sup> )	None (valid <sup>1</sup> )
EVEX.z	P[23]	Vector instruction using opmask as source or destination <sup>4</sup> .	If EVEX.z != 0	If EVEX.z != 0
		Store instructions or gather/scatter instructions.	If EVEX.z != 0	If EVEX.z != 0
		Instruction supporting conditional processing mask with EVEX.aaa = 000b.	If EVEX.z != 0	If EVEX.z != 0
VEX.vvvv	Varies	K-regs are instruction operands not mask control.	If vvvv = 0xxx	None

#### NOTES:

1. E.g., VPBROADCASTMxxx, VPMOVM2x, VPMOVx2M.

2. E.g., Gather/Scatter family.

3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in KO.

4. E.g., VFPClassPD/PS, VCMPB/D/Q/W family, VPMOVM2x, VPMOVx2M.

Table 2-41 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

**Table 2-41. #UD Conditions Dependent on EVEX.b Context**

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	Reg-reg, FP instructions with rounding semantic.	None (valid <sup>1</sup> )	None (valid <sup>1</sup> )
		Other reg-reg, FP instructions that can cause #XM.	None (valid <sup>2</sup> )	None (valid <sup>2</sup> )
		Other reg-mem instructions in Table 2-34.	None (valid <sup>3</sup> )	None (valid <sup>3</sup> )
		Other instruction classes <sup>4</sup> in Table 2-35.	If EVEX.b > 0	If EVEX.b > 0

**NOTES:**

1. L'L specifies rounding control, see Table 2-36, supports {er} syntax.
2. L'L specifies vector length, see Table 2-36, supports {sae} syntax.
3. L'L specifies vector length, see Table 2-36, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

### 2.6.12 Device Not Available

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

### 2.6.13 Scalar Instructions

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

## 2.7 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

**Table 2-42. EVEX-Encoded Instruction Exception Class Summary**

Exception Class	Instruction set	Mem arg	(#XM)
Type E1	Vector Moves/Load/Stores	Explicitly aligned, w/ fault suppression	None
Type E1NF	Vector Non-temporal Stores	Explicitly aligned, no fault suppression	None
Type E2	FP Vector Load+op	Support fault suppression	Yes
Type E2NF	FP Vector Load+op	No fault suppression	Yes
Type E3	FP Scalar/Partial Vector, Load+Op	Support fault suppression	Yes
Type E3NF	FP Scalar/Partial Vector, Load+Op	No fault suppression	Yes
Type E4	Integer Vector Load+op	Support fault suppression	No
Type E4NF	Integer Vector Load+op	No fault suppression	No
Type E5	Legacy-like Promotion	Varies, Support fault suppression	No

Table 2-42. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E5NF	Legacy-like Promotion	Varies, No fault suppression	No
Type E6	Post AVX Promotion	Varies, w/ fault suppression	No
Type E6NF	Post AVX Promotion	Varies, no fault suppression	No
Type E7NM	Register-to-register op	None	None
Type E9NF	Miscellaneous 128-bit	Vector-length Specific, no fault suppression	None
Type E10	Non-XF Scalar	Vector Length ignored, w/ fault suppression	None
Type E10NF	Non-XF Scalar	Vector Length ignored, no fault suppression	None
Type E11	VCVTPH2PS, VCVTPS2PH	Half Vector Length, w/ fault suppression	Yes
Type E12	Gather and Scatter Family	VSIB addressing, w/ fault suppression	None
Type E12NP	Gather and Scatter Prefetch Family	VSIB addressing, w/o page fault	None

Table 2-43 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-43. EVEX Instructions in Each Exception Class

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPs, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2DQ, VCVTPS2UDQs, VCVTQ2PD, VCVTQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VDIVPD, VDIVPS, VEXP2PD, VEXP2PS, VFIXUPIMMPD, VFIXUPIMMPS, VFMADDxxxPD, VFMADDxxxPS, VFMADDSUBxxxPD, VFMADDSUBxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS, VSCALEFPD, VSCALEFPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS
Type E3	VADDSd, VADDSs, VCMPSD, VCMPSs, VCVTPS2QQ, VCVTPS2UQQ, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VCVTTPS2QQ, VCVTTPS2UQQ, VDIVSD, VDIVSS, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VMAXSD, VMAXSS, VMINSD, VMINSS, VMULSD, VMULSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS, VSQRTSD, VSQRTSS, VSUBSD, VSUBSS
Type E3NF	VCOMISD, VCOMISS, VCVTSD2SI, VCVTSD2USI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTSS2USI, VCVTTSD2SI, VCVTTSD2USI, VCVTTSS2SI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SS, VUCOMISD, VUCOMISS
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VBLENDMPD, VBLENDMPS, VFPCLASSPD, VFPCLASSPS, VORPD, VORPS, VPABSD, VPABSQ, VPADD, VPADDQ, VPAND, VPANDQ, VPANDND, VPANDNQ, VPBLENDMB, VPBLENDMD, VPBLENDMQ, VPBLENDMw, VPCMPD, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPMADD52LUQ, VPMADD52HUQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPROLD, VPROLQ, VPROLVD, VPROLVQ, VPRORD, VPRORQ, VPRORVD, VPRORVQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRAVw, VPSRAVD, VPSRAVw, VPSRAVQ, VPSRLD, VPSRLQ) <sup>1</sup> , VPSUBD, VPSUBQ, VPSUBUSB, VPSUBUSw, VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VPXORD, VPXORQ, VPSLLVD, VPSLLVQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VXORPD, VXORPS

Table 2-43. EVEX Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
E4.nb <sup>2</sup>	VCOMPRESSPD, VCOMPRESSPS, VEXPANDPD, VEXPANDPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VMOVUPD, VMOVUPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPB, VPCMPQB, VPCMPQW, VPCMPGTB, VPCMPGTW, VPCMPW, VPCMPUB, VPCMPUW, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW, VPSLLVW, VPSLLW, VPSRAW, VPSRLVW, VPSRLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW
Type E4NF	VALIGND, VALIGNQ, VPACKSSDW, VPACKUSDW, VPCONFLICTD, VPCONFLICTQ, VPERMD, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMPD, VPERMPS, VPERMQ, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VPMULTISHIFTQB, VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2, VSHUFFD, VSHUFFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS
E4NF.nb <sup>2</sup>	VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP, VPSADBW, VPSHUFB, VPSHUFBW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ) <sup>3</sup> , VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD, VPERMW, VPERMI2W, VPERMT2W
Type E5	PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW, PMOVZXBW, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ, VCVTDQ2PD, VCVTUDQ2PD, VPMOVSXxx, VPMOVZXxx
Type E5NF	VMOVDDUP
Type E6	VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VBROADCASTSD, VBROADCASTSS, VFPCCLASSSD, VFPCCLASSSS, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSD, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E6NF	VEXTRACTF32X4, VEXTRACTF32X8, VEXTRACTF64X2, VEXTRACTF64X4, VEXTRACTI32X4, VEXTRACTI32X8, VEXTRACTI64X2, VEXTRACTI64X4, VINSERTF32X4, VINSERTF32X8, VINSERTF64X2, VINSERTF64X4, VINSERTI32X4, VINSERTI32X8, VINSERTI64X2, VINSERTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D
Type E7NM.128 <sup>4</sup>	VMOVHLP, VMOVLHPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW) <sup>5</sup> , VPMOVBM2M, VPMOVD2M, VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVQ2M, VPMOVW2M
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ
Type E10	VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS
Type E10NF	(VCVTISI2SD, VCVTUSI2SD) <sup>6</sup>
Type E11	VCVTPH2PS, VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS
Type E12NP	VGATHERPFODPD, VGATHERPFODPS, VGATHERPFOQPD, VGATHERPFOQPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPFODPD, VSCATTERPFODPS, VSCATTERPFOQPD, VSCATTERPFOQPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

**NOTES:**

1. Operand encoding Full tupletype with immediate.
2. Embedded broadcast is not supported with the “.nb” suffix.
3. Operand encoding Mem128 tupletype.

4. #UD raised if EVEX.L'L != 00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.

## 2.7.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

**Table 2-44. Type E1 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

**Table 2-45. Type E1NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.



## 2.7.2 Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2.

**Table 2-46. Type E2 Class Exception Conditions**

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.7.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

**Table 2-47. Type E3 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

**Table 2-48. Type E3NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.7.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

**Table 2-49. Type E4 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0 and in E4.nb subclass (see E4.nb entries in Table 2-43).</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

**Table 2-50. Type E4NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-43).</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

## 2.7.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

**Table 2-51. Type E5 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

Table 2-52. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.7.6 Exceptions Type E6 and E6NF

Table 2-53. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
			X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.



EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

**Table 2-54. Type E6NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.7.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

**Table 2-55. Type E7NM Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ Instruction specific EVEX.L'L restriction not met.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

## 2.7.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

**Table 2-56. Type E9 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 00b (VL=128).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

**Table 2-57. Type E9NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 00b (VL=128).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.7.9 Exceptions Type E10 and E10NF

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and support memory fault suppression follow exception class E10.

**Table 2-58. Type E10 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and do not support memory fault suppression follow exception class E10NF.

**Table 2-59. Type E10NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

## 2.7.10 Exception Type E11 (EVEX-only, Mem Arg, No AC, Floating-point Exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11.

**Table 2-60. Type E11 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.

## 2.7.11 Exception Type E12 and E12NP (VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-61. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> <li>If vvvv != 1111b.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
X	X	X	X	If index = destination register (gather operation).	
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.



EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

**Table 2-62. Type E12NP Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

## 2.8 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

The exception behavior of VEX-encoded opmask instructions are listed below.

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

**Table 2-63. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> </ul>
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If ModRM:[7:6] != 11b.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

Exception conditions of Opmask instructions that address memory are listed as Type K21.

**Table 2-64. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> </ul>
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, #SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.



## 9. Updates to Chapter 3, Volume 2A

Change bars and green text show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*.

---

Changes to this chapter:

- ADD: Typo correction in description.
- CALL: Update to pseudocode and exceptions.
- CLRSSBSY: Corrected typos in pseudocode.
- CPUID: Added reserved leaf 21H under "Unimplemented CPUID Leaf Functions" and corrected typos in leaf 12H (Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)).
- CVTPI2PS: Updated destination operand to be r, w.
- CVTSD2SS: Updated destination operand for SSE version of instruction to be r, w. Added (r) where missing.
- CVTSS2SD: Updated destination operand for SSE version of instruction to be r, w. Added (r) where missing.
- INT n/INTO/INT3/INT1: Update to pseudocode and exceptions.
- IRET: Update to pseudocode.
- JMP: Updated operand encoding.

Additionally, there are three updates made to various instructions that are categorized as either typo corrections or cross-reference updates. These instruction pages are not included in this change document as their impact is evident.

- Updated instructions that were missing "(r)" after EVEX.vvvv or VEX.vvvvv: ADDPD, ADDPS, ADDSD, ADDSS, ANDPD, ANDPS, ANDNPD, ANDNPS, CMPPD, CMPPS, CMPSD, CMPSS, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, DIVPD, DIVPS, DIVSD, DIVSS, and INSERTPS.
- Updated instructions incorrectly using "EVEX.LIG" to use "EVEX.LLIG": ADDSD, ADDSS, CMPSD, CMPSS, COMISD, COMISS, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTSD2SI, CVTTSS2SI, DIVSD, and DIVSS.
- Updated all instructions that list a specific exceptions type to use a cross-reference in order to allow users to jump directly to the applicable exception section for the instruction.

## ADD—Add

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	I	Valid	Valid	Add <i>imm8</i> to AL.
05 <i>iw</i>	ADD AX, <i>imm16</i>	I	Valid	Valid	Add <i>imm16</i> to AX.
05 <i>id</i>	ADD EAX, <i>imm32</i>	I	Valid	Valid	Add <i>imm32</i> to EAX.
REX.W + 05 <i>id</i>	ADD RAX, <i>imm32</i>	I	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to RAX.
80 /0 <i>ib</i>	ADD <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to <i>r/m8</i> .
REX + 80 /0 <i>ib</i>	ADD <i>r/m8</i> <sup>*</sup> , <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to <i>r/m8</i> .
81 /0 <i>iw</i>	ADD <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to <i>r/m16</i> .
81 /0 <i>id</i>	ADD <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to <i>r/m32</i> .
REX.W + 81 /0 <i>id</i>	ADD <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to <i>r/m64</i> .
83 /0 <i>ib</i>	ADD <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to <i>r/m16</i> .
83 /0 <i>ib</i>	ADD <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to <i>r/m32</i> .
REX.W + 83 /0 <i>ib</i>	ADD <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to <i>r/m64</i> .
00 /r	ADD <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Add <i>r8</i> to <i>r/m8</i> .
REX + 00 /r	ADD <i>r/m8</i> <sup>*</sup> , <i>r8</i> <sup>*</sup>	MR	Valid	N.E.	Add <i>r8</i> to <i>r/m8</i> .
01 /r	ADD <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Add <i>r16</i> to <i>r/m16</i> .
01 /r	ADD <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Add <i>r32</i> to <i>r/m32</i> .
REX.W + 01 /r	ADD <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Add <i>r64</i> to <i>r/m64</i> .
02 /r	ADD <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Add <i>r/m8</i> to <i>r8</i> .
REX + 02 /r	ADD <i>r8</i> <sup>*</sup> , <i>r/m8</i> <sup>*</sup>	RM	Valid	N.E.	Add <i>r/m8</i> to <i>r8</i> .
03 /r	ADD <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Add <i>r/m16</i> to <i>r16</i> .
03 /r	ADD <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Add <i>r/m32</i> to <i>r32</i> .
REX.W + 03 /r	ADD <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Add <i>r/m64</i> to <i>r64</i> .

### NOTES:

\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA
MR	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	ModRM:reg ( <i>r</i> )	NA	NA
MI	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	<i>imm8/16/32</i>	NA	NA
I	AL/AX/EAX/RAX	<i>imm8/16/32</i>	NA	NA

### Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the **OF** and **CF** flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The **SF** flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

DEST := DEST + SRC;

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## CALL—Call Procedure

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
E8 <i>cw</i>	CALL <i>rel16</i>	D	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 <i>cd</i>	CALL <i>rel32</i>	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF <i>/2</i>	CALL <i>r/m16</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m16</i> .
FF <i>/2</i>	CALL <i>r/m32</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m32</i> .
FF <i>/2</i>	CALL <i>r/m64</i>	M	Valid	N.E.	Call near, absolute indirect, address given in <i>r/m64</i> .
9A <i>cd</i>	CALL <i>ptr16:16</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
9A <i>cp</i>	CALL <i>ptr16:32</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
FF <i>/3</i>	CALL <i>m16:16</i>	M	Valid	Valid	Call far, absolute indirect address given in <i>m16:16</i> . In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF <i>/3</i>	CALL <i>m16:32</i>	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W FF <i>/3</i>	CALL <i>m16:64</i>	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m ( <i>r</i> )	NA	NA	NA

## Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, “Task Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for information on performing task switches with the CALL instruction.



**Near Call.** When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32*, or *r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

**Far Calls in Real-Address or Virtual-8086 Mode.** When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Calls in Protected Mode.** When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedure's stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute a task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

**Mixing 16-Bit and 32-Bit Calls.** When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 20, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

**Far Calls in Compatibility Mode.** When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This is specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

**Near(Far) Calls in 64-bit Mode.** When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for CET details.

**Instruction ordering.** Instructions following a far call may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far call have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Certain situations may lead to the next sequential instruction after a near indirect CALL being speculatively executed. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an LFENCE instruction opcode can be placed after the near indirect CALL in order to block speculative execution.

## Operation

IF near call

    THEN IF near relative call

        THEN

            IF OperandSize = 64

                THEN

                    tempDEST := SignExtend(DEST); (\* DEST is rel32 \*)

                    tempRIP := RIP + tempDEST;

                    IF stack not large enough for a 8-byte return address

                        THEN #SS(0); FI;

                    Push(RIP);

                    IF ShadowStackEnabled(CPL) AND DEST != 0

                        ShadowStackPush8B(RIP);

                    FI;

                    RIP := tempRIP;

        FI;

        IF OperandSize = 32

            THEN

                tempEIP := EIP + DEST; (\* DEST is rel32 \*)

                IF tempEIP is not within code segment limit THEN #GP(0); FI;

                IF stack not large enough for a 4-byte return address

                    THEN #SS(0); FI;

                Push(EIP);

                IF ShadowStackEnabled(CPL) AND DEST != 0

                    ShadowStackPush4B(EIP);

                FI;

                EIP := tempEIP;

        FI;

        IF OperandSize = 16

            THEN

                tempEIP := (EIP + DEST) AND 0000FFFFH; (\* DEST is rel16 \*)

                IF tempEIP is not within code segment limit THEN #GP(0); FI;

                IF stack not large enough for a 2-byte return address

                    THEN #SS(0); FI;

                Push(IP);

                IF ShadowStackEnabled(CPL) AND DEST != 0

                    (\* IP is zero extended and pushed as a 32 bit value on shadow stack \*)

                    ShadowStackPush4B(IP);

                FI;

                EIP := tempEIP;

        FI;

    ELSE (\* Near absolute call \*)

        IF OperandSize = 64

```

THEN
    tempRIP := DEST; (* DEST is r/m64 *)
    IF stack not large enough for a 8-byte return address
        THEN #SS(0); FI;
    Push(RIP);
    IF ShadowStackEnabled(CPL)
        ShadowStackPush8B(RIP);
    FI;
    RIP := tempRIP;
FI;
IF OperandSize = 32
    THEN
        tempEIP := DEST; (* DEST is r/m32 *)
        IF tempEIP is not within code segment limit THEN #GP(0); FI;
        IF stack not large enough for a 4-byte return address
            THEN #SS(0); FI;
        Push(EIP);
        IF ShadowStackEnabled(CPL)
            ShadowStackPush4B(EIP);
        FI;
        EIP := tempEIP;
    FI;
IF OperandSize = 16
    THEN
        tempEIP := DEST AND 0000FFFFH; (* DEST is r/m16 *)
        IF tempEIP is not within code segment limit THEN #GP(0); FI;
        IF stack not large enough for a 2-byte return address
            THEN #SS(0); FI;
        Push(IP);
        IF ShadowStackEnabled(CPL)
            (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
            ShadowStackPush4B(IP);
        FI;
        EIP := tempEIP;
    FI;
FI;rel/abs
IF (Call near indirect, absolute indirect)
    IF EndbranchEnabledAndNotSuppressed(CPL)
        IF CPL = 3
            THEN
                IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
            ELSE
                IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
            FI;
        FI;
    FI;
FI; near

```

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (\* Real-address or virtual-8086 mode \*)

```

THEN
  IF OperandSize = 32
    THEN
      IF stack not large enough for a 6-byte return address
        THEN #SS(0); FI;
      IF DEST[31:16] is not zero THEN #GP(0); FI;
      Push(CS); (* Padded with 16 high-order bits *)
      Push(EIP);
      CS := DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
      EIP := DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
    ELSE (* OperandSize = 16 *)
      IF stack not large enough for a 4-byte return address
        THEN #SS(0); FI;
      Push(CS);
      Push(IP);
      CS := DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
      EIP := DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
    FI;
  FI;
FI;

```

IF far call and (PE = 1 and VM = 0) (\* Protected mode or IA-32e Mode, not virtual-8086 mode\*)

```

THEN
  IF segment selector in target operand NULL
    THEN #GP(0); FI;
  IF segment selector index not within descriptor table limits
    THEN #GP(new code segment selector); FI;
  Read type and access rights of selected segment descriptor;
  IF IA32_EFER.LMA = 0
    THEN
      IF segment type is not a conforming or nonconforming code segment, call
        gate, task gate, or TSS
        THEN #GP(segment selector); FI;
    ELSE
      IF segment type is not a conforming or nonconforming code segment or
        64-bit call gate,
        THEN #GP(segment selector); FI;
    FI;
  Depending on type and access rights:
  GO TO CONFORMING-CODE-SEGMENT;
  GO TO NONCONFORMING-CODE-SEGMENT;
  GO TO CALL-GATE;
  GO TO TASK-GATE;
  GO TO TASK-STATE-SEGMENT;
FI;

```

CONFORMING-CODE-SEGMENT:

```

IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
  THEN GP(new code segment selector); FI;
IF DPL > CPL
  THEN #GP(new code segment selector); FI;
IF segment not present
  THEN #NP(new code segment selector); FI;
IF stack not large enough for return address

```

```

    THEN #SS(0); FI;
tempEIP := DEST(Offset);
IF target mode = Compatibility mode
    THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
    THEN
        tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF OperandSize = 32
        THEN
            tempPushLIP = CSBASE + EIP;
        ELSE
            IF OperandSize = 16
                THEN
                    tempPushLIP = CSBASE + IP;
                ELSE (* OperandSize = 64 *)
                    tempPushLIP = RIP;
            FI;
        FI;
    tempPushCS = CS;
FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS := DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) := CPL;
        EIP := tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS := DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) := CPL;
                EIP := tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
                Push(RIP);
                CS := DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) := CPL;
                RIP := tempEIP;
            FI;
        FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)

```

```

        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
FI;
(* align to 8 byte boundary if not already aligned *)
tempSSP = SSP;
Shadow_stack_store 4 bytes of 0 to (SSP - 4)
SSP = SSP & 0xFFFFFFFFFFFFFFF8H
ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(tempPushLIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

NONCONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF (RPL > CPL) or (DPL ≠ CPL)
    THEN #GP(new code segment selector); FI;
IF segment not present
    THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
    THEN #SS(0); FI;
tempEIP := DEST(Offset);
IF target mode = Compatibility mode
    THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF IA32_EFER.LMA & CS.L
        tempPushLIP = RIP
    ELSE
        tempPushLIP = CSBASE + EIP;
    FI;
tempPushCS = CS;
FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS := DEST(CodeSegmentSelector);

```



```

(* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
ELSE
  IF OperandSize = 16
    THEN
      Push(CS);
      Push(IP);
      CS := DEST(CodeSegmentSelector);
      (* Segment descriptor information also loaded *)
      CS(RPL) := CPL;
      EIP := tempEIP;
    ELSE (* OperandSize = 64 *)
      Push(CS); (* Padded with 48 high-order bits *)
      Push(RIP);
      CS := DEST(CodeSegmentSelector);
      (* Segment descriptor information also loaded *)
      CS(RPL) := CPL;
      RIP := tempEIP;
  FI;
FI;
IF ShadowStackEnabled(CPL)
  IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
    (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
    IF (SSP & 0xFFFFFFFF00000000 != 0)
      THEN #GP(0); FI;
  FI;
(* align to 8 byte boundary if not already aligned *)
tempSSP = SSP;
Shadow_stack_store 4 bytes of 0 to (SSP - 4)
SSP = SSP & 0xFFFFFFFFFFFFFFF8H
ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
ShadowStackPush8B(tempPushLIP); (* Padded 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled(CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;
END;

CALL-GATE:
  IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); FI;
  IF call gate not present
    THEN #NP(call-gate selector); FI;
  IF call-gate code-segment selector is NULL
    THEN #GP(0); FI;

```

```

IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); FI;
Read call-gate code-segment descriptor;
IF call-gate code-segment descriptor does not indicate a code segment
or call-gate code-segment descriptor DPL > CPL
    THEN #GP(call-gate code-segment selector); FI;
IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
    THEN #GP(call-gate code-segment selector); FI;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); FI;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
FI;
END;

```

MORE-PRIVILEGE:

```

IF current TSS is 32-bit
    THEN
        TSSstackAddress := (new code-segment DPL * 8) + 4;
        IF (TSSstackAddress + 5) > current TSS limit
            THEN #TS(current TSS selector); FI;
        NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 4);
        NewESP := 4 bytes loaded from (TSS base + TSSstackAddress);
    ELSE
        IF current TSS is 16-bit
            THEN
                TSSstackAddress := (new code-segment DPL * 4) + 2
                IF (TSSstackAddress + 3) > current TSS limit
                    THEN #TS(current TSS selector); FI;
                NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                NewESP := 2 bytes loaded from (TSS base + TSSstackAddress);
            ELSE (* current TSS is 64-bit *)
                TSSstackAddress := (new code-segment DPL * 8) + 4;
                IF (TSSstackAddress + 7) > current TSS limit
                    THEN #TS(current TSS selector); FI;
                NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read new stack-segment descriptor;
    IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
writable data segment)
        THEN #TS(NewSS); FI
    IF IA32_EFER.LMA = 0 and new stack segment not present
        THEN #SS(NewSS); FI;
    IF CallGateSize = 32
        THEN
            IF new stack does not have room for parameters plus 16 bytes
                THEN #SS(NewSS); FI;

```

```

IF CallGate(InstructionPointer) not within new code-segment limit
    THEN #GP(0); FI;
SS := newSS; (* Segment descriptor information also loaded *)
ESP := newESP;
CS:EIP := CallGate(CS:InstructionPointer);
(* Segment descriptor information also loaded *)
Push(oldSS:oldESP); (* From calling procedure *)
temp := parameter count from call gate, masked to 5 bits;
Push(parameters from calling procedure's stack, temp)
Push(oldCS:oldEIP); (* Return address to calling procedure *)
ELSE
    IF CallGateSize = 16
        THEN
            IF new stack does not have room for parameters plus 8 bytes
                THEN #SS(NewSS); FI;
            IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                THEN #GP(0); FI;
            SS := newSS; (* Segment descriptor information also loaded *)
            ESP := newESP;
            CS:IP := CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            temp := parameter count from call gate, masked to 5 bits;
            Push(parameters from calling procedure's stack, temp)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE (* CallGateSize = 64 *)
            IF pushing 32 bytes on the stack would use a non-canonical address
                THEN #SS(NewSS); FI;
            IF (CallGate(InstructionPointer) is non-canonical)
                THEN #GP(0); FI;
            SS := NewSS; (* NewSS is NULL)
            RSP := NewESP;
            CS:IP := CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        FI;
    FI;
IF ShadowStackEnabled(CPL) AND CPL = 3
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
            ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                IA32_PL3_SSP := LA_adjust(SSP);
        FI;
    FI;
CPL := CodeSegment(DPL)
CS(RPL) := CPL
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := IA32_PLi_SSP; (* where i is the CPL *)
    IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then #GP *)
        THEN #GP(0); FI;
    (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region*)

```

```

IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
    #GP(0); FI;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;
expected_token_value = SSP          (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT    (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
    THEN #GP(0); FI;
IF oldSS.DPL != 3
    (* These stack pushes should not cause faults, VM exits, or data breakpoints *)
    (* Such events will apply to the earlier accesses to the token, which is in the same naturally aligned 32-byte region *)
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(oldSSP);
FI;
FI;
IF EndbranchEnabled (CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;
END;

SAME-PRIVILEGE:
IF CallGateSize = 32
    THEN
        IF stack does not have room for 8 bytes
            THEN #SS(0); FI;
        IF CallGate(InstructionPointer) not within code segment limit
            THEN #GP(0); FI;
        CS:EIP := CallGate(CS:EIP) (* Segment descriptor information also loaded *)
        Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
        If CallGateSize = 16
            THEN
                IF stack does not have room for 4 bytes
                    THEN #SS(0); FI;
                IF CallGate(InstructionPointer) not within code segment limit
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:instruction pointer);
                (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64)
                IF pushing 16 bytes on the stack touches non-canonical addresses
                    THEN #SS(0); FI;
                IF RIP non-canonical
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:instruction pointer);
                (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
        FI;
    FI;
CS(RPL) := CPL
IF ShadowStackEnabled(CPL)
    (* Align to next 8 byte boundary *)

```

```

    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)
    SSP = SSP & 0xFFFFFFFFFFFFF8H;
    (* push cs:lip:ssp on shadow stack *)
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

```

## TASK-GATE:

```

    IF task_gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task_gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

## TASK-STATE-SEGMENT:

```

    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

## Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

## Protected Mode Exceptions

#GP(0)	<p>If the target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the gate is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p> <p>If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.</p> <p>If “supervisor Shadow Stack” token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.</p> <p>If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).</p>
#GP(selector)	<p>If a code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment’s segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS’s segment selector.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>
#SS(selector)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.</p> <p>If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.</p> <p>If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.</p>
#NP(selector)	<p>If a code segment, data segment, call gate, task gate, or TSS is not present.</p>
#TS(selector)	<p>If the new stack segment selector and ESP are beyond the end of the TSS.</p> <p>If the new stack segment selector is NULL.</p> <p>If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.</p> <p>If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.</p>

	If the new stack segment is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the target offset is beyond the code segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the target offset is beyond the code segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

### 64-Bit Mode Exceptions

#GP(0)	<p>If a memory address is non-canonical.</p> <p>If target offset in destination operand is non-canonical.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the 64-bit gate is NULL.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p> <p>If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.</p> <p>If "supervisor Shadow Stack" token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit mode or compatibility mode, but SSP address in "super-visor shadow" stack token is beyond 4GB.</p> <p>If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).</p>
#GP(selector)	<p>If code segment or 64-bit call gate is outside descriptor table limits.</p> <p>If code segment or 64-bit call gate overlaps non-canonical space.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.</p> <p>If the upper type field of a 64-bit call gate is not 0x0.</p>

	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
	If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 CLRSSBSY m64	M	V/V	CET_SS	Clear busy flag in supervisor shadow stack token reference by m64.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
M	NA	ModRM:r/m (r, w)	NA	NA	NA

### Description

Clear busy flag in supervisor shadow stack token reference by m64. Subsequent to marking the shadow stack as not busy the SSP is loaded with value 0.

### Operation

```
IF (CR4.CET = 0)
  THEN #UD; FI;
```

```
IF (IA32_S_CET.SH_STK_EN = 0)
  THEN #UD; FI;
```

```
IF CPL > 0
  THEN GP(0); FI;
```

```
SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
  THEN #GP(0); FI;
```

```
expected_token_value = SSP_LA | BUSY_BIT (* busy bit - bit position 0 - must be set *)
```

```
new_token_value = SSP_LA (* Clear the busy bit *)
```

```
IF shadow_stack_lock_cmpxchg8b(SSP_LA, new_token_value, expected_token_value) != expected_token_value
  invalid_token := 1; FI
```

```
(* Set the CF if invalid token was detected *)
```

```
RFLAGS.CF = (invalid_token == 1) ? 1 : 0;
```

```
RFLAGS.ZF,PF,AF,OF,SF := 0;
```

```
SSP := 0
```

### Flags Affected

CF is set if an invalid token was detected, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

**Protected Mode Exceptions**

#UD	If the LOCK prefix is used. If CR4.CET = 0. IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If destination is located in a non-writable segment. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. If CPL is not 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#UD	The CLRSSBSY instruction is not recognized in real-address mode.
-----	--

**Virtual-8086 Mode Exceptions**

#UD	The CLRSSBSY instruction is not recognized in virtual-8086 mode.
-----	--

**Compatibility Mode Exceptions**

#UD	Same exceptions as in protected mode.
#GP(0)	Same exceptions as in protected mode.
#PF(fault-code)	If a page fault occurs.

**64-Bit Mode Exceptions**

#UD	If the LOCK prefix is used. If CR4.CET = 0. IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes. If CPL is not 0. If the memory address is in a non-canonical form. If token is invalid.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

## CPUID—CPU Identification

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F A2	CPUID	Z0	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.<sup>1</sup> The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using some Intel processors, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)2
CPUID.EAX = 1FH (* Returns V2 Extended Topology Enumeration leaf. *)2
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

### See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.
2. CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of CPUID leaf 1FH before using leaf 0BH.

**Table 3-8. Information Returned by CPUID Instruction**

Initial EAX Value	Information Provided about the Processor	
<i>Basic CPUID Information</i>		
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).
	EBX	Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value * 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31 - 24: Initial APIC ID**.
	ECX	Feature Information (see Figure 3-7 and Table 3-10).
	EDX	Feature Information (see Figure 3-8 and Table 3-11).
		<b>NOTES:</b> * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1. ** The 8-bit initial APIC ID in EBX[31:24] is replaced by the 32-bit x2APIC ID, available in Leaf 0BH and Leaf 1FH.
02H	EAX	Cache and TLB Information (see Table 3-12).
	EBX	Cache and TLB Information.
	ECX	Cache and TLB Information.
	EDX	Cache and TLB Information.
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		<b>NOTES:</b> Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
<i>Deterministic Cache Parameters Leaf</i>		
04H		<b>NOTES:</b> Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 244.
	EAX	Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p>Bits 07 - 05: Cache Level (starts at 1).                      Bit 08: Self Initializing cache level (does not need SW initialization).                      Bit 09: Fully Associative cache.</p> <p>Bits 13 - 10: Reserved.                      Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**, ***.                      Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.</p> <p>EBX Bits 11 - 00: L = System Coherency Line Size**.                      Bits 21 - 12: P = Physical Line partitions**.                      Bits 31 - 22: W = Ways of associativity**.</p> <p>ECX Bits 31-00: S = Number of Sets**.</p> <p>EDX Bit 00: Write-Back Invalidate/Invalidate.                      0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache.                      1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.</p> <p>Bit 01: Cache Inclusiveness.                      0 = Cache is not inclusive of lower cache levels.                      1 = Cache is inclusive of lower cache levels.</p> <p>Bit 02: Complex Cache Indexing.                      0 = Direct mapped cache.                      1 = A complex function is used to index the cache, potentially using all address bits.</p> <p>Bits 31 - 03: Reserved = 0.</p> <p><b>NOTES:</b></p> <p>* If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0.</p> <p>** Add one to the return value to get the result.</p> <p>***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache.</p> <p>**** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.</p> <p>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p>
	<i>MONITOR/MWAIT Leaf</i>
05H	<p>EAX Bits 15 - 00: Smallest monitor-line size in bytes (default is processor’s monitor granularity).                      Bits 31 - 16: Reserved = 0.</p> <p>EBX Bits 15 - 00: Largest monitor-line size in bytes (default is processor’s monitor granularity).                      Bits 31 - 16: Reserved = 0.</p> <p>ECX Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported.                      Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled.                      Bits 31 - 02: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	<p>Bits 03 - 00: Number of C0* sub C-states supported using MWAIT.            Bits 07 - 04: Number of C1* sub C-states supported using MWAIT.            Bits 11 - 08: Number of C2* sub C-states supported using MWAIT.            Bits 15 - 12: Number of C3* sub C-states supported using MWAIT.            Bits 19 - 16: Number of C4* sub C-states supported using MWAIT.            Bits 23 - 20: Number of C5* sub C-states supported using MWAIT.            Bits 27 - 24: Number of C6* sub C-states supported using MWAIT.            Bits 31 - 28: Number of C7* sub C-states supported using MWAIT.</p> <p><b>NOTE:</b>            * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.</p>
<i>Thermal and Power Management Leaf</i>		
06H	EAX	<p>Bit 00: Digital temperature sensor is supported if set.            Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]).            Bit 02: ARAT. APIC-Timer-always-running feature is supported if set.            Bit 03: Reserved.            Bit 04: PLN. Power limit notification controls are supported if set.            Bit 05: ECMD. Clock modulation duty cycle extension is supported if set.            Bit 06: PTM. Package thermal management is supported if set.            Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set.            Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set.            Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set.            Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set.            Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set.            Bit 12: Reserved.            Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set.            Bit 14: Intel® Turbo Boost Max Technology 3.0 available.            Bit 15: HWP Capabilities. Highest Performance change is supported if set.            Bit 16: HWP PECL override is supported if set.            Bit 17: Flexible HWP is supported if set.            Bit 18: Fast access mode for the IA32_HWP_REQUEST MSR is supported if set.            Bit 19: HW_FEEDBACK. IA32_HW_FEEDBACK_PTR MSR, IA32_HW_FEEDBACK_CONFIG MSR, IA32_PACKAGE_THERM_STATUS MSR bit 26, and IA32_PACKAGE_THERM_INTERRUPT MSR bit 25 are supported if set.            Bit 20: Ignoring Idle Logical Processor HWP request is supported if set.            Bits 31 - 21: Reserved.</p>
	EBX	<p>Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor.            Bits 31 - 04: Reserved.</p>
	ECX	<p>Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency.            Bits 02 - 01: Reserved = 0.            Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1BOH).            Bits 31 - 04: Reserved = 0.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>EDX</p> <p>Bits 7-0: Bitmap of supported hardware feedback interface capabilities.            0 = When set to 1, indicates support for performance capability reporting.            1 = When set to 1, indicates support for energy efficiency capability reporting.            2-7 = Reserved</p> <p>Bits 11-8: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages; add one to the return value to get the result.</p> <p>Bits 31-16: Index (starting at 0) of this logical processor's row in the hardware feedback interface structure. Note that on some parts the index may be same for multiple logical processors. On some parts the indices may not be contiguous, i.e., there may be unused rows in the hardware feedback interface structure.</p> <p><b>NOTE:</b>            Bits 0 and 1 will always be set together.</p>
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>	
07H	<p style="text-align: center;">Sub-leaf 0 (Input ECX = 0). *</p> <p>EAX</p> <p>Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.</p> <p>EBX</p> <p>Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1.            Bit 01: IA32_TSC_ADJUST MSR is supported if 1.            Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1.            Bit 03: BMI1.            Bit 04: HLE.            Bit 05: AVX2.            Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1.            Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1.            Bit 08: BMI2.            Bit 09: Supports Enhanced REP MOVSB/STOSB if 1.            Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers.            Bit 11: RTM.            Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1.            Bit 13: Deprecates FPU CS and FPU DS values if 1.            Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1.            Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1.            Bit 16: AVX512F.            Bit 17: AVX512DQ.            Bit 18: RDSEED.            Bit 19: ADX.            Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1.            Bit 21: AVX512_IFMA.            Bit 22: Reserved.            Bit 23: CLFLUSHOPT.            Bit 24: CLWB.            Bit 25: Intel Processor Trace.            Bit 26: AVX512PF. (Intel® Xeon Phi™ only.)            Bit 27: AVX512ER. (Intel® Xeon Phi™ only.)            Bit 28: AVX512CD.            Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1.            Bit 30: AVX512BW.            Bit 31: AVX512VL.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
ECX	<p>Bit 00: PREFETCHWT1. (Intel® Xeon Phi™ only.)</p> <p>Bit 01: AVX512_VBMI.</p> <p>Bit 02: UMIP. Supports user-mode instruction prevention if 1.</p> <p>Bit 03: PKU. Supports protection keys for user-mode pages if 1.</p> <p>Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions).</p> <p>Bit 05: WAITPKG.</p> <p>Bit 06: AVX512_VBMI2.</p> <p>Bit 07: CET_SS. Supports CET shadow stack features if 1. Processors that set this bit define bits 1:0 of the IA32_U_CET and IA32_S_CET MSRs. Enumerates support for the following MSRs: IA32_INTERRUPT_SPP_TABLE_ADDR, IA32_PL3_SSP, IA32_PL2_SSP, IA32_PL1_SSP, and IA32_PLO_SSP.</p> <p>Bit 08: GFNI.</p> <p>Bit 09: VAES.</p> <p>Bit 10: VPCLMULQDQ.</p> <p>Bit 11: AVX512_VNNI.</p> <p>Bit 12: AVX512_BITALG.</p> <p>Bits 13: TME_EN. If 1, the following MSRs are supported: IA32_TME_CAPABILITY, IA32_TME_ACTIVATE, IA32_TME_EXCLUDE_MASK, and IA32_TME_EXCLUDE_BASE.</p> <p>Bit 14: AVX512_VPOPCNTDQ.</p> <p>Bit 15: Reserved.</p> <p>Bit 16: LA57. Supports 57-bit linear addresses and five-level paging if 1.</p> <p>Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode.</p> <p>Bit 22: RDPID and IA32_TSC_AUX are available if 1.</p> <p>Bit 23: KL. Supports Key Locker if 1.</p> <p>Bit 24: Reserved.</p> <p>Bit 25: CLDEMOTE. Supports cache line demote if 1.</p> <p>Bit 26: Reserved.</p> <p>Bit 27: MOVDIRI. Supports MOVDIRI if 1.</p> <p>Bit 28: MOVDIR64B. Supports MOVDIR64B if 1.</p> <p>Bit 29: Reserved.</p> <p>Bit 30: SGX_LC. Supports SGX Launch Configuration if 1.</p> <p>Bit 31: PKS. Supports protection keys for supervisor-mode pages if 1.</p>
EDX	<p>Bit 01: Reserved.</p> <p>Bit 02: AVX512_4VNNIW. (Intel® Xeon Phi™ only.)</p> <p>Bit 03: AVX512_4FMAPS. (Intel® Xeon Phi™ only.)</p> <p>Bit 04: Fast Short REP MOV.</p> <p>Bits 07-05: Reserved.</p> <p>Bit 08: AVX512_VP2INTERSECT.</p> <p>Bit 09: Reserved.</p> <p>Bit 10: MD_CLEAR supported.</p> <p>Bits 14-11: Reserved.</p> <p>Bit 15: Hybrid. If 1, the processor is identified as a hybrid part.</p> <p>Bits 17-16: Reserved.</p> <p>Bit 18: PCONFIG. Supports PCONFIG if 1.</p> <p>Bit 19: Reserved.</p> <p>Bit 20: CET_IBT. Supports CET indirect branch tracking features if 1. Processors that set this bit define bits 5:2 and bits 63:10 of the IA32_U_CET and IA32_S_CET MSRs.</p> <p>Bits 25 - 21: Reserved.</p> <p>Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB).</p>



**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p>Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP).</p> <p>Bit 28: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH).</p> <p>Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR.</p> <p>Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR.</p> <p>IA32_CORE_CAPABILITIES is an architectural MSR that enumerates model-specific features. A bit being set in this MSR indicates that a model specific feature is supported; software must still consult CPUID family/model/stepping to determine the behavior of the enumerated feature as features enumerated in IA32_CORE_CAPABILITIES may have different behavior on different processor models.</p> <p>Additionally, on hybrid parts (CPUID.07H.0H:EDX[15]=1), software must consult the native model ID and core type from the Hybrid Information Enumeration Leaf.</p> <p>Bit 31: Enumerates support for Speculative Store Bypass Disable (SSBD). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[2] (SSBD).</p> <p><b>NOTE:</b></p> <p>* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.</p>
<i>Structured Extended Feature Enumeration Sub-leaf (EAX = 07H, ECX = 1)</i>	
07H	<p><b>NOTES:</b></p> <p>Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.</p> <p>EAX This field reports 0 if the sub-leaf index, 1, is invalid. Bits 04-00: Reserved. Bit 05: AVX512_BF16. Vector Neural Network Instructions supporting BFLOAT16 inputs and conversion instructions from IEEE single precision. Bits 31-06: Reserved.</p> <p>EBX This field reports 0 if the sub-leaf index, 1, is invalid; otherwise it is reserved.</p> <p>ECX This field reports 0 if the sub-leaf index, 1, is invalid; otherwise it is reserved.</p> <p>EDX This field reports 0 if the sub-leaf index, 1, is invalid; otherwise it is reserved.</p>
<i>Direct Cache Access Information Leaf</i>	
09H	<p>EAX Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).</p> <p>EBX Reserved.</p> <p>ECX Reserved.</p> <p>EDX Reserved.</p>
<i>Architectural Performance Monitoring Leaf</i>	
0AH	<p>EAX Bits 07 - 00: Version ID of architectural performance monitoring. Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor. Bits 23 - 16: Bit width of general-purpose, performance monitoring counter. Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events. Architectural event x is supported if EBX[x]=0 &amp;&amp; EAX[31:24]&gt;x.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>EBX Bit 00: Core cycle event not available if 1 or if EAX[31:24]&lt;1.            Bit 01: Instruction retired event not available if 1 or if EAX[31:24]&lt;2.            Bit 02: Reference cycles event not available if 1 or if EAX[31:24]&lt;3.            Bit 03: Last-level cache reference event not available if 1 or if EAX[31:24]&lt;4.            Bit 04: Last-level cache misses event not available if 1 or if EAX[31:24]&lt;5.            Bit 05: Branch instruction retired event not available if 1 or if EAX[31:24]&lt;6.            Bit 06: Branch mispredict retired event not available if 1 or if EAX[31:24]&lt;7.            Bit 07: Top-down slots event not available if 1 or if EAX[31:24]&lt;8.            Bits 31 - 08: Reserved = 0.</p> <p>ECX Bits 31 - 00: Supported fixed counters bit mask. Fixed-function performance counter 'i' is supported if bit 'i' is 1 (first counter index starts at zero). It is recommended to use the following logic to determine if a Fixed Counter is supported: FxCtr[i]_is_supported := ECX[i]    (EDX[4:0] &gt; i);</p> <p>EDX Bits 04 - 00: Number of contiguous fixed-function performance counters starting from 0 (if Version ID &gt; 1).            Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID &gt; 1).            Bits 14 - 13: Reserved = 0.            Bit 15: AnyThread deprecation.            Bits 31 - 16: Reserved = 0.</p>
<i>Extended Topology Enumeration Leaf</i>	
OBH	<p><b>NOTES:</b></p> <p><i>CPUID leaf 1FH is a preferred superset to leaf OBH. Intel recommends first checking for the existence of Leaf 1FH before using leaf OBH.</i></p> <p>Most of Leaf OBH output depends on the initial value in ECX.            The EDX output of leaf OBH is always valid and does not vary with input value in ECX.            Output value in ECX[7:0] always equals input value in ECX[7:0].            Sub-leaf index 0 enumerates SMT level. Each subsequent higher sub-leaf index enumerates a higher-level topological entity in hierarchical order.            For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0.            If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX &gt; n also return 0 in ECX[15:8].</p> <p>EAX Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level.            Bits 31 - 05: Reserved.</p> <p>EBX Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**.            Bits 31 - 16: Reserved.</p> <p>ECX Bits 07 - 00: Level number. Same value in ECX input.            Bits 15 - 08: Level type***.            Bits 31 - 16: Reserved.</p> <p>EDX Bits 31 - 00: x2APIC ID the current logical processor.</p> <p><b>NOTES:</b>            * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p>** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.</p> <p>*** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding:                      0: Invalid.                      1: SMT.                      2: Core.                      3-255: Reserved.</p>
<i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>	
<p>0DH</p>	<p><b>NOTES:</b>                      Leaf 0DH main leaf (ECX = 0).</p> <p>EAX     Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCRO. XCRO[n] can be set to 1 only if EAX[n] is 1.                      Bit 00: x87 state.                      Bit 01: SSE state.                      Bit 02: AVX state.                      Bits 04 - 03: MPX state.                      Bits 07 - 05: AVX-512 state.                      Bit 08: Used for IA32_XSS.                      Bit 09: PKRU state.                      Bits 12 - 10: Reserved.                      Bit 13: Used for IA32_XSS.                      Bits 15 - 14: Reserved.                      Bit 16: Used for IA32_XSS.                      Bits 31 - 17: Reserved.</p> <p>EBX     Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCRO. May be different than ECX if some features at the end of the XSAVE save area are not enabled.</p> <p>ECX     Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCRO.</p> <p>EDX     Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCRO. XCRO[n+32] can be set to 1 only if EDX[n] is 1.                      Bits 31 - 00: Reserved.</p>
<i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>	
<p>0DH</p>	<p>EAX     Bit 00: XSAVEOPT is available.                      Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set.                      Bit 02: Supports XGETBV with ECX = 1 if set.                      Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set.                      Bits 31 - 04: Reserved.</p> <p>EBX     Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCRO   IA32_XSS.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p>ECX    Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1.  Bits 07 - 00: Used for XCRO.  Bit 08: PT state.  Bit 09: Used for XCRO.  Bit 10: Reserved.  Bit 11: CET user state.  Bit 12: CET supervisor state.  Bit 13: HDC state.  Bits 15 - 14: Reserved.  Bit 16: HWP state.  Bits 31 - 17: Reserved.</p> <p>EDX    Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1.  Bits 31 - 00: Reserved.</p>
<i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n &gt; 1)</i>	
0DH	<p><b>NOTES:</b>  Leaf 0DH output depends on the initial value in ECX.  Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCRO register or the IA32_XSS MSR.  * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (<math>0 \leq n \leq 31</math>) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (<math>32 \leq n \leq 63</math>) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].</p> <p>EAX    Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n.</p> <p>EBX    Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area.  This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCRO register*.</p> <p>ECX    Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCRO.  Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component).  Bits 31 - 02 are reserved.  This field reports 0 if the sub-leaf index, n, is invalid*.</p> <p>EDX    This field reports 0 if the sub-leaf index, n, is invalid*; otherwise it is reserved.</p>
<i>Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)</i>	
0FH	<p><b>NOTES:</b>  Leaf 0FH output depends on the initial value in ECX.  Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.</p> <p>EAX    Reserved.</p> <p>EBX    Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.</p> <p>ECX    Reserved.</p> <p>EDX    Bit 00: Reserved.  Bit 01: Supports L3 Cache Intel RDT Monitoring if 1.  Bits 31 - 02: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
<i>L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)</i>	
0FH	<p><b>NOTES:</b> Leaf 0FH output depends on the initial value in ECX.</p> <p>EAX Reserved.</p> <p>EBX Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes) and Memory Bandwidth Monitoring (MBM) metrics.</p> <p>ECX Maximum range (zero-based) of RMID of this resource type.</p> <p>EDX Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31 - 03: Reserved.</p>
<i>Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>	
10H	<p><b>NOTES:</b> Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.</p> <p>EAX Reserved.</p> <p>EBX Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31 - 04: Reserved.</p> <p>ECX Reserved.</p> <p>EDX Reserved.</p>
<i>L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 1)</i>	
10H	<p><b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.</p> <p>EAX Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31 - 05: Reserved.</p> <p>EBX Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.</p> <p>ECX Bits 01 - 00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31 - 03: Reserved.</p> <p>EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.</p>
<i>L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 2)</i>	
10H	<p><b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.</p> <p>EAX Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31 - 05: Reserved.</p> <p>EBX Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.</p> <p>ECX Bits 31 - 00: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
<i>Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)</i>	
10H	<p><b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.</p> <p>EAX Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID. Add one to the return value to get the result. Bits 31 - 12: Reserved.</p> <p>EBX Bits 31 - 00: Reserved.</p> <p>ECX Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.</p> <p>EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.</p>
<i>Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)</i>	
12H	<p><b>NOTES:</b> Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.</p> <p>EAX Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bits 04 - 02: Reserved. Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVIRTUAL, EDECVIRTUAL, and ESETCONTEXT. Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC. Bits 31 - 07: Reserved.</p> <p>EBX Bits 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.</p> <p>ECX Bits 31 - 00: Reserved.</p> <p>EDX Bits 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is 2<sup>(EDX[7:0])</sup>. Bits 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is 2<sup>(EDX[15:8])</sup>. Bits 31 - 16: Reserved.</p>
<i>Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)</i>	
12H	<p><b>NOTES:</b> Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.</p> <p>EAX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.</p> <p>EBX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.</p> <p>ECX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.</p> <p>EDX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.</p>
<i>Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)</i>	
12H	<p><b>NOTES:</b> Leaf 12H sub-leaf 2 or higher (ECX &gt;= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p>EAX Bit 03 - 00: Sub-leaf Type                      0000b: Indicates this sub-leaf is invalid.                      0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section.                      All other type encodings are reserved.</p> <p>Type 0000b. This sub-leaf is invalid.                      EDX:ECX:EBX:EAX return 0.</p> <p>Type 0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows.                      EAX[11:04]: Reserved (enumerate 0).                      EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section.</p> <p>EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section.                      EBX[31:20]: Reserved.</p> <p>ECX[03:00]: EPC section property encoding defined as follows:                      If ECX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0.                      If ECX[3:0] 0001b, then this section has confidentiality and integrity protection.                      If ECX[3:0] 0010b, then this section has confidentiality protection only.                      All other encodings are reserved.                      ECX[11:04]: Reserved (enumerate 0).                      ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.</p> <p>EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory.                      EDX[31:20]: Reserved.</p>
<i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>	
14H	<p><b>NOTES:</b>                      Leaf 14H main leaf (ECX = 0).</p> <p>EAX Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.</p> <p>EBX Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed.                      Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode.                      Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset.                      Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets.                      Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets.                      Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation.                      Bit 06: If 1, indicates support for PSB and PMI preservation. Writes can set IA32_RTIT_CTL[56] (InjectPsb-PmiOnEnable), enabling the processor to set IA32_RTIT_STATUS[7] (PendTopaPMI) and/or IA32_RTIT_STATUS[6] (PendPSB) in order to preserve ToPA PMIs and/or PSBs otherwise lost due to Intel PT disable. Writes can also set PendToPAPMI and PendPSB.                      Bit 31 - 07: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	ECX	<p>Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSR can be accessed.</p> <p>Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOffsetTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.</p> <p>Bit 02: If 1, indicates support of Single-Range Output scheme.</p> <p>Bit 03: If 1, indicates support of output to Trace Transport subsystem.</p> <p>Bit 30 - 04: Reserved.</p> <p>Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.</p>
	EDX	Bits 31 - 00: Reserved.
<i>Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)</i>		
14H	EAX	<p>Bits 02 - 00: Number of configurable Address Ranges for filtering.</p> <p>Bits 15 - 03: Reserved.</p> <p>Bits 31 - 16: Bitmap of supported MTC period encodings.</p>
	EBX	<p>Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings.</p> <p>Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.</p>
	ECX	Bits 31 - 00: Reserved.
	EDX	Bits 31 - 00: Reserved.
<i>Time Stamp Counter and Nominal Core Crystal Clock Information Leaf</i>		
15H		<p><b>NOTES:</b></p> <p>If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated.</p> <p>EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency.</p> <p>If ECX is 0, the nominal core crystal clock frequency is not enumerated.</p> <p>"TSC frequency" = "core crystal clock frequency" * EBX/EAX.</p> <p>The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</p>
	EAX	Bits 31 - 00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31 - 00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31 - 00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31 - 00: Reserved = 0.
<i>Processor Frequency Information Leaf</i>		
16H	EAX	<p>Bits 15 - 00: Processor Base Frequency (in MHz).</p> <p>Bits 31 - 16: Reserved = 0.</p>
	EBX	<p>Bits 15 - 00: Maximum Frequency (in MHz).</p> <p>Bits 31 - 16: Reserved = 0.</p>
	ECX	<p>Bits 15 - 00: Bus (Reference) Frequency (in MHz).</p> <p>Bits 31 - 16: Reserved = 0.</p>
	EDX	Reserved.



**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p><b>NOTES:</b>                      * Data is returned from this interface in accordance with the processor’s specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.</p> <p>While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.</p>
<i>System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)</i>	
17H	<p><b>NOTES:</b>                      Leaf 17H main leaf (ECX = 0).                      Leaf 17H output depends on the initial value in ECX.                      Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String.                      Leaf 17H is valid if MaxSOCID_Index &gt;= 3.                      Leaf 17H sub-leaves 4 and above are reserved.</p> <p>EAX      Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.                      EBX      Bits 15 - 00: SOC Vendor ID.                                Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel.                                Bits 31 - 17: Reserved = 0.                      ECX      Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.                      EDX      Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.</p>
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)</i>	
17H	<p>EAX      Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.                      EBX      Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.                      ECX      Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.                      EDX      Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.</p> <p><b>NOTES:</b>                      Leaf 17H output depends on the initial value in ECX.                      SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H.                      The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.</p>
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX &gt; MaxSOCID_Index)</i>	
17H	<p><b>NOTES:</b>                      Leaf 17H output depends on the initial value in ECX.</p> <p>EAX      Bits 31 - 00: Reserved = 0.                      EBX      Bits 31 - 00: Reserved = 0.                      ECX      Bits 31 - 00: Reserved = 0.                      EDX      Bits 31 - 00: Reserved = 0.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
<i>Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)</i>	
18H	<p><b>NOTES:</b></p> <p>Each sub-leaf enumerates a different address translation structure.  If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.</p> <p>* Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the <i>Intel® 64 and IA-32 Architectures Optimization Reference Manual</i> for details of a particular product.</p> <p>** Add one to the return value to get the result.</p> <p>EAX     Bits 31 - 00: Reports the maximum input value of supported sub-leaf in leaf 18H.</p> <p>EBX     Bit 00: 4K page size entries supported by this structure.  Bit 01: 2MB page size entries supported by this structure.  Bit 02: 4MB page size entries supported by this structure.  Bit 03: 1 GB page size entries supported by this structure.  Bits 07 - 04: Reserved.  Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure).  Bits 15 - 11: Reserved.  Bits 31 - 16: W = Ways of associativity.</p> <p>ECX     Bits 31 - 00: S = Number of Sets.</p> <p>EDX     Bits 04 - 00: Translation cache type field.  00000b: Null (indicates this sub-leaf is not valid).  00001b: Data TLB.  00010b: Instruction TLB.  00011b: Unified TLB*.  00100b: Load Only TLB. Hit on loads; fills on both loads and stores.  00101b: Store Only TLB. Hit on stores; fill on stores.  All other encodings are reserved.  Bits 07 - 05: Translation cache level (starts at 1).  Bit 08: Fully associative structure.  Bits 13 - 09: Reserved.  Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache**  Bits 31 - 26: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
<i>Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)</i>	
18H	<p><b>NOTES:</b></p> <p>Each sub-leaf enumerates a different address translation structure.</p> <p>If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.</p> <p>* Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the <i>Intel® 64 and IA-32 Architectures Optimization Reference Manual</i> for details of a particular product.</p> <p>** Add one to the return value to get the result.</p> <p>EAX      Bits 31 - 00: Reserved.</p> <p>EBX      Bit 00: 4K page size entries supported by this structure.  Bit 01: 2MB page size entries supported by this structure.  Bit 02: 4MB page size entries supported by this structure.  Bit 03: 1 GB page size entries supported by this structure.  Bits 07 - 04: Reserved.  Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure).  Bits 15 - 11: Reserved.  Bits 31 - 16: W = Ways of associativity.</p> <p>ECX      Bits 31 - 00: S = Number of Sets.</p> <p>EDX      Bits 04 - 00: Translation cache type field.  0000b: Null (indicates this sub-leaf is not valid).  0001b: Data TLB.  0010b: Instruction TLB.  0011b: Unified TLB*.  All other encodings are reserved.  Bits 07 - 05: Translation cache level (starts at 1).  Bit 08: Fully associative structure.  Bits 13 - 09: Reserved.  Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache**  Bits 31 - 26: Reserved.</p>
<i>Key Locker Leaf (EAX = 19H)</i>	
19H	<p>EAX      Bit 00: Key Locker restriction of CPL0-only supported.  Bit 01: Key Locker restriction of no-encrypt supported.  Bit 02: Key Locker restriction of no-decrypt supported.  Bits 31-03: Reserved.</p> <p>EBX      Bit 00: AESKLE. If 1, the AES Key Locker instructions are fully enabled.  Bit 01: Reserved.  Bit 02: If 1, the AES wide Key Locker instructions are supported.  Bit 03: Reserved.  Bit 04: If 1, the platform supports the Key Locker MSRs (IA32_COPY_LOCAL_TO_PLATFORM, IA23_COPY_PLATFORM_TO_LOCAL, IA32_COPY_STATUS, and IA32_IWKEYBACKUP_STATUS) and backing up the internal wrapping key.  Bits 31-05: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 00: If 1, the NoBackup parameter to LOADIWKEY is supported. Bit 01: If 1, KeySource encoding of 1 (randomization of the internal wrapping key) is supported. Bits 31 - 02: Reserved.
	EDX	Reserved.
<i>Hybrid Information Enumeration Leaf (EAX = 1AH, ECX = 0)</i>		
1AH	EAX	Enumerates the native model ID and core type. Bits 31-24: Core type 10H: Reserved 20H: Intel Atom® 30H: Reserved 40H: Intel® Core™ Bits 23-0: Native model ID of the core. The core-type and native mode ID can be used to uniquely identify the microarchitecture of the core. This native model ID is not unique across core types, and not related to the model ID reported in CPUID leaf 01H, and does not identify the SOC.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
<i>PCONFIG Information Sub-leaf (EAX = 1BH, ECX ≥ 0)</i>		
1BH	For details on this sub-leaf, see “INPUT EAX = 1BH: Returns PCONFIG Information” on page 3-246. <b>NOTE:</b> Leaf 1BH is supported if CPUID.(EAX=07H, ECX=0H);EDX[18] = 1.	
<i>V2 Extended Topology Enumeration Leaf</i>		
1FH	<p><b>NOTES:</b></p> <p><i>CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH and using this if available.</i></p> <p>Most of Leaf 1FH output depends on the initial value in ECX.</p> <p>The EDX output of leaf 1FH is always valid and does not vary with input value in ECX.</p> <p>Output value in ECX[7:0] always equals input value in ECX[7:0].</p> <p>Sub-leaf index 0 enumerates SMT level. Each subsequent higher sub-leaf index enumerates a higher-level topological entity in hierarchical order.</p> <p>For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0.</p> <p>If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX &gt; n also return 0 in ECX[15:8].</p>	
	EAX	Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31 - 05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31 - 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31 - 00: x2APIC ID the current logical processor.
	<p><b>NOTES:</b></p> <p>* Software should use this field (EAX[4:0]) to enumerate processor topology of the system.</p>	

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	<p>** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.</p> <p>*** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding:  0: Invalid.  1: SMT.  2: Core.  3: Module.  4: Tile.  5: Die.  6-255: Reserved.</p>	
<i>Unimplemented CPUID Leaf Functions</i>		
21H	Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is 21H. If the value returned by CPUID.0:EAX (the maximum input value for basic CPUID information) is at least 21H, 0 is returned in the registers EAX, EBX, ECX, and EDX. Otherwise, the data for the highest basic information leaf is returned.	
40000000H - 4FFFFFFFH	Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.	
<i>Extended Function CPUID Information</i>		
80000000H	EAX EBX ECX EDX	Maximum Input Value for Extended Function CPUID Information. Reserved. Reserved. Reserved.
80000001H	EAX EBX ECX	Extended Processor Signature and Feature Bits. Reserved. Bit 00: LAHF/SAHF available in 64-bit mode.* Bits 04 - 01: Reserved. Bit 05: LZCNT. Bits 07 - 06: Reserved. Bit 08: PREFETCHW. Bits 31 - 09: Reserved.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor																	
	EDX	Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET.** Bits 19 - 12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25 - 21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31 - 30: Reserved = 0.  <b>NOTES:</b> * LAHF and SAHF are always available in other modes, regardless of the enumeration of this feature flag. ** Intel processors support SYSCALL and SYSRET only in 64-bit mode. This feature flag is always enumerated as 0 outside 64-bit mode.																
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.																
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.																
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.																
80000005H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.																
80000006H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Bits 07 - 00: Cache Line size in bytes. Bits 11 - 08: Reserved. Bits 15 - 12: L2 Associativity field *. Bits 31 - 16: Cache size in 1K units. Reserved = 0.  <b>NOTES:</b> * L2 associativity field encodings: <table style="width: 100%; border: none;"> <tr> <td>00H - Disabled</td> <td>08H - 16 ways</td> </tr> <tr> <td>01H - 1 way (direct mapped)</td> <td>09H - Reserved</td> </tr> <tr> <td>02H - 2 ways</td> <td>0AH - 32 ways</td> </tr> <tr> <td>03H - Reserved</td> <td>0BH - 48 ways</td> </tr> <tr> <td>04H - 4 ways</td> <td>0CH - 64 ways</td> </tr> <tr> <td>05H - Reserved</td> <td>0DH - 96 ways</td> </tr> <tr> <td>06H - 8 ways</td> <td>0EH - 128 ways</td> </tr> <tr> <td>07H - See CPUID leaf 04H, sub-leaf 2**</td> <td>0FH - Fully associative</td> </tr> </table> ** CPUID leaf 04H provides details of deterministic cache parameters, including the L2 cache in sub-leaf 2	00H - Disabled	08H - 16 ways	01H - 1 way (direct mapped)	09H - Reserved	02H - 2 ways	0AH - 32 ways	03H - Reserved	0BH - 48 ways	04H - 4 ways	0CH - 64 ways	05H - Reserved	0DH - 96 ways	06H - 8 ways	0EH - 128 ways	07H - See CPUID leaf 04H, sub-leaf 2**	0FH - Fully associative
00H - Disabled	08H - 16 ways																	
01H - 1 way (direct mapped)	09H - Reserved																	
02H - 2 ways	0AH - 32 ways																	
03H - Reserved	0BH - 48 ways																	
04H - 4 ways	0CH - 64 ways																	
05H - Reserved	0DH - 96 ways																	
06H - 8 ways	0EH - 128 ways																	
07H - See CPUID leaf 04H, sub-leaf 2**	0FH - Fully associative																	

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
80000007H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Bits 07 - 00: Reserved = 0. Bit 08: Invariant TSC available if 1. Bits 31 - 09: Reserved = 0.
80000008H	EAX  EBX  ECX EDX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0.  Bits 08-00: Reserved = 0. Bit 09: WBNOINVD is available if 1. Bits 31-10: Reserved = 0. Reserved = 0. Reserved = 0.  <b>NOTES:</b> * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

**INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String**

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

```
EBX := 756e6547h (* "Genu", with G in the low eight bits of BL *)
EDX := 49656e69h (* "inel", with i in the low eight bits of DL *)
ECX := 6c65746eh (* "ntel", with n in the low eight bits of CL *)
```

**INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information**

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

**IA32\_BIOS\_SIGN\_ID Returns Microcode Update Signature**

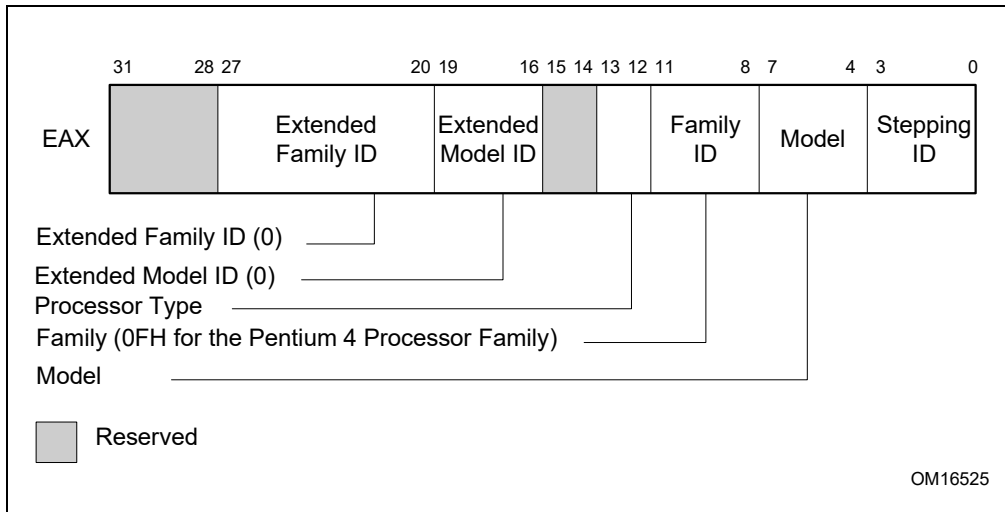
For processors that support the microcode update facility, the IA32\_BIOS\_SIGN\_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**INPUT EAX = 01H: Returns Model, Family, Stepping Information**

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.



**Figure 3-6. Version Information Returned by CPUID in EAX**

**Table 3-9. Processor Type Field**

Type	Encoding
Original OEM Processor	00B
Intel OverDrive <sup>®</sup> Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

#### NOTE

See Chapter 20 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```

IF Family_ID ≠ 0FH
  THEN DisplayFamily = Family_ID;
  ELSE DisplayFamily = Extended_Family_ID + Family_ID;
  (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show DisplayFamily as HEX field. *)

```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```

IF (Family_ID = 06H or Family_ID = 0FH)
  THEN DisplayModel = (Extended_Model_ID << 4) + Model_ID;
  (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
  ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)

```



**INPUT EAX = 01H: Returns Additional Information in EBX**

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

**INPUT EAX = 01H: Returns Feature Information in ECX and EDX**

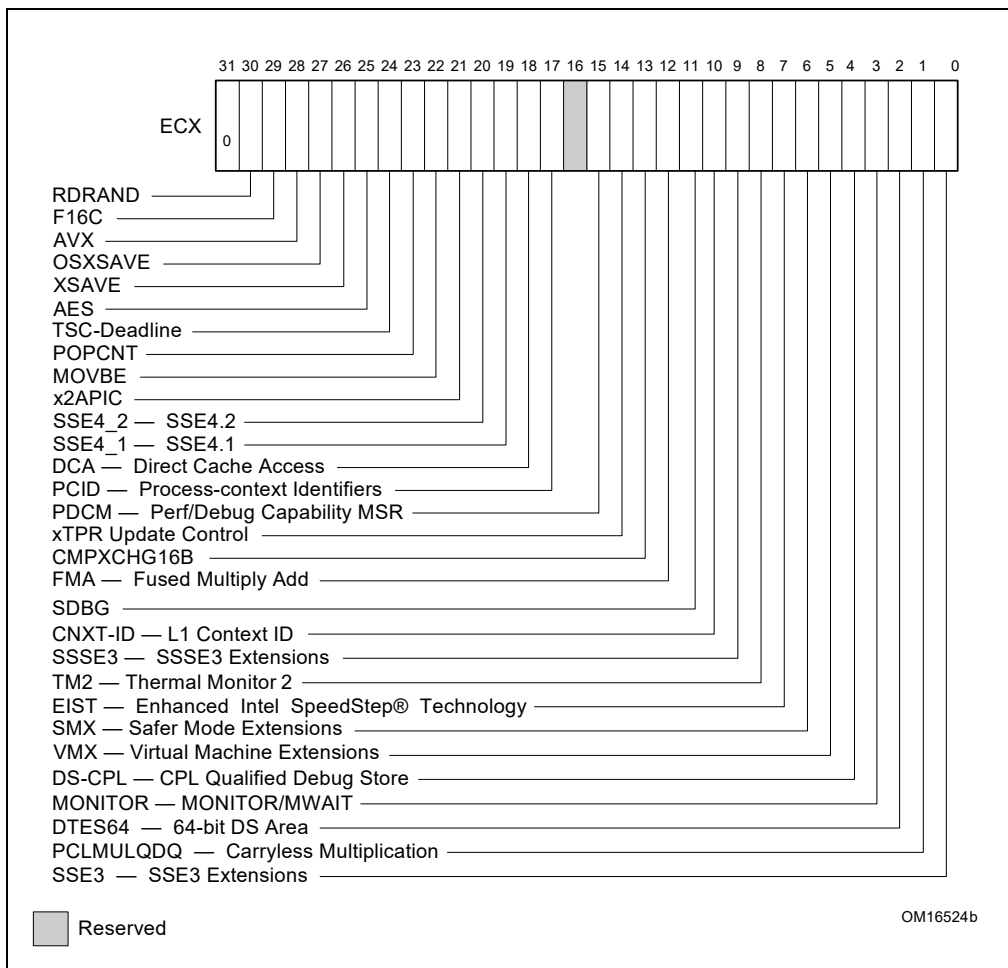
When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

**NOTE**

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.



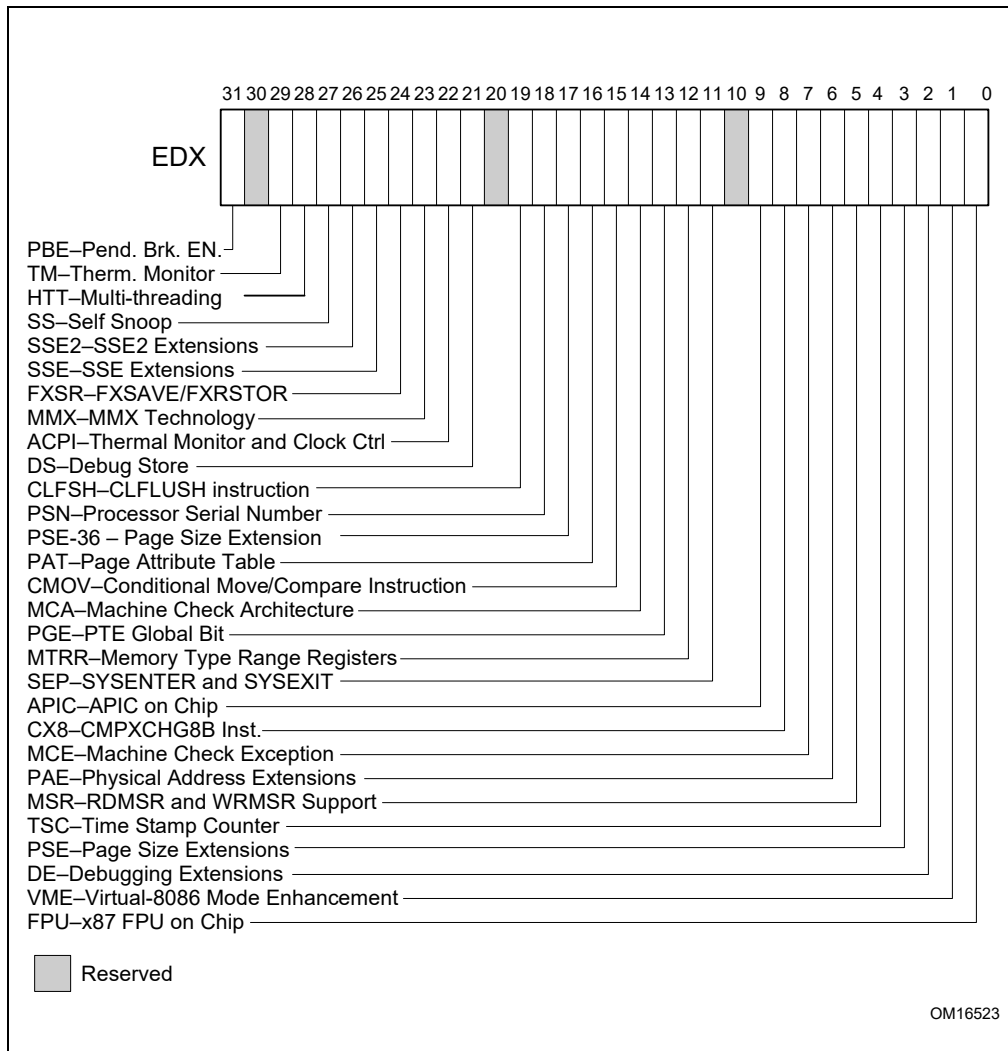
**Figure 3-7. Feature Information Returned in the ECX Register**

**Table 3-10. Feature Information Returned in the ECX Register**

Bit #	Mnemonic	Description
0	SSE3	<b>Streaming SIMD Extensions 3 (SSE3).</b> A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	<b>PCLMULQDQ.</b> A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	<b>64-bit DS Area.</b> A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	<b>MONITOR/MWAIT.</b> A value of 1 indicates the processor supports this feature.
4	DS-CPL	<b>CPL Qualified Debug Store.</b> A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	<b>Virtual Machine Extensions.</b> A value of 1 indicates that the processor supports this technology.
6	SMX	<b>Safer Mode Extensions.</b> A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”.
7	EIST	<b>Enhanced Intel SpeedStep® technology.</b> A value of 1 indicates that the processor supports this technology.
8	TM2	<b>Thermal Monitor 2.</b> A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.
10	CNXT-ID	<b>L1 Context ID.</b> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	<b>CMPXCHG16B Available.</b> A value of 1 indicates that the feature is available. See the “CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes” section in this chapter for a description.
14	xTPR Update Control	<b>xTPR Update Control.</b> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	<b>Perfmon and Debug Capability:</b> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	<b>Process-context identifiers.</b> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4_1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4_2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor’s local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCRO.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCRO and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.

**Table 3-10. Feature Information Returned in the ECX Register (Contd.)**

Bit #	Mnemonic	Description
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.



**Figure 3-8. Feature Information Returned in the EDX Register**

**Table 3-11. More on Feature Information Returned in the EDX Register**

Bit #	Mnemonic	Description
0	FPU	<b>Floating Point Unit On-Chip.</b> The processor contains an x87 FPU.
1	VME	<b>Virtual 8086 Mode Enhancements.</b> Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	<b>Debugging Extensions.</b> Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	<b>Page Size Extension.</b> Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	<b>Time Stamp Counter.</b> The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	<b>Model Specific Registers RDMSR and WRMSR Instructions.</b> The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	<b>Physical Address Extension.</b> Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	<b>Machine Check Exception.</b> Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	<b>CMPXCHG8B Instruction.</b> The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	<b>APIC On-Chip.</b> The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	<b>SYSENTER and SYSEXIT Instructions.</b> The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	<b>Memory Type Range Registers.</b> MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	<b>Page Global Bit.</b> The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	<b>Machine Check Architecture.</b> A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	<b>Conditional Move Instructions.</b> The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	<b>Page Attribute Table.</b> Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	<b>36-Bit Page Size Extension.</b> 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	<b>Processor Serial Number.</b> The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	<b>CLFLUSH Instruction.</b> CLFLUSH Instruction is supported.
20	Reserved	Reserved

**Table 3-11. More on Feature Information Returned in the EDX Register (Contd.)**

Bit #	Mnemonic	Description
21	DS	<b>Debug Store.</b> The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and processor event-based sampling (PEBS) facilities (see Chapter 22, "Introduction to Virtual-Machine Extensions," in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C</i> ).
22	ACPI	<b>Thermal Monitor and Software Controlled Clock Facilities.</b> The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	<b>Intel MMX Technology.</b> The processor supports the Intel MMX technology.
24	FXSR	<b>FXSAVE and FXRSTOR Instructions.</b> The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	<b>SSE.</b> The processor supports the SSE extensions.
26	SSE2	<b>SSE2.</b> The processor supports the SSE2 extensions.
27	SS	<b>Self Snoop.</b> The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	<b>Max APIC IDs reserved field is Valid.</b> A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	<b>Thermal Monitor.</b> The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	<b>Pending Break Enable.</b> The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt.

**INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX**

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

Table 3-12. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages, 128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages, 256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
70H	Cache	Trace cache: 12 K- $\mu$ op, 8-way set associative
71H	Cache	Trace cache: 16 K- $\mu$ op, 8-way set associative
72H	Cache	Trace cache: 32 K- $\mu$ op, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size



**Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)**

Value	Type	Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GByte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FEH	General	CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters



**Example 3-1. Example of Cache and TLB Interpretation**

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```
EAX    66 5B 50 01H
EBX    0H
ECX    0H
EDX    00 7A 70 00H
```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K- $\mu$ op, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

**INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level**

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

$$= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line\_Size} + 1) * (\text{Sets} + 1)$$

$$= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**INPUT EAX = 05H: Returns MONITOR and MWAIT Features**

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

**INPUT EAX = 06H: Returns Thermal and Power Management Features**

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.

**INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information**

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

**INPUT EAX = 09H: Returns Direct Cache Access Information**

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

**INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features**

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 22, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

**INPUT EAX = 0BH: Returns Extended Topology Information**

*CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.*

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is  $\geq 0BH$ , and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

**INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information**

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n ( $n > 1$ , and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

For i = 2 to 62 // sub-leaf 1 is reserved

IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1) // VECTOR is the 64-bit value of EDX:EAX

Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;

FI;

**INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information**

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n ( $n \geq 1$ , and is a valid ResID), the processor returns information software can use to program IA32\_PQR\_ASSOC, IA32\_QM\_EVTSEL MSRs before reading QoS data from the IA32\_QM\_CTR MSR.

**INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information**

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32\_resourceType\_Mask\_n.

**INPUT EAX = 12H: Returns Intel SGX Enumeration Information**

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

**INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information**

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

**INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information**

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

**INPUT EAX = 16H: Returns Processor Frequency Information**

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

**INPUT EAX = 17H: Returns System-On-Chip Information**

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

**INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information**

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

**INPUT EAX = 19H: Returns Key Locker Information**

When CPUID executes with EAX set to 19H, the processor returns information about Key Locker. See Table 3-8.

**INPUT EAX = 1AH: Returns Hybrid Information**

When CPUID executes with EAX set to 1AH, the processor returns information about hybrid capabilities. See Table 3-8.

**INPUT EAX = 1BH: Returns PCONFIG Information**

When CPUID executes with EAX set to 1BH, the processor returns information about PCONFIG capabilities. This information is enumerated in sub-leaves selected by the value of ECX (starting with 0).

Each sub-leaf of CPUID function 1BH enumerates its **sub-leaf type** in EAX. If a sub-leaf type is 0, the sub-leaf is invalid and zero is returned in EBX, ECX, and EDX. In this case, all subsequent sub-leaves (selected by larger input values of ECX) are also invalid.

The only valid sub-leaf type currently defined is 1, meaning **target identifier**, indicating that the sub-leaf enumerates target identifiers for the PCONFIG instruction. Any non-zero value returned in EBX, ECX, or EDX indicates a valid target of the PCONFIG instruction (any value of zero should be ignored). The only target identifier currently defined is 1, indicating MKTME. See the “PCONFIG — Platform Configuration” instruction in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B* for more information.

### INPUT EAX = 1FH: Returns V2 Extended Topology Information

When CPUID executes with EAX set to 1FH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 1FH by verifying (a) the highest leaf index supported by CPUID is  $\geq$  1FH, and (b) CPUID.1FH:EBX[15:0] reports a non-zero value. See Table 3-8.

### METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

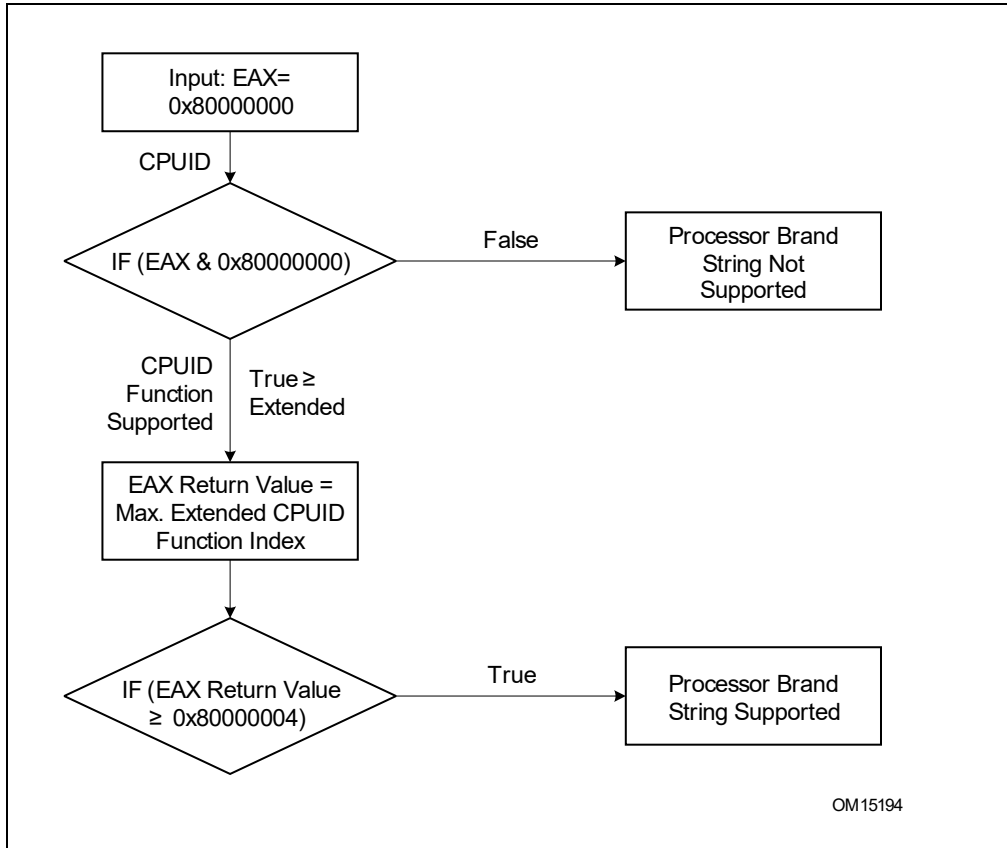
1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: “Identification of Earlier IA-32 Processors” in Chapter 20 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

#### The Processor Brand String Method

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.



**Figure 3-9. Determination of Support for the Processor Brand String**

**How Brand Strings Work**

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

**Table 3-13. Processor Brand String Returned with Pentium 4 Processor**

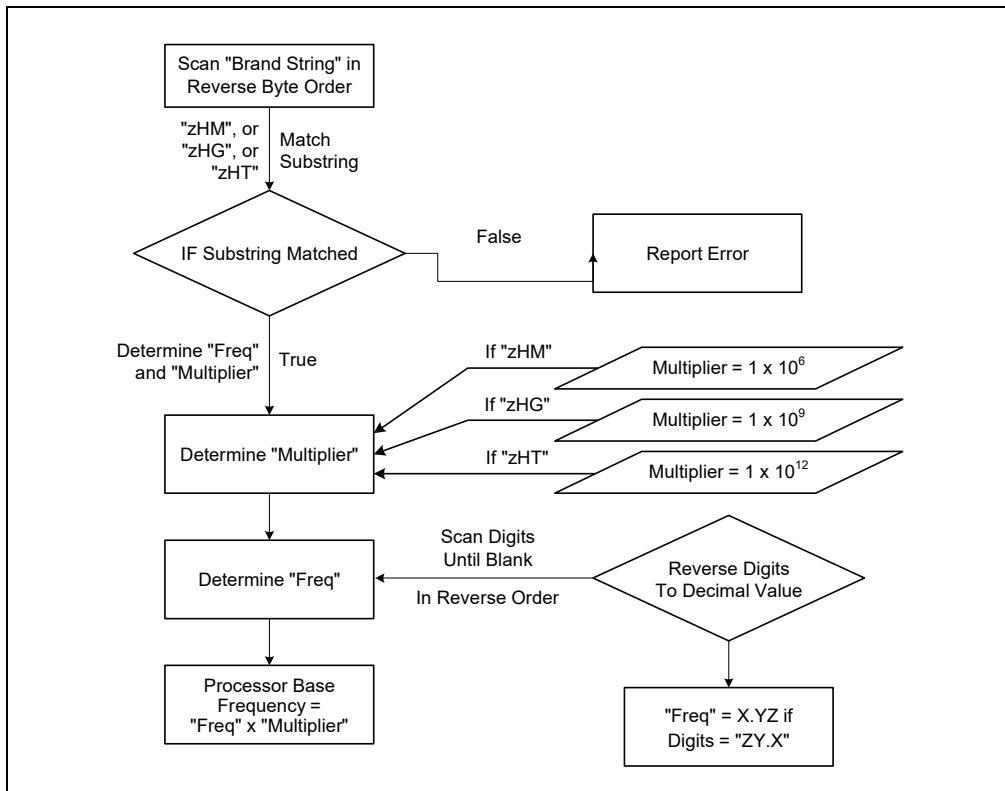
EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	" " " " " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P )R" "itne" "R(mu"

**Table 3-13. Processor Brand String Returned with Pentium 4 Processor (Contd.)**

EAX Input Value	Return Values	ASCII Equivalent
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4 )" " UPC" "0051" "\0zHM"

### Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.



**Figure 3-10. Algorithm for Extracting Processor Frequency**

### The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-14 shows brand indices that have identification strings associated with them.

**Table 3-14. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings**

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor <sup>1</sup>
02H	Intel(R) Pentium(R) III processor <sup>1</sup>
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor <sup>1</sup>
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
18H – 0FFH	RESERVED

## NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

**IA-32 Architecture Compatibility**

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

**Operation**

IA32\_BIOS\_SIGN\_ID MSR := Update with installed microcode revision number;

CASE (EAX) OF

EAX = 0:

EAX := Highest basic function input value understood by CPUID;

EBX := Vendor identification string;

EDX := Vendor identification string;

ECX := Vendor identification string;

BREAK;

EAX = 1H:

EAX[3:0] := Stepping ID;

EAX[7:4] := Model;

EAX[11:8] := Family;

EAX[13:12] := Processor type;  
 EAX[15:14] := Reserved;  
 EAX[19:16] := Extended Model;  
 EAX[27:20] := Extended Family;  
 EAX[31:28] := Reserved;  
 EBX[7:0] := Brand Index; (\* Reserved if the value is zero. \*)  
 EBX[15:8] := CLFLUSH Line Size;  
 EBX[16:23] := Reserved; (\* Number of threads enabled = 2 if MT enable fuse set. \*)  
 EBX[24:31] := Initial APIC ID;  
 ECX := Feature flags; (\* See Figure 3-7. \*)  
 EDX := Feature flags; (\* See Figure 3-8. \*)

BREAK;

EAX = 2H:

EAX := Cache and TLB information;  
 EBX := Cache and TLB information;  
 ECX := Cache and TLB information;  
 EDX := Cache and TLB information;

BREAK;

EAX = 3H:

EAX := Reserved;  
 EBX := Reserved;  
 ECX := ProcessorSerialNumber[31:0];  
 (\* Pentium III processors only, otherwise reserved. \*)  
 EDX := ProcessorSerialNumber[63:32];  
 (\* Pentium III processors only, otherwise reserved. \*)

BREAK

EAX = 4H:

EAX := Deterministic Cache Parameters Leaf; (\* See Table 3-8. \*)  
 EBX := Deterministic Cache Parameters Leaf;  
 ECX := Deterministic Cache Parameters Leaf;  
 EDX := Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:

EAX := MONITOR/MWAIT Leaf; (\* See Table 3-8. \*)  
 EBX := MONITOR/MWAIT Leaf;  
 ECX := MONITOR/MWAIT Leaf;  
 EDX := MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:

EAX := Thermal and Power Management Leaf; (\* See Table 3-8. \*)  
 EBX := Thermal and Power Management Leaf;  
 ECX := Thermal and Power Management Leaf;  
 EDX := Thermal and Power Management Leaf;

BREAK;

EAX = 7H:

EAX := Structured Extended Feature Flags Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX := Structured Extended Feature Flags Enumeration Leaf;  
 ECX := Structured Extended Feature Flags Enumeration Leaf;  
 EDX := Structured Extended Feature Flags Enumeration Leaf;

BREAK;

EAX = 8H:

EAX := Reserved = 0;  
 EBX := Reserved = 0;  
 ECX := Reserved = 0;



```

    EDX := Reserved = 0;
BREAK;
EAX = 9H:
    EAX := Direct Cache Access Information Leaf; (* See Table 3-8. *)
    EBX := Direct Cache Access Information Leaf;
    ECX := Direct Cache Access Information Leaf;
    EDX := Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX := Architectural Performance Monitoring Leaf; (* See Table 3-8. *)
    EBX := Architectural Performance Monitoring Leaf;
    ECX := Architectural Performance Monitoring Leaf;
    EDX := Architectural Performance Monitoring Leaf;
BREAK;
EAX = BH:
    EAX := Extended Topology Enumeration Leaf; (* See Table 3-8. *)
    EBX := Extended Topology Enumeration Leaf;
    ECX := Extended Topology Enumeration Leaf;
    EDX := Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX := Reserved = 0;
    EBX := Reserved = 0;
    ECX := Reserved = 0;
    EDX := Reserved = 0;
BREAK;
EAX = DH:
    EAX := Processor Extended State Enumeration Leaf; (* See Table 3-8. *)
    EBX := Processor Extended State Enumeration Leaf;
    ECX := Processor Extended State Enumeration Leaf;
    EDX := Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
    EAX := Reserved = 0;
    EBX := Reserved = 0;
    ECX := Reserved = 0;
    EDX := Reserved = 0;
BREAK;
EAX = FH:
    EAX := Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    ECX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    EDX := Intel Resource Director Technology Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
    EAX := Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel Resource Director Technology Allocation Enumeration Leaf;
    ECX := Intel Resource Director Technology Allocation Enumeration Leaf;
    EDX := Intel Resource Director Technology Allocation Enumeration Leaf;
BREAK;
EAX = 12H:
    EAX := Intel SGX Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel SGX Enumeration Leaf;
    ECX := Intel SGX Enumeration Leaf;

```

EDX := Intel SGX Enumeration Leaf;  
BREAK;  
EAX = 14H:  
EAX := Intel Processor Trace Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := Intel Processor Trace Enumeration Leaf;  
ECX := Intel Processor Trace Enumeration Leaf;  
EDX := Intel Processor Trace Enumeration Leaf;  
BREAK;  
EAX = 15H:  
EAX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (\* See Table 3-8. \*)  
EBX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;  
ECX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;  
EDX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;  
BREAK;  
EAX = 16H:  
EAX := Processor Frequency Information Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := Processor Frequency Information Enumeration Leaf;  
ECX := Processor Frequency Information Enumeration Leaf;  
EDX := Processor Frequency Information Enumeration Leaf;  
BREAK;  
EAX = 17H:  
EAX := System-On-Chip Vendor Attribute Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := System-On-Chip Vendor Attribute Enumeration Leaf;  
ECX := System-On-Chip Vendor Attribute Enumeration Leaf;  
EDX := System-On-Chip Vendor Attribute Enumeration Leaf;  
BREAK;  
EAX = 18H:  
EAX := Deterministic Address Translation Parameters Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := Deterministic Address Translation Parameters Enumeration Leaf;  
ECX := Deterministic Address Translation Parameters Enumeration Leaf;  
EDX := Deterministic Address Translation Parameters Enumeration Leaf;  
BREAK;  
EAX = 19H:  
EAX := Key Locker Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := Key Locker Enumeration Leaf;  
ECX := Key Locker Enumeration Leaf;  
EDX := Key Locker Enumeration Leaf;  
BREAK;  
EAX = 1AH:  
EAX := Hybrid Information Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := Hybrid Information Enumeration Leaf;  
ECX := Hybrid Information Enumeration Leaf;  
EDX := Hybrid Information Enumeration Leaf;  
BREAK;  
EAX = 1BH:  
EAX := PCONFIG Information Enumeration Leaf; (\* See "INPUT EAX = 1BH: Returns PCONFIG Information" on page 3-246. \*)  
EBX := PCONFIG Information Enumeration Leaf;  
ECX := PCONFIG Information Enumeration Leaf;  
EDX := PCONFIG Information Enumeration Leaf;  
BREAK;  
EAX = 1FH:  
EAX := V2 Extended Topology Enumeration Leaf; (\* See Table 3-8. \*)  
EBX := V2 Extended Topology Enumeration Leaf;  
ECX := V2 Extended Topology Enumeration Leaf;

EDX := V2 Extended Topology Enumeration Leaf;  
 BREAK;  
 EAX = 80000000H:  
 EAX := Highest extended function input value understood by CPUID;  
 EBX := Reserved;  
 ECX := Reserved;  
 EDX := Reserved;  
 BREAK;  
 EAX = 80000001H:  
 EAX := Reserved;  
 EBX := Reserved;  
 ECX := Extended Feature Bits (\* See Table 3-8.\*);  
 EDX := Extended Feature Bits (\* See Table 3-8.\*);  
 BREAK;  
 EAX = 80000002H:  
 EAX := Processor Brand String;  
 EBX := Processor Brand String, continued;  
 ECX := Processor Brand String, continued;  
 EDX := Processor Brand String, continued;  
 BREAK;  
 EAX = 80000003H:  
 EAX := Processor Brand String, continued;  
 EBX := Processor Brand String, continued;  
 ECX := Processor Brand String, continued;  
 EDX := Processor Brand String, continued;  
 BREAK;  
 EAX = 80000004H:  
 EAX := Processor Brand String, continued;  
 EBX := Processor Brand String, continued;  
 ECX := Processor Brand String, continued;  
 EDX := Processor Brand String, continued;  
 BREAK;  
 EAX = 80000005H:  
 EAX := Reserved = 0;  
 EBX := Reserved = 0;  
 ECX := Reserved = 0;  
 EDX := Reserved = 0;  
 BREAK;  
 EAX = 80000006H:  
 EAX := Reserved = 0;  
 EBX := Reserved = 0;  
 ECX := Cache information;  
 EDX := Reserved = 0;  
 BREAK;  
 EAX = 80000007H:  
 EAX := Reserved = 0;  
 EBX := Reserved = 0;  
 ECX := Reserved = 0;  
 EDX := Reserved = Misc Feature Flags;  
 BREAK;  
 EAX = 80000008H:  
 EAX := Reserved = Physical Address Size Information;  
 EBX := Reserved = Virtual Address Size Information;  
 ECX := Reserved = 0;

EDX := Reserved = 0;  
 BREAK;  
 EAX >= 40000000H and EAX <= 4FFFFFFFH:  
 DEFAULT: (\* EAX = Value outside of recognized range for CPUID. \*)  
 (\* If the highest basic information leaf data depend on ECX input value, ECX is honored. \*)  
 EAX := Reserved; (\* Information returned for highest basic information leaf. \*)  
 EBX := Reserved; (\* Information returned for highest basic information leaf. \*)  
 ECX := Reserved; (\* Information returned for highest basic information leaf. \*)  
 EDX := Reserved; (\* Information returned for highest basic information leaf. \*)  
 BREAK;  
 ESAC;

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD                    If the LOCK prefix is used.  
                          In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.

## CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 2A /r CVTPI2PS <i>xmm, mm/m64</i>	RM	Valid	Valid	Convert two signed doubleword integers from <i>mm/m64</i> to two single-precision floating-point values in <i>xmm</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r, w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA

### Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
CVTPI2PS:  __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)
```

### SIMD Floating-Point Exceptions

Precision

### Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1, xmm2, xmm3/m64	B	V/V	AVX	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.
EVEX.LLIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Converts a double-precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand.

When the “convert-from” operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VCVTSD2SS (EVEX encoded version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

### VCVTSD2SS (VEX.128 encoded version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

### CVTSD2SS (128-bit Legacy SSE version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSD2SS __m128 __mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 __mm_maskz_cvtsd_ss(__mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 __mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 __mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 __mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int r);
CVTSD2SS __m128 __mm_cvtsd_ss(__m128 a, __m128d b)

```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

## Other Exceptions

VEX-encoded instructions, see [Table 2-20, “Type 3 Class Exception Conditions”](#).

EVEX-encoded instructions, see [Table 2-47, “Type E3 Class Exception Conditions”](#).

## CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.
EVEX.LLIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Converts a single-precision floating-point value in the “convert-from” source operand to a double-precision floating-point value in the destination operand. When the “convert-from” source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### VCVTSS2SD (EVEX encoded version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0] := Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] = 0

FI;

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0



**VCVTSS2SD (VEX.128 encoded version)**

DEST[63:0] := Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0])

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0

**CVTSS2SD (128-bit Legacy SSE version)**

DEST[63:0] := Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);

DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SD \_\_m128d \_\_mm\_cvt\_roundss\_sd(\_\_m128d a, \_\_m128 b, int r);

VCVTSS2SD \_\_m128d \_\_mm\_mask\_cvt\_roundss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b, int r);

VCVTSS2SD \_\_m128d \_\_mm\_maskz\_cvt\_roundss\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128 a, int r);

VCVTSS2SD \_\_m128d \_\_mm\_mask\_cvtss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b);

VCVTSS2SD \_\_m128d \_\_mm\_maskz\_cvtss\_sd(\_\_mmask8 m, \_\_m128d a, \_\_m128 b);

CVTSS2SD \_\_m128d \_\_mm\_cvtss\_sd(\_\_m128d a, \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see [Table 2-20, “Type 3 Class Exception Conditions”](#).

EVEX-encoded instructions, see [Table 2-47, “Type E3 Class Exception Conditions”](#).

## INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CC	INT3	Z0	Valid	Valid	Generate breakpoint trap.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Generate software interrupt with vector specified by immediate byte.
CE	INTO	Z0	Invalid	Valid	Generate overflow trap if overflow flag is 1.
F1	INT1	Z0	Valid	Valid	Generate debug trap.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
I	imm8	NA	NA	NA

### Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6.<sup>1</sup> Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The action of the INT *n* instruction (including the INTO, INT3, and INT1 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

Each of the INT *n*, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.

even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hardware vendors performing hardware debug.)

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the “Operation” section for this instruction (except #GP).

**Table 3-52. Decision Table**

PE	0	1	1	1	1	1	1	1
VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	=3
DPL/CPL RELATIONSHIP	-	DPL < CPL	-	DPL > CPL	DPL = CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

**NOTES:**

- Don't Care.
- Y Yes, action taken.
- Blank Action not taken.
- S/W Applies to INT n, INT3, and INTO, but not to INT1.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for CET details.

**Instruction ordering.** Instructions following an INT  $n$  may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the INT  $n$  have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible). This applies also to the INTO, INT3, and INT1 instructions, but not to executions of INTO when EFLAGS.OF = 0.

## Operation

The following operational description applies not only to the INT  $n$ , INTO, INT3, or INT1 instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction `error_code(num, idt, ext)`, where `idt` and `ext` are bit values. The pseudofunction produces an error code as follows: (1) if `idt` is 0, the error code is  $(\text{num} \& \text{FCH}) \mid \text{ext}$ ; (2) if `idt` is 1, the error code is  $(\text{num} \ll 3) \mid 2 \mid \text{ext}$ .

In many cases, the pseudofunction `error_code` is invoked with a pseudovariable `EXT`. The value of `EXT` depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt (INT  $n$ , INT3, or INTO), `EXT` is 0; otherwise (including INT1), `EXT` is 1.

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT  $n$ )
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT  $n$  *)
      ELSE
        IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT  $n$ )
          THEN
            Consult bit  $n$  of the software interrupt redirection bit map in the TSS;
            IF bit  $n$  is clear
              THEN (* redirect interrupt to 8086 program interrupt handler *)
                Push EFLAGS[15:0]; (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                Push CS;
                Push IP;
                IF IOPL = 3
                  THEN IF := 0; (* Clear interrupt flag *)
                  ELSE VIF := 0; (* Clear virtual interrupt flag *)
                FI;
                TF := 0; (* Clear trap flag *)
                load CS and EIP (lower 16 bits only) from entry  $n$  in interrupt vector table referenced from TSS;
              ELSE
                IF IOPL = 3
                  THEN GOTO PROTECTED-MODE;
                  ELSE #GP(0); (* Bit 0 of error code is 0 because INT  $n$  *)
                FI;
            FI;
          ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
            IF (IA32_EFER.LMA = 0)
              THEN (* Protected mode, or virtual-8086 mode interrupt *)
                GOTO PROTECTED-MODE;
              ELSE (* IA-32e mode interrupt *)
                GOTO IA-32e-MODE;
            FI;
          FI;
        FI;
      FI;
    FI;
  FI;

```

```

        FI;
    FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number << 2) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF := 0; (* Clear interrupt flag *)
    TF := 0; (* Clear trap flag *)
    AC := 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS := IDT(Descriptor (vector_number << 2), selector);
    EIP := IDT(Descriptor (vector_number << 2), offset); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number << 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0)); FI;
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
            FI;
            IF gate not present
                THEN #NP(error_code(vector_number,1,EXT)); FI;
                (* idt operand to error_code set because vector is used *)
            IF task gate (* Specified in the selected interrupt table descriptor *)
                THEN GOTO TASK-GATE;
            ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
        FI;
END;
IA-32e-MODE:
    IF INTO and CS.L = 1 (64-bit mode)
        THEN #UD;
    FI;
    IF ((vector_number << 4) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0));
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
            FI;
        FI;

```

```

        FI;
FI;
IF gate not present
    THEN #NP(error_code(vector_number,1,EXT));
    (* idt operand to error_code set because vector is used *)
FI;
GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
    IF local/global bit is set to local or index not within GDT limits
        THEN #GP(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(EXT); FI;
            Push(error code);
FI;
IF EIP not within code segment limit
    THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                    DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL ≠ 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));

```

```

        (* idt operand to error_code is 0 because selector is used *)
        GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
        (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
    FI;
ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
    IF VM = 1
        THEN #GP(error_code(new code-segment selector,0,EXT));
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is conforming or new code-segment DPL = CPL
        THEN
            GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
        ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
            #GP(error_code(new code-segment selector,0,EXT));
            (* idt operand to error_code is 0 because selector is used *)
        FI;
    FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        (* Identify stack-segment selector for new privilege level in current TSS *)
        IF current TSS is 32-bit
            THEN
                TSSstackAddress := (new code-segment DPL << 3) + 4;
                IF (TSSstackAddress + 5) > current TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                NewESP := 4 bytes loaded from (TSS base + TSSstackAddress);
            ELSE (* current TSS is 16-bit *)
                TSSstackAddress := (new code-segment DPL << 2) + 2
                IF (TSSstackAddress + 3) > current TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                NewESP := 2 bytes loaded from (TSS base + TSSstackAddress);
            FI;
        IF NewSS is NULL
            THEN #TS(EXT); FI;
        IF NewSS index is not within its descriptor-table limits
        or NewSS RPL ≠ new code-segment DPL
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Read new stack-segment descriptor for NewSS in GDT or LDT;
        IF new stack-segment DPL ≠ new code-segment DPL
        or new stack-segment Type does not indicate writable data segment
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF NewSS is not present
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
            NewSSP := IA32_Pli_SSP (* where i = new code-segment DPL *)
        ELSE (* IA-32e mode *)

```

```

IF IDT-gate IST = 0
    THEN TSSstackAddress := (new code-segment DPL << 3) + 4;
    ELSE TSSstackAddress := (IDT gate IST << 3) + 28;
FI;
IF (TSSstackAddress + 7) > current TSS limit
    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
IF IDT-gate IST = 0
    THEN
        NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
    ELSE
        NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT-gate IST << 3)
        (* Check if shadow stacks are enabled at CPL 0 *)
        IF ShadowStackEnabled(CPL 0)
            THEN NewSSP := 8 bytes loaded from NewSSPAddress; FI;
    FI;
FI;
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 24 bytes (error code pushed)
        or 20 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        FI
    ELSE
        IF IDT gate is 16-bit
            THEN
                IF new stack does not have room for 12 bytes (error code pushed)
                or 10 bytes (no error code pushed);
                    THEN #SS(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                ELSE (* 64-bit IDT gate*)
                    IF StackAddress is non-canonical
                        THEN #SS(EXT); FI; (* Error code contains NULL selector *)
                FI;
            FI;
        IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
            THEN
                IF instruction pointer from IDT gate is not within new code-segment limits
                    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                ESP := NewESP;
                SS := NewSS; (* Segment descriptor information also loaded *)
            ELSE (* IA-32e mode *)
                IF instruction pointer from IDT gate contains a non-canonical address
                    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                RSP := NewRSP & FFFFFFFF0H;
                SS := NewSS;
            FI;
        IF IDT gate is 32-bit
            THEN
                CS:EIP := Gate(CS:EIP); (* Segment descriptor information also loaded *)
            ELSE

```



```

    IF IDT gate 16-bit
    THEN
        CS:IP := Gate(CS:IP);
        (* Segment descriptor information also loaded *)
    ELSE (* 64-bit IDT gate *)
        CS:RIP := Gate(CS:RIP);
        (* Segment descriptor information also loaded *)
    FI;
FI;
IF IDT gate is 32-bit
THEN
    Push(far pointer to old stack);
    (* Old SS and ESP, 3 words padded to 4 *)
    Push(EFLAGS);
    Push(far pointer to return instruction);
    (* Old CS and EIP, 3 words padded to 4 *)
    Push(ErrorCode); (* If needed, 4 bytes *)
ELSE
    IF IDT gate 16-bit
    THEN
        Push(far pointer to old stack);
        (* Old SS and SP, 2 words *)
        Push(EFLAGS(15:0));
        Push(far pointer to return instruction);
        (* Old CS and IP, 2 words *)
        Push(ErrorCode); (* If needed, 2 bytes *)
    ELSE (* 64-bit IDT gate *)
        Push(far pointer to old stack);
        (* Old SS and SP, each an 8-byte push *)
        Push(RFLAGS); (* 8-byte push *)
        Push(far pointer to return instruction);
        (* Old CS and RIP, each an 8-byte push *)
        Push(ErrorCode); (* If needed, 8-bytes *)
    FI;
FI;
IF ShadowStackEnabled(CPL) AND CPL = 3
THEN
    IF IA32_EFER.LMA = 0
    THEN IA32_PL3_SSP := SSP;
    ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
        IA32_PL3_SSP := LA_adjust(SSP);
    FI;
FI;
CPL := new code-segment DPL;
CS(RPL) := CPL;
IF ShadowStackEnabled(CPL)
oldSSP := SSP
SSP := NewSSP
IF SSP & 0x07 != 0
    THEN #GP(0); FI;
(* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
    #GP(0); FI;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)

```

```

    THEN #GP(0); FI;
    expected_token_value = SSP          (* busy bit - bit position 0 - must be clear *)
    new_token_value = SSP | BUSY_BIT    (* Set the busy bit *)
    IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
        THEN #GP(0); FI;

    IF oldSS.DPL != 3
        (* These stack pushes should not cause faults, VM exits, data breakpoints *)
        (* Such events will apply to the earlier accesses to the token, which is in the same 32-byte region *)
        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit RIP *)
        ShadowStackPush8B(oldSSP);
    FI;
FI;
IF EndbranchEnabled (CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
    IA32_S_CET.SUPPRESS = 0
FI;
IF IDT gate is interrupt gate
    THEN IF := 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
TF := 0;
VM := 0;
RF := 0;
NT := 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
(* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
    THEN
        IF TSS limit < 9
            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
            NewSS := 2 bytes loaded from (current TSS base + 8);
            NewESP := 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
                NewSS := 2 bytes loaded from (current TSS base + 4);
                NewESP := 2 bytes loaded from (current TSS base + 2);
            FI;
        IF NewSS is NULL
            THEN #TS(EXT); FI; (* Error code contains NULL selector *)
        IF NewSS index is not within its descriptor table limits
        or NewSS RPL ≠ 0
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Read new stack-segment descriptor for NewSS in GDT or LDT;
        IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF new stack segment not present
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)

```

```

NewSSP := IA32_Pli_SSP (* where i = new code-segment DPL *)
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 40 bytes (error code pushed)
        or 36 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        ELSE (* IDT gate is 16-bit *)
            IF new stack does not have room for 20 bytes (error code pushed)
            or 18 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
        FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
tempEFLAGS := EFLAGS;
VM := 0;
TF := 0;
RF := 0;
NT := 0;
IF service through interrupt gate
    THEN IF = 0; FI;
TempSS := SS;
TempESP := ESP;
SS := NewSS;
ESP := NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
Push(TempEFlags);
Push(CS);
Push(EIP);
GS := 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS := 0;
DS := 0;
ES := 0;
CS := Gate(CS); (* Segment descriptor information also loaded *)
CS(RPL) := 0;
CPL := 0;
IF IDT gate is 32-bit
    THEN
        EIP := Gate(instruction pointer);
    ELSE (* IDT gate is 16-bit *)
        EIP := Gate(instruction pointer) AND 0000FFFFH;
    FI;
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := NewSSP
    IF SSP & 0x07 != 0

```

```

        THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
        IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
            #GP(0); FI;
    IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
        THEN #GP(0); FI;
    expected_token_value = SSP (* busy bit - bit position 0 - must be clear *)
    new_token_value = SSP | BUSY_BIT (* Set the busy bit *)
    IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
        THEN #GP(0); FI;
        IF oldSS.DPL != 3
            (* These stack pushes should not cause faults, VM exits, or data breakpoints *)
            (* Such events will apply to the earlier accesses to the token, which is in the same 32-byte region *)
            ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
            ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
            ShadowStackPush8B(oldSSP);
        FI;
    FI;
    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
        IA32_S_CET.SUPPRESS = 0
    FI;
(* Start execution of new routine in Protected Mode *)
END;

```

#### INTRA-PRIVILEGE-LEVEL-INTERRUPT:

```

    NewSSP = SSP;
    CHECK_SS_TOKEN = 0
    (* PE = 1, DPL = CPL or conforming segment *)
    IF IA32_EFER.LMA = 1 (* IA-32e mode *)
        IF IDT-descriptor IST ≠ 0
            THEN
                TSSstackAddress := (IDT-descriptor IST << 3) + 28;
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                    NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
                ELSE NewRSP := RSP;
            FI;
        IF ShadowStackEnabled(CPL)
            THEN
                NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT_gate IST << 3)
                NewSSP := 8 bytes loaded from NewSSPAddress
                CHECK_SS_TOKEN = 1
            FI;
        FI;
    IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
                or 12 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
            ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
                IF current stack does not have room for 8 bytes (error code pushed)
                    or 6 bytes (no error code pushed)

```

```

        THEN #SS(EXT); FI; (* Error code contains NULL selector *)
ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
    IF NewRSP contains a non-canonical address
        THEN #SS(EXT); (* Error code contains NULL selector *)
FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limit
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP := NewRSP & FFFFFFFF00000000H;
FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
    THEN
        Push (EFLAGS);
        Push (far pointer to return instruction); (* 3 words padded to 4 *)
        CS:EIP := Gate(CS:EIP); (* Segment descriptor information also loaded *)
        Push (ErrorCode); (* If any *)
    ELSE
        IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
            THEN
                Push (FLAGS);
                Push (far pointer to return location); (* 2 words *)
                CS:IP := Gate(CS:IP);
                (* Segment descriptor information also loaded *)
                Push (ErrorCode); (* If any *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                Push(far pointer to old stack);
                (* Old SS and SP, each an 8-byte push *)
                Push(RFLAGS); (* 8-byte push *)
                Push(far pointer to return instruction);
                (* Old CS and RIP, each an 8-byte push *)
                Push(ErrorCode); (* If needed, 8 bytes *)
                CS:RIP := GATE(CS:RIP);
                (* Segment descriptor information also loaded *)
FI;
FI;
CS(RPL) := CPL;
IF ShadowStackEnabled(CPL)
    IF CHECK_SS_TOKEN == 1
        THEN
            IF NewSSP & 0x07 != 0
                THEN #GP(0); FI;
            (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
            IF (NewSSP & ~0x1F) != ((NewSSP - 24) & ~0x1F)
                #GP(0); FI;

            IF ((IA32_EFER.LMA and CS.L) = 0 AND NewSSP[63:32] != 0)
                THEN #GP(0); FI;
            expected_token_value = NewSSP (* busy bit - bit position 0 - must be clear *)
            new_token_value = NewSSP | BUSY_BIT (* Set the busy bit *)

```

```

        IF shadow_stack_lock_cmpxchg8b(NewSSP, new_token_value, expected_token_value) != expected_token_value
            THEN #GP(0); FI;
FI;
(* Align to next 8 byte boundary *)
tempSSP = SSP;
Shadow_stack_store 4 bytes of 0 to (NewSSP - 4)
SSP = newSSP & 0xFFFFFFFFFFFFFFF8H;
(* These stack pushes should not cause faults, VM exits, or data breakpoints, if CHECK_SS_TOKEN is 1 *)
(* Such events will apply to the earlier accesses to the token, which is in the same 32-byte region *)
(* push cs:lip:ssp on shadow stack *)
ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
IF IDT gate is interrupt gate
    THEN IF := 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
TF := 0;
NT := 0;
VM := 0;
RF := 0;
END;

```

### Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the “Operation” section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task’s TSS.

### Protected Mode Exceptions

#GP(error\_code) If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.

If the segment selector in the interrupt, trap, or task gate is NULL.

If an interrupt, trap, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.

If the vector selects a descriptor outside the IDT limits.

If an IDT descriptor is not an interrupt, trap, or task gate.

If an interrupt is generated by the INT *n*, INT3, or INTO instruction and the DPL of an interrupt, trap, or task gate is less than the CPL.

If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

If SSP in IA32\_PLi\_SSP (where *i* is the new CPL) is not 8 byte aligned.

If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.

If “supervisor Shadow Stack” token on new shadow stack is marked busy.

If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.

If SSP address in “supervisor shadow stack” token does not match SSP address in IA32\_PLi\_SSP (where i is the new CPL).

#SS(error_code)	<p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p> <p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.</p>
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	<p>If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.</p> <p>If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.</p> <p>If the stack segment selector in the TSS is NULL.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

### Real-Address Mode Exceptions

#GP	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the interrupt vector number is outside the IDT limits.</p>
#SS	<p>If stack limit violation on push.</p> <p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.</p>
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(error_code)	<p>(For INT <i>n</i>, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate is not equal to 3.</p> <p>If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt, trap, or task gate is NULL.</p> <p>If a interrupt gate, trap gate, task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt, trap, or task gate.</p> <p>If an interrupt is generated by INT <i>n</i>, INT3, or INTO and the DPL of an interrupt, trap, or task gate is less than the CPL.</p> <p>If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p>
#SS(error_code)	<p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.</p>

#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. If the stack segment selector in the TSS is NULL. If the stack segment for the TSS is not a writable data segment. If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical. If the segment selector in the 64-bit interrupt or trap gate is NULL. If the vector selects a descriptor outside the IDT limits. If the vector points to a gate which is in non-canonical space. If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate. If the descriptor pointed to by the gate selector is outside the descriptor table limit. If the descriptor pointed to by the gate selector is in non-canonical space. If the descriptor pointed to by the gate selector is not a code segment. If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set. If the descriptor pointed to by the gate selector has DPL > CPL. If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned. If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack. If "supervisor shadow stack" token on new shadow stack is marked busy. If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4GB. If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#SS(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch. If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	If an attempt to load RSP from the TSS causes an access to non-canonical space. If the RSP from the TSS is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.



## IRET/IRETD/IRETQ—Interrupt Return

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CF	IRET	Z0	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	Z0	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	Z0	Valid	N.E.	Interrupt return (64-bit operand size).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction performs a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, “Handling Multiple NMIs” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), execution of the IRET instruction unblocks NMIs.

This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for CET details.

**Instruction ordering.** IRET is a serializing instruction. See Section 8.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 24 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```

IF PE = 0
  THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
  THEN
    IF (EFLAGS.VM = 1)
      THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
      ELSE GOTO PROTECTED-MODE;
    FI;
  ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE;
  IF OperandSize = 32
    THEN
      EIP := Pop();
      CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
      tempEFLAGS := Pop();
      EFLAGS := (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
    ELSE (* OperandSize = 16 *)
      EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
      CS := Pop(); (* 16-bit pop *)
      EFLAGS[15:0] := Pop();
    FI;
  END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
  IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
    THEN IF OperandSize = 32
      THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        EFLAGS := Pop();
        (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
        IF EIP not within CS limit
          THEN #GP(0); FI;
        ELSE (* OperandSize = 16 *)
          EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
          CS := Pop(); (* 16-bit pop *)
          EFLAGS[15:0] := Pop(); (* IOPL in EFLAGS not modified by pop *)
        FI;
      FI;
    FI;
  
```

```

        IF EIP not within CS limit
            THEN #GP(0); FI;
        FI;
    ELSE
        #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS := Pop();
        ELSE (* OperandSize = 16 *)
            EIP := Pop(); (* 16-bit pop; clear upper bits *)
            CS := Pop(); (* 16-bit pop *)
            tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
        FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
    FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    (* If shadow stack or indirect branch tracking at CPL3 then #GP(0) *)
    IF CR4.CET AND (IA32_U_CET.ENDBR_EN OR IA32_U_CET.SHSTK_EN)
        THEN #GP(0); FI;
    shadowStackEnabled = ShadowStackEnabled(CPL)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS := tempEFLAGS;
    ESP := Pop();
    SS := Pop(); (* Pop 2 words; throw away high-order word *)
    ES := Pop(); (* Pop 2 words; throw away high-order word *)
    DS := Pop(); (* Pop 2 words; throw away high-order word *)
    FS := Pop(); (* Pop 2 words; throw away high-order word *)
    GS := Pop(); (* Pop 2 words; throw away high-order word *)
    IF shadowStackEnabled
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;

```

```

CPL := 3;
(* Resume execution in Virtual-8086 mode *)
tempOldSSP = SSP;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF shadowStackEnabled
    expected_token_value = tempOldSSP | BUSY_BIT    (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                    (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
END;

```

```

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

```

```

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
            tempESP := Pop();
            tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE IF OperandSize = 16
            THEN
                tempESP := Pop(); (* 16-bit pop; clear upper bits *)
                tempSS := Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                tempRSP := Pop();
                tempSS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
            FI;
        IF new mode ≠ 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
                ELSE (* new mode = 64-bit mode *)
                    IF RIP is non-canonical
                        THEN #GP(0); FI;
                FI;
            EFLAGS(CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
            IF CPL ≤ IOPL
                THEN EFLAGS(IF) := tempEFLAGS; FI;
            IF CPL = 0
                THEN
                    EFLAGS(IOPL) := tempEFLAGS;
                    IF OperandSize = 32 or OperandSize = 64
                        THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
            FI;
            IF ShadowStackEnabled(CPL)
                (* check if 8 byte aligned *)
                IF SSP AND 0x7 ≠ 0

```

```

    THEN #CP(FAR-RET/IRET); FI;
IF CS(RPL) != 3
    THEN
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        (* Do 64 bit compare to detect bits beyond 15 being set *)
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
        IF CSBASE + RIP != tempSsEIP
            THEN #CP(FAR-RET/IRET); FI;
        (* check if 4 byte aligned *)
        IF tempSSP AND 0x3 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;
tempOldCPL = CPL;
CPL := CS(RPL);
IF OperandSize = 64
    THEN
        RSP := tempRSP;
        SS := tempSS;
    ELSE
        ESP := tempESP;
        SS := tempSS;
    FI;
IF new mode != 64-Bit Mode
    THEN
        IF EIP is not within CS limit
            THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                THEN #GP(0); FI;
    FI;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
   ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
    THEN #GP(0); FI;
SSP := tempSSP;
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;

```

```

FOR each SegReg in (ES, FS, GS, and DS)
  DO
    tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
    IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
      THEN (* Segment register invalid *)
        SegmentSelector := 0; (*Segment selector becomes null*)
    FI;
  OD;
END;

```

```

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)

```

```

  IF new mode ≠ 64-Bit Mode
    THEN
      IF EIP is not within CS limit
        THEN #GP(0); FI;
      ELSE (* new mode = 64-bit mode *)
        IF RIP is non-canonical
          THEN #GP(0); FI;
        FI;
      EFLAGS(CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
      IF OperandSize = 32 or OperandSize = 64
        THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
      IF CPL ≤ IOPL
        THEN EFLAGS(IF) := tempEFLAGS; FI;
      IF CPL = 0
        THEN
          EFLAGS(IOPL) := tempEFLAGS;
          IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
        FI;
      IF ShadowStackEnabled(CPL)
        IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
          THEN #CP(FAR-RET/IRET); FI;
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
          THEN #CP(FAR-RET/IRET); FI;
        IF CSBASE + RIP != tempSsLIP (* 64 bit compare; CSBASE+RIP zero padded to 64 bit for 32 bit LIP *)
          THEN #CP(FAR-RET/IRET); FI;
        IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
          THEN #CP(FAR-RET/IRET); FI;
        IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
          ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
          THEN #GP(0); FI;
        FI;
      IF ShadowStackEnabled(CPL)
        IF IA32_EFER.LMA = 1
          (* In IA-32e-mode the IRET may be switching stacks if the interrupt/exception was delivered
          through an IDT with a non-zero IST *)
          (* In IA-32e mode for same CPL IRET there is always a stack switch. The below check verifies if the
          stack switch was to self stack and if so, do not try to free the token on this shadow stack. If the

```

```

tempSSP was not to same stack then there was a stack switch so do attempt to free the token *)
  IF tempSSP != SSP
    THEN
      expected_token_value = SSP | BUSY_BIT      (* busy bit - bit position 0 - must be set *)
      new_token_value = SSP                      (* clear the busy bit *)
      shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)
    FI;
  FI;
  SSP := tempSSP
FI;
END;

```

```

IA-32e-MODE:
  IF NT = 1
    THEN #GP(0);
  ELSE IF OperandSize = 32
    THEN
      EIP := Pop();
      CS := Pop();
      tempEFLAGS := Pop();
    ELSE IF OperandSize = 16
      THEN
        EIP := Pop(); (* 16-bit pop; clear upper bits *)
        CS := Pop(); (* 16-bit pop *)
        tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
      FI;
    ELSE (* OperandSize = 64 *)
      THEN
        RIP := Pop();
        CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        tempRFLAGS := Pop();
      FI;
    IF CS.RPL > CPL
      THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE
      IF instruction began in 64-Bit Mode
        THEN
          IF OperandSize = 32
            THEN
              ESP := Pop();
              SS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE IF OperandSize = 16
              THEN
                ESP := Pop(); (* 16-bit pop; clear upper bits *)
                SS := Pop(); (* 16-bit pop *)
              ELSE (* OperandSize = 64 *)
                RSP := Pop();
                SS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
              FI;
            FI;
          GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
    END;

```

## Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

## Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL. If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits. If the return code segment selector RPL is less than the CPL. If the DPL of a conforming-code segment is greater than the return code segment selector RPL. If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. If the stack segment is not a writable data segment. If the stack segment selector RPL is not equal to the RPL of the return code segment selector. If the segment descriptor for a code segment does not indicate it is a code segment. If the segment selector for a TSS has its local/global bit set for local. If a TSS segment descriptor specifies that the TSS is not busy. If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits. If the return stack segment is not present.
#NP (selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP (Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned. If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB. If return instruction pointer from stack and shadow stack do not match.

## Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

## Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit. If IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.
#UD	If the LOCK prefix is used.



## Compatibility Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.  
Other exceptions same as in Protected Mode.

## 64-Bit Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.  
If the return code segment selector is NULL.  
If the stack segment selector is NULL going back to compatibility mode.  
If the stack segment selector is NULL going back to CPL3 64-bit mode.  
If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.  
If the return instruction pointer is not within the return code segment limit.  
If the return instruction pointer is non-canonical.

#GP(Selector) If a segment selector index is outside its descriptor table limits.  
If a segment descriptor memory address is non-canonical.  
If the segment descriptor for a code segment does not indicate it is a code segment.  
If the proposed new code segment descriptor has both the D-bit and L-bit set.  
If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.  
If CPL is greater than the RPL of the code segment selector.  
If the DPL of a conforming-code segment is greater than the return code segment selector RPL.  
If the stack segment is not a writable data segment.  
If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.  
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0) If an attempt to pop a value off the stack violates the SS limit.  
If an attempt to pop a value off the stack causes a non-canonical address to be referenced.  
If the return stack segment is not present.

#NP (selector) If the return code segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

#UD If the LOCK prefix is used.

#CP (Far-RET/IRET) If the previous SSP from shadow stack (when returning to CPL <3) or from IA32\_PL3\_SSP (returning to CPL 3) is not 4 byte aligned.  
If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32\_PL3\_SSP (returning to CPL 3) is beyond 4GB.  
If return instruction pointer from stack and shadow stack do not match.

## JMP—Jump

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
EB <i>cb</i>	JMP <i>rel8</i>	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 <i>cw</i>	JMP <i>rel16</i>	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 <i>cd</i>	JMP <i>rel32</i>	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF <i>14</i>	JMP <i>r/m16</i>	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended <i>r/m16</i> . Not supported in 64-bit mode.
FF <i>14</i>	JMP <i>r/m32</i>	M	N.S.	Valid	Jump near, absolute indirect, address given in <i>r/m32</i> . Not supported in 64-bit mode.
FF <i>14</i>	JMP <i>r/m64</i>	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-bit offset from register or memory
EA <i>cd</i>	JMP <i>ptr16:16</i>	S	Inv.	Valid	Jump far, absolute, address given in operand
EA <i>cp</i>	JMP <i>ptr16:32</i>	S	Inv.	Valid	Jump far, absolute, address given in operand
FF <i>15</i>	JMP <i>m16:16</i>	M	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:16</i>
FF <i>15</i>	JMP <i>m16:32</i>	M	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:32</i> .
REX.W FF <i>15</i>	JMP <i>m16:64</i>	M	Valid	N.E.	Jump far, absolute indirect, address given in <i>m16:64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
S	Segment + Absolute Address	NA	NA	NA
D	Offset	NA	NA	NA
M	ModRM:r/m ( <i>r</i> )	NA	NA	NA

### Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to  $-128$  to  $+127$  from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

**Near and Short Jumps.** When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current

value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

**Far Jumps in Real-Address or Virtual-8086 Mode.** When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Jumps in Protected Mode.** When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for CET details.

**In 64-Bit Mode.** The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

**Instruction ordering.** Instructions following a far jump may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far jump have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Certain situations may lead to the next sequential instruction after a near indirect JMP being speculatively executed. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an INT3 or LFENCE instruction opcode can be placed after the near indirect JMP in order to block speculative execution.

## Operation

```

IF near jump
  IF 64-bit Mode
    THEN
      IF near relative jump
        THEN
          tempRIP := RIP + DEST; (* RIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempRIP := DEST;
        FI;
      ELSE
        IF near relative jump
          THEN
            tempEIP := EIP + DEST; (* EIP is instruction following JMP instruction*)
          ELSE (* Near absolute jump *)
            tempEIP := DEST;
          FI;
        FI;
      IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
        and tempEIP outside code segment limit
          THEN #GP(0); FI
      IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
      FI;
      IF OperandSize = 32
        THEN
          EIP := tempEIP;
        ELSE
          IF OperandSize = 16
            THEN (* OperandSize = 16 *)
              EIP := tempEIP AND 0000FFFFH;
            ELSE (* OperandSize = 64)

```

```

        RIP := tempRIP;
    FI;
FI;
IF (JMP near indirect, absolute indirect)
    IF EndbranchEnabledAndNotSuppressed(CPL)
        IF CPL = 3
            THEN
                IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                ELSE
                    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                        FI;
                    FI;
                FI;
            FI;
        FI;
    FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        tempEIP := DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS := DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP := tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP := tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
            FI;
        FI;
    FI;
IF far jump and (PE = 1 and VM = 0)
    (* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (IA32_EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
            FI;
        Depending on type and access rights:
        GO TO CONFORMING-CODE-SEGMENT;

```

```

        GO TO NONCONFORMING-CODE-SEGMENT;
        GO TO CALL-GATE;
        GO TO TASK-GATE;
        GO TO TASK-STATE-SEGMENT;
ELSE
    #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF DPL > CPL
    THEN #GP(segment selector); FI;
IF segment not present
    THEN #NP(segment selector); FI;
tempEIP := DEST(Offset);
IF OperandSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH;
FI;
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
tempEIP outside code segment limit
    THEN #GP(0); FI
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
    FI;
FI;
CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL
EIP := tempEIP;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;
NONCONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF (RPL > CPL) OR (DPL ≠ CPL)
    THEN #GP(code segment selector); FI;
IF segment not present
    THEN #NP(segment selector); FI;
tempEIP := DEST(Offset);
IF OperandSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH; FI;

```

```

IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
and tempEIP outside code segment limit
    THEN #GP(0); FI
IF tempEIP is non-canonical THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
    FI;
FI;
CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

```

## CALL-GATE:

```

IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); FI;
IF call gate not present
    THEN #NP(call gate selector); FI;
IF call gate code-segment selector is NULL
    THEN #GP(0); FI;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); FI;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
or code-segment segment descriptor is conforming and DPL > CPL
or code-segment segment descriptor is non-conforming and DPL ≠ CPL
    THEN #GP(code segment selector); FI;
IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
or code-segment segment descriptor has both L-Bit and D-bit set)
    THEN #GP(code segment selector); FI;
IF code segment is not present
    THEN #NP(code-segment selector); FI;
tempEIP := DEST(Offset);
IF GateSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH; FI;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
outside code segment limit
    THEN #GP(0); FI
CS := DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;

```

```

EIP := tempEIP;
IF EndbranchEnabled(CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;
END;
TASK-GATE:
  IF task gate DPL < CPL
  or task gate DPL < task gate segment-selector RPL
    THEN #GP(task gate selector); FI;
  IF task gate not present
    THEN #NP(gate selector); FI;
  Read the TSS segment selector in the task-gate descriptor;
  IF TSS segment selector local/global bit is set to local
  or index not within GDT limits
  or descriptor is not a TSS segment
  or descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); FI;
  IF TSS not present
    THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
  IF TSS DPL < CPL
  or TSS DPL < TSS segment-selector RPL
  or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); FI;
  IF TSS is not present
    THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
END;

```

### Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

### Protected Mode Exceptions

#GP(0)	<p>If offset in target operand, call gate, or TSS is beyond the code segment limits.</p> <p>If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p>
#GP(selector)	If the segment selector index is outside descriptor table limits.



If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.

If the DPL for a nonconforming-code segment is not equal to the CPL

(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

If the segment descriptor for selector in a call gate does not indicate it is a code segment.

If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP (selector) If the code segment being accessed is not present.

If call gate, task gate, or TSS not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0) If the target operand is beyond the code segment limits.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)

#UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

### 64-Bit Mode Exceptions

#GP(0) If a memory address is non-canonical.

If target offset in destination operand is non-canonical.

If target offset in destination operand is beyond the new code segment limit.

If the segment selector in the destination operand is NULL.

If the code segment selector in the 64-bit gate is NULL.

If transitioning to compatibility mode and the SSP is beyond 4GB.

#GP(selector) If the code segment or 64-bit call gate is outside descriptor table limits.

If the code segment or 64-bit call gate overlaps non-canonical space.

If the segment descriptor from a 64-bit call gate is in non-canonical space.

If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.

If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set.

If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.

If the upper type field of a 64-bit call gate is not 0x0.

If the segment selector from a 64-bit call gate is beyond the descriptor table limits.

If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.

If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.

If the code segment is non-conforming and  $CPL \neq DPL$ .

If the code segment is confirming and  $CPL < DPL$ .

#NP(selector) If a code segment or 64-bit call gate is not present.

#UD (64-bit mode only) If a far jump is direct to an absolute address in memory.

If the LOCK prefix is used.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## 10. Updates to Chapter 4, Volume 2B

Change bars and green text show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U*.

---

Changes to this chapter:

- MOV: Updated control register information.
- PCONFIG: Updated with register usage information.
- RDPMC: Corrected a typo in the description.
- RET: Updates to pseudocode.
- RSTORSSP: Corrected typos in the pseudocode.
- UD: Updated footnote to remove the term "older".

Additionally, there are three updates made to various instructions that are categorized as either typo corrections or cross-reference updates. These instruction pages are not included in this change document as their impact is evident.

- Updated instructions that were missing "(r)" after EVEX.vvvv or VEX.vvvvv: MAXPD, MAXPS, MAXSD, MAXSS, MINPD, MINPS, MINSD, MINSS, MOVHPD, MOVHPS, MOVLPD, and MOVLPS.
- Updated instructions incorrectly using "EVEX.LIG" to use "EVEX.LLIG": MAXSD, MAXSS, MINSD, MINSS, MOVSD, MOVSS, MULSD, MULSS, SQRTSD, SQRTSS, SUBSD, SUBSS, UCOMISD, and UCOMISS.
- Updated all instructions that list a specific exceptions type to use a cross-reference in order to allow users to jump directly to the applicable exception section for the instruction.

## MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 20/r MOV r32, CR0–CR7	MR	N.E.	Valid	Move control register to r32.
OF 20/r MOV r64, CR0–CR7	MR	Valid	N.E.	Move extended control register to r64.
REX.R + OF 20 /0 MOV r64, CR8	MR	Valid	N.E.	Move extended CR8 to r64. <sup>1</sup>
OF 22 /r MOV CR0–CR7, r32	RM	N.E.	Valid	Move r32 to control register.
OF 22 /r MOV CR0–CR7, r64	RM	Valid	N.E.	Move r64 to extended control register.
REX.R + OF 22 /0 MOV CR8, r64	RM	Valid	N.E.	Move r64 to extended CR8. <sup>1</sup>

### NOTE:

1. MOV CR\* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general-purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. **On a 64-bit capable processor, an execution of MOV to CR outside of 64-bit mode zeros the upper 32 bits of the control register.** (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. **Some of the bits in CR0, CR3 and CR4 are reserved and must be written with zeros. Attempting to set any reserved bits in CR0[31:0] is ignored. Attempting to set any reserved bits in CR0[63:32] results in a general-protection exception, #GP(0). When PCIDs are not enabled, bits 2:0 and bits 11:5 of CR3 are not used and attempts to set them are ignored. Attempting to set any reserved bits in CR3[63:MAXPHYADDR] results in #GP(0). Attempting to set any reserved bits in CR4 results in #GP(0). On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.**

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8–R15). Use of the REX.W prefix or 66H prefix is ignored. Use of

the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 24 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

DEST := SRC;

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).</p> <p>If an attempt is made to write a 1 to any reserved bit in CR4.</p> <p>If an attempt is made to write 1 to CR4.PCIDE.</p> <p>If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.</p>

## Real-Address Mode Exceptions

#GP	<p>If an attempt is made to write a 1 to any reserved bit in CR4.</p> <p>If an attempt is made to write 1 to CR4.PCIDE.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.</p>

## Virtual-8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual-8086 mode.
--------	---

## Compatibility Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).</p> <p>If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.</p> <p>If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.</p> <p>If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.</p>

### 64-Bit Mode Exceptions

- #GP(0)
  - If the current privilege level is not 0.
  - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
  - If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
  - If an attempt is made to clear CR0.PG[bit 31].
  - If an attempt is made to write a 1 to any reserved bit in CR4.
  - If an attempt is made to write a 1 to any reserved bit in CR8.
  - If an attempt is made to write a 1 to any reserved bit in CR3[63:MAXPHYADDR].
  - If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
- #UD
  - If the LOCK prefix is used.
  - If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.
  - If the REX.R prefix is used to specify a register other than CR8.

## PCONFIG – Platform Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C5 PCONFIG	A	V/V	PCONFIG	This instruction is used to execute functions for configuring platform features.

### Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

### Description

PCONFIG allows software to configure certain platform features. PCONFIG supports multiple leaf functions, with a leaf function identified by the value in EAX. The registers RBX, RCX, and RDX may provide input or output information for certain leaves. All leaves write status information to EAX but do not modify RBX, RCX, or RDX unless they are being used as leaf-specific output.

Each PCONFIG leaf function applies to a specific hardware block called a PCONFIG target, and each PCONFIG target is associated with a numerical **target identifier**. Supported target identifiers are enumerated, along with other PCONFIG capabilities, in the sub-leaves of the PCONFIG-information leaf of CPUID (EAX = 1BH). An attempt to execute an undefined leaf function, or a leaf function that applies to an unsupported target identifier, results in a general-protection exception (#GP). (In the future, the PCONFIG-information leaf of CPUID may enumerate PCONFIG capabilities in addition to the supported target identifiers.)

Addresses and operands are 32 bits outside 64-bit mode and are 64 bits in 64-bit mode. The value of CS.D does not affect operand size or address size.

Table 4-15 shows the leaf encodings for PCONFIG, and Table 4-16 shows the leaf register usage for PCONFIG.

**Table 4-15. PCONFIG Leaf Encodings**

Leaf	Encoding	Description
MKTME_KEY_PROGRAM	00000000H	This leaf is used to program the key and encryption mode associated with a KeyID.
RESERVED	00000001H - FFFFFFFFH	Reserved for future use (#GP(0) if used).

**Table 4-16. PCONFIG Leaf Register Usage**

Leaf	RBX	RCX	RDX
MKTME_KEY_PROGRAM	Input only.	Input only.	Input only.
RESERVED	Reserved for future use.	Reserved for future use.	Reserved for future use.

The MKTME\_KEY\_PROGRAM leaf of PCONFIG pertains to the MKTME<sup>1</sup> target, which has target identifier 1. It is used by software to manage the key associated with a KeyID. The leaf function is invoked by setting the leaf value of 0 in EAX and the address of MKTME\_KEY\_PROGRAM\_STRUCT in RBX. Successful execution of the leaf clears RAX (set to zero) and ZF, CF, PF, AF, OF, and SF are cleared. In case of failure, the failure reason is indicated in RAX with ZF set to 1 and CF, PF, AF, OF, and SF are cleared. The MKTME\_KEY\_PROGRAM leaf uses the MKTME\_KEY\_PROGRAM\_STRUCT in memory shown in Table 4-17.

1. Further details on MKTME usage can be found here:

<https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf>

Table 4-17. MKTME\_KEY\_PROGRAM\_STRUCT Format

Field	Offset (bytes)	Size (bytes)	Comments
KEYID	0	2	Key Identifier.
KEYID_CTRL	2	4	KeyID control: <ul style="list-style-type: none"> <li>▪ Bits [7:0]: COMMAND.</li> <li>▪ Bits [23:8]: ENC_ALG.</li> <li>▪ Bits [31:24]: Reserved, must be zero.</li> </ul>
RESERVED	6	58	Reserved, must be zero.
KEY_FIELD_1	64	64	Software supplied KeyID data key or entropy for KeyID data key.
KEY_FIELD_2	128	64	Software supplied KeyID tweak key or entropy for KeyID tweak key.

A description of each of the fields in MKTME\_KEY\_PROGRAM\_STRUCT is provided below:

- **KEYID:** Key Identifier being programmed to the MKTME engine.
- **KEYID\_CTRL:** The KEYID\_CTRL field carries two sub-fields used by software to control the behavior of a KeyID: Command and KeyID encryption algorithm.

The command used controls the encryption mode for a KeyID. Table 4-18 provides a summary of the commands supported.

Table 4-18. Supported Key Programming Commands

Command	Encoding	Description
KEYID_SET_KEY_DIRECT	0	Software uses this mode to directly program a key for use with KeyID.
KEYID_SET_KEY_RANDOM	1	CPU generates and assigns an ephemeral key for use with a KeyID. Each time the instruction is executed, the CPU generates a new key using a hardware random number generator and the keys are discarded on reset.
KEYID_CLEAR_KEY	2	Clear the (software programmed) key associated with the KeyID. On execution of this command, the KeyID gets TME behavior (encrypt with platform TME key or bypass TME encryption).
KEYID_NO_ENCRYPT	3	Do not encrypt memory when this KeyID is in use.

The encryption algorithm field (ENC\_ALG) allows software to select one of the activated encryption algorithms for the KeyID. The BIOS can activate a set of algorithms to allow for use when programming keys using the IA32\_TME\_ACTIVATE MSR (does not apply to KeyID 0 which uses the TME policy when TME encryption is not bypassed). The processor checks to ensure that the algorithm selected by software is one of the algorithms that has been activated by the BIOS.

- **KEY\_FIELD\_1:** This field carries the software supplied data key to be used for the KeyID if the direct key programming option is used (KEYID\_SET\_KEY\_DIRECT). When the random key programming option is used (KEYID\_SET\_KEY\_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random data key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.
- **KEY\_FIELD\_2:** This field carries the software supplied tweak key to be used for the KeyID if the direct key programming option is used (KEYID\_SET\_KEY\_DIRECT). When the random key programming option is used (KEYID\_SET\_KEY\_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random tweak key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.

All KeyIDs default to TME behavior (encrypt with TME key or bypass encryption) on MKTME activation. Software can at any point decide to change the key for a KeyID using the PCONFIG instruction. Change of



keys for a KeyID does NOT change the state of the TLB caches or memory pipeline. It is software's responsibility to take appropriate actions to ensure correct behavior.

Table 4-19 shows the return values associated with the MKTME\_KEY\_PROGRAM leaf of PCONFIG. On instruction execution, RAX is populated with the return value.

**Table 4-19. Supported Key Error Codes**

Return Value	Encoding	Description
PROG_SUCCESS	0	KeyID was successfully programmed.
INVALID_PROG_CMD	1	Invalid KeyID programming command.
ENTROPY_ERROR	2	Insufficient entropy.
INVALID_KEYID	3	KeyID not valid.
INVALID_ENC_ALG	4	Invalid encryption algorithm chosen (not supported).
DEVICE_BUSY	5	Failure to access key table.

### PCONFIG Virtualization

Software in VMX root operation can control the execution of PCONFIG in VMX non-root operation using the following VM-execution controls introduced for PCONFIG:

- **PCONFIG\_ENABLE:** This control is a single bit control and enables the PCONFIG instruction in VMX non-root operation. If 0, the execution of PCONFIG in VMX non-root operation causes #UD. Otherwise, execution of PCONFIG works according to PCONFIG\_EXITING.
- **PCONFIG\_EXITING:** This is a 64b control and allows VMX root operation to cause a VM-exit for various leaf functions of PCONFIG. This control does not have any effect if the PCONFIG\_ENABLE control is clear. It is recommended that VMMs intercept execution of any PCONFIG leaves with which they are not familiar and convert such executions into #GP(0).

### PCONFIG Concurrency

In a scenario where the MKTME\_KEY\_PROGRAM leaf of PCONFIG is executed concurrently on multiple logical processors, only one logical processor will succeed in updating the key table. PCONFIG execution will return with an error code (DEVICE\_BUSY) on other logical processors and software must retry. In cases where the instruction execution fails with a DEVICE\_BUSY error code, the key table is not updated, thereby ensuring that either the key table is updated in its entirety with the information for a KeyID, or it is not updated at all. In order to accomplish this, the MKTME\_KEY\_PROGRAM leaf of PCONFIG maintains a writer lock for updating the key table. This lock is referred to as the Key table lock and denoted in the instruction flows as KEY\_TABLE\_LOCK. The lock can either be unlocked, when no logical processor is holding the lock (also the initial state of the lock) or be in an exclusive state where a logical processor is trying to update the key table. There can be only one logical processor holding the lock in exclusive state. The lock, being exclusive, can only be acquired when the lock is in unlocked state.

PCONFIG uses the following syntax to acquire KEY\_TABLE\_LOCK in exclusive mode and release the lock:

- KEY\_TABLE\_LOCK.ACQUIRE(WRITE)
- KEY\_TABLE\_LOCK.RELEASE()

### Operation

**Table 4-20. PCONFIG Operation Variables**

Variable Name	Type	Size (Bytes)	Description
TMP_KEY_PROGRAM_STRUCT	MKTME_KEY_PROGRAM_STRUCT	192	Structure holding the key programming structure.
TMP_RND_DATA_KEY	UINT128	16	Random data key generated for random key programming option.
TMP_RND_TWEAK_KEY	UINT128	16	Random tweak key generated for random key programming option.

```
(* #UD if PCONFIG is not enumerated or CPL>0 *)
IF (CPUID.7.0:EDX[18] == 0 OR CPL > 0) #UD;
```

```
IF (in VMX non-root mode)
{
  IF (VMCS.PCONFIG_ENABLE == 1)
  {
    IF ((EAX > 62 AND VMCS.PCONFIG_EXITING[63] == 1) OR
        (EAX < 63 AND VMCS.PCONFIG_EXITING[EAX] == 1))
    {
      Set VMCS.EXIT_REASON = PCONFIG; //No Exit qualification
      Deliver VMEXIT;
    }
  }
  ELSE
  {
    #UD
  }
}
```

```
(* #GP(0) for an unsupported leaf *)
IF (EAX != 0) #GP(0)
```

```
(* KEY_PROGRAM leaf flow *)
```

```
IF (EAX == 0)
{
  (* #GP(0) if TME_ACTIVATE MSR is not locked or does not enable hardware encryption or multiple keys are not enabled *)
  IF (IA32_TME_ACTIVATE.LOCK != 1 OR IA32_TME_ACTIVATE.ENABLE != 1 OR IA32_TME_ACTIVATE.MK_TME_KEYID_BITS == 0)
  #GP(0)
```

```
(* Check MKTME_KEY_PROGRAM_STRUCT is 256B aligned *)
IF (DS:RBX is not 256B aligned) #GP(0);
```

```
(* Check that MKTME_KEY_PROGRAM_STRUCT is read accessible *)
<<DS: RBX should be read accessible>>
```

```
(* Copy MKTME_KEY_PROGRAM_STRUCT to a temporary variable *)
TMP_KEY_PROGRAM_STRUCT = DS:RBX.*;
```

```
(* RSVD field check *)
IF (TMP_KEY_PROGRAM_STRUCT.RSVD != 0) #GP(0);
```

```
IF (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.RSVD != 0) #GP(0);
```

```
IF (TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[63:16] != 0) #GP(0);
```

```
IF (TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[63:16] != 0) #GP(0);
```

```
(* Check for a valid command *)
IF (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND is not a valid command)
{
  RFLAGS.ZF = 1;
  RAX = INVALID_PROG_CMD;
  goto EXIT;
```

```

}
(* Check that the KEYID being operated upon is a valid KEYID *)
IF (TMP_KEY_PROGRAM_STRUCT.KEYID >
    2^IA32_TME_ACTIVATE.MK_TME_KEYID_BITS - 1
    OR TMP_KEY_PROGRAM_STRUCT.KEYID >
    IA32_TME_CAPABILITY.MK_TME_MAX_KEYS
    OR TMP_KEY_PROGRAM_STRUCT.KEYID == 0)
{
    RFLAGS.ZF = 1;
    RAX = INVALID_KEYID;
    goto EXIT;
}

(* Check that only one algorithm is requested for the KeyID and it is one of the activated algorithms *)
IF (NUM_BITS(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG) != 1 ||
    (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG &
    IA32_TME_ACTIVATE.MK_TME_CRYPTO_ALGS == 0))
{
    RFLAGS.ZF = 1;
    RAX = INVALID_ENC_ALG;
    goto EXIT;
}

(* Try to acquire exclusive lock *)
IF (NOT KEY_TABLE_LOCK.ACQUIRE(WRITE))
{
    //PCONFIG failure
    RFLAGS.ZF = 1;
    RAX = DEVICE_BUSY;
    goto EXIT;
}

(* Lock is acquired and key table will be updated as per the command
   Before this point no changes to the key table are made *)

switch(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND)
{
case KEYID_SET_KEY_DIRECT:
    <<Write
        DATA_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1,
        TWEAK_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2,
        ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
        to MKTME Key table at index TMP_KEY_PROGRAM_STRUCT.KEYID
    >>
    break;

case KEYID_SET_KEY_RANDOM:
    TMP_RND_DATA_KEY = <<Generate a random key using hardware RNG>>
    IF (NOT ENOUGH_ENTROPY)
    {
        RFLAGS.ZF = 1;
        RAX = ENTROPY_ERROR;
        goto EXIT;
    }
    TMP_RND_TWEAK_KEY = <<Generate a random key using hardware RNG>>

```

```

IF (NOT ENOUGH ENTROPY)
{
    RFLAGS.ZF = 1;
    RAX = ENTROPY_ERROR;
    goto EXIT;
}
(* Mix user supplied entropy to the data key and tweak key *)
TMP_RND_DATA_KEY = TMP_RND_KEY XOR
    TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[15:0];
TMP_RND_TWEAK_KEY = TMP_RND_TWEAK_KEY XOR
    TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[15:0];

<<Write
    DATA_KEY=TMP_RND_DATA_KEY,
    TWEAK_KEY=TMP_RND_TWEAK_KEY,
    ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
    to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
>>
break;

case KEYID_CLEAR_KEY:
    <<Write
        DATA_KEY=0,
        TWEAK_KEY=0,
        ENCRYPTION_MODE = ENCRYPT_WITH_TME_KEY_OR_BYPASS,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
    >>

    break;
case KD_NO_ENCRYPT:
    <<Write
        ENCRYPTION_MODE=NO_ENCRYPTION,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
    >>
    break;
}

RAX = 0;
RFLAGS.ZF = 0;

//Release Lock
KEY_TABLE_LOCK(RELEASE);

EXIT:
RFLAGS.CF=0;
RFLAGS.PF=0;
RFLAGS.AF=0;
RFLAGS.OF=0;
RFLAGS.SF=0;
}

end_of_flow

```

**Protected Mode Exceptions**

#GP(0)	<p>If input value in EAX encodes an unsupported leaf.</p> <p>If IA32_TME_ACTIVATE MSR is not locked.</p> <p>If hardware encryption and MKTME capability are not enabled in IA32_TME_ACTIVATE MSR.</p> <p>If the memory operand is not 256B aligned.</p> <p>If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.</p> <p>If a memory operand effective address is outside the DS segment limit.</p>
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	<p>If any of the LOCK/REP/OSIZE/VEX prefixes are used.</p> <p>If current privilege level is not 0.</p> <p>If CPUID.7.0:EDX[bit 18] = 0</p> <p>If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.</p>

**Real-Address Mode Exceptions**

#GP	<p>If input value in EAX encodes an unsupported leaf.</p> <p>If IA32_TME_ACTIVATE MSR is not locked.</p> <p>If hardware encryption and MKTME capability are not enabled in IA32_TME_ACTIVATE MSR.</p> <p>If a memory operand is not 256B aligned.</p> <p>If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.</p>
#UD	<p>If any of the LOCK/REP/OSIZE/VEX prefixes are used.</p> <p>If current privilege level is not 0.</p> <p>If CPUID.7.0:EDX.PCONFIG[bit 18] = 0</p> <p>If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.</p>

**Virtual-8086 Mode Exceptions**

#UD	PCONFIG instruction is not recognized in virtual-8086 mode.
-----	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	<p>If input value in EAX encodes an unsupported leaf.</p> <p>If IA32_TME_ACTIVATE MSR is not locked.</p> <p>If hardware encryption and MKTME capability are not enabled in IA32_TME_ACTIVATE MSR.</p> <p>If a memory operand is not 256B aligned.</p> <p>If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.</p> <p>If a memory operand is non-canonical form.</p>
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	<p>If any of the LOCK/REP/OSIZE/VEX prefixes are used.</p> <p>If the current privilege level is not 0.</p> <p>If CPUID.7.0:EDX.PCONFIG[bit 18] = 0.</p> <p>If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.</p>

## RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 33	RDPMC	Z0	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Reads the contents of the performance monitoring counter (PMC) specified in ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the PMC and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the PMC being read, unimplemented bits returned to EDX:EAX will have value zero.

The width of PMCs on processors supporting architectural performance monitoring (CPUID.0AH:EAX[7:0] ≠ 0) are reported by CPUID.0AH:EAX[23:16]. On processors that do not support architectural performance monitoring (CPUID.0AH:EAX[7:0]=0), the width of general-purpose performance PMCs is 40 bits, while the widths of special-purpose PMCs are implementation specific.

Use of ECX to specify a PMC depends on whether the processor supports architectural performance monitoring:

- If the processor does not support architectural performance monitoring (CPUID.0AH:EAX[7:0]=0), ECX[30:0] specifies the index of the PMC to be read. Setting ECX[31] selects “fast” read mode if supported. In this mode, RDPMC returns bits 31:0 of the PMC in EAX while clearing EDX to zero.
- If the processor does support architectural performance monitoring (CPUID.0AH:EAX[7:0] ≠ 0), ECX[31:16] specifies type of PMC while ECX[15:0] specifies the index of the PMC to be read within that type. The following PMC types are currently defined:
  - General-purpose counters use type 0. The index x (to read IA32\_PMCx) must be less than the value enumerated by CPUID.0AH:EAX[15:8] (thus ECX[15:8] must be zero).
  - Fixed-function counters use type 4000H. The index x (to read IA32\_FIXED\_CTRx) can be used if either CPUID.0AH:EDX[4:0] > x or CPUID.0AH:ECX[x] = 1 (thus ECX[15:5] must be 0).
  - Performance metrics use type 2000H. This type can be used only if IA32\_PERF\_CAPABILITIES.PERF\_METRICS\_AVAILABLE[bit 15]=1. For this type, the index in ECX[15:0] is implementation specific.

Specifying an unsupported PMC encoding will cause a general protection exception #GP(0). For PMC details see Chapter 18, “Performance Monitoring”, in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

When in protected or virtual 8086 mode, the **Performance-monitoring Counters Enabled** (PCE) flag in register CR4 restricts the use of the RDPMC instruction. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.) The PMCs can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the PMC, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have PMCs, but they must be read with the RDMSR instruction.

### Operation

MSCB = Most Significant Counter Bit (\* Model-specific \*)

IF (((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)) and (ECX indicates a supported counter))

THEN

EAX := counter[31:0];

EDX := ZeroExtend(counter[MSCB:32]);

ELSE (\* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 \*)

#GP(0);

FI;

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

- #GP If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

- #GP(0) If the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

## RET—Return from Procedure

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	Z0	Valid	Valid	Near return to calling procedure.
CB	RET	Z0	Valid	Valid	Far return to calling procedure.
C2 <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Near return to calling procedure and pop <i>imm16</i> bytes from stack.
CA <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Far return to calling procedure and pop <i>imm16</i> bytes from stack.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
I	<i>imm16</i>	NA	NA	NA

### Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure’s stack and the calling procedure’s stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.



Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for CET details.

**Instruction ordering.** Instructions following a far return may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far return have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Unlike near indirect CALL and near indirect JMP, the processor will not speculatively execute the next sequential instruction after a near RET unless that instruction is also the target of a jump or is a target in a branch predictor.

## Operation

(\* Near return \*)

IF instruction = near return

THEN;

IF OperandSize = 32

THEN

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP := Pop();

IF ShadowStackEnabled(CPL)

tempSsEIP = ShadowStackPop4B();

IF EIP != TempSsEIP

THEN #CP(NEAR\_RET); FI;

FI;

ELSE

IF OperandSize = 64

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

RIP := Pop();

IF ShadowStackEnabled(CPL)

tempSsEIP = ShadowStackPop8B();

IF RIP != tempSsEIP

THEN #CP(NEAR\_RET); FI;

FI;

ELSE (\* OperandSize = 16 \*)

IF top 2 bytes of stack not within stack limits

THEN #SS(0); FI;

tempEIP := Pop();

tempEIP := tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits

THEN #GP(0); FI;

EIP := tempEIP;

IF ShadowStackEnabled(CPL)

tempSsEip = ShadowStackPop4B();

IF EIP != tempSsEIP

THEN #CP(NEAR\_RET); FI;

FI;

FI;

FI;

IF instruction has immediate operand

THEN (\* Release parameters from stack \*)

IF StackAddressSize = 32

THEN

```

        ESP := ESP + SRC;
    ELSE
        IF StackAddressSize = 64
            THEN
                RSP := RSP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP := SP + SRC;
        FI;
    FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP := Pop();
                CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP := Pop();
                tempEIP := tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP := tempEIP;
                CS := Pop(); (* 16-bit pop *)
            FI;
        IF instruction has immediate operand
            THEN (* Release parameters from stack *)
                SP := SP + (SRC AND FFFFH);
        FI;
    FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
            FI;
        IF return code segment selector is NULL
            THEN #GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN #GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment

```

```

    THEN #GP(selector); FI;
IF return code segment selector RPL < CPL
    THEN #GP(selector); FI;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is non-conforming and return code
segment DPL ≠ return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is not present
    THEN #NP(selector); FI;
IF return code segment selector RPL > CPL
    THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
FI;
FI;

RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE (* OperandSize = 16 *)
        EIP := Pop();
        EIP := EIP AND 0000FFFFH;
        CS := Pop(); (* 16-bit pop *)
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP := SP + SRC;
        FI;
FI;
IF ShadowStackEnabled(CPL)
    (* SSP must be 8 byte aligned *)
    IF SSP AND 0x7 != 0
        THEN #CP(FAR-RET/IRET); FI;
tempSsCS = shadow_stack_load 8 bytes from SSP+16;
tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
prevSSP = shadow_stack_load 8 bytes from SSP;
SSP = SSP + 24;
(* do a 64 bit-compare to check if any bits beyond bit 15 are set *)
tempCS = CS; (* zero pad to 64 bit *)
IF tempCS != tempSsCS
    THEN #CP(FAR-RET/IRET); FI;
(* do a 64 bit-compare; pad CSBASE+RIP with 0 for 32 bit LIP*)
IF CSBASE + RIP != tempSsLIP
    THEN #CP(FAR-RET/IRET); FI;
(* prevSSP must be 4 byte aligned *)

```

```

IF prevSSP AND 0x3 != 0
    THEN #CP(FAR-RET/IRET); FI;
(* In legacy mode SSP must be in low 4GB *)
IF prevSSP[63:32] != 0
    THEN #GP(0); FI;
SSP := prevSSP
FI;

```

## RETURN-TO-OUTER-PRIVILEGE-LEVEL:

```

IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
Read return segment selector;
IF stack segment selector is NULL
    THEN #GP(0); FI;
IF return stack segment selector index is not within its descriptor table limits
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
        CS(RPL) := ReturnCodeSegmentSelector(RPL);
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP := ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP := SP + SRC;
                FI;
            FI;
        tempESP := Pop();
        tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
    ELSE (* OperandSize = 16 *)
        EIP := Pop();
        EIP := EIP AND 0000FFFFH;
        CS := Pop(); (* 16-bit pop; segment descriptor loaded *)
        CS(RPL) := ReturnCodeSegmentSelector(RPL);
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP := ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP := SP + SRC;
                FI;
            FI;
    FI;

```

```

        FI;
    FI;
    tempESP := Pop();
    tempSS := Pop(); (* 16-bit pop; segment descriptor loaded *)
FI;
IF ShadowStackEnabled(CPL)
(* check if 8 byte aligned *)
IF SSP AND 0x7 != 0
    THEN #CP(FAR-RET/IRET); FI;
IF ReturnCodeSegmentSelector(RPL) != 3
    THEN
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        (* Do 64 bit compare to detect bits beyond 15 being set *)
        tempCS = CS; (* zero extended to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LA *)
        IF CSBASE + RIP != tempSsLIP
            THEN #CP(FAR-RET/IRET); FI;
        (* check if 4 byte aligned *)
        IF tempSSP AND 0x3 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;
FI;
tempOldCPL = CPL;

CPL := ReturnCodeSegmentSelector(RPL);
ESP := tempESP;
SS := tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
    IF tempSSP[63:32] != 0
        THEN #GP(0); FI;
    SSP := tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
FI;

FOR each SegReg in (ES, FS, GS, and DS)
    DO
        tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
        IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
            THEN (* Segment register invalid *)

```

```

        SegmentSelector := 0; (*Segment selector becomes null*)
    FI;
OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP := SP + SRC;
        FI;
    FI;

(* IA-32e Mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                IF first or second doubleword on stack is not in canonical space
                    THEN #SS(0); FI;
            ELSE
                IF OperandSize = 16
                    THEN
                        IF second word on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second word on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE (* OperandSize = 64 *)
                        IF first or second quadword on stack is not in canonical space
                            THEN #SS(0); FI;
                FI;
        FI;
    FI;
IF return code segment selector is NULL
    THEN GP(0); FI;
IF return code segment selector addresses descriptor beyond descriptor table limit
    THEN GP(selector); FI;
IF return code segment selector addresses descriptor in non-canonical space
    THEN GP(selector); FI;
Obtain descriptor to which return code segment selector points from descriptor table;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); FI;
IF return code segment descriptor has L-bit = 1 and D-bit = 1
    THEN #GP(selector); FI;
IF return code segment selector RPL < CPL
    THEN #GP(selector); FI;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is non-conforming
and return code segment DPL ≠ return code segment selector RPL
    THEN #GP(selector); FI;

```

```

    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP := Pop();
                EIP := EIP AND 0000FFFFH;
                CS := Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP := Pop();
                CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
    FI;
FI;

IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP := SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP := RSP + SRC;
                FI;
            FI;
        FI;
    FI;
FI;

IF ShadowStackEnabled(CPL)
    IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
        THEN #CP(FAR-RET/IRET); FI;
    tempSsCS = shadow_stack_load 8 bytes from SSP+16;
    tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
    tempSSP = shadow_stack_load 8 bytes from SSP;
    SSP = SSP + 24;
    tempCS = CS; (* zero padded to 64 bit *)
    IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
        THEN #CP(FAR-RET/IRET); FI;
    IF CSBASE + RIP != tempSsLIP (* 64 bit compare *)
        THEN #CP(FAR-RET/IRET); FI;

```

```

IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
    THEN #CP(FAR-RET/IRET); FI;
IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
    (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
    THEN #GP(0); FI;
SSP := tempSSP
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
FI;
IF return stack segment descriptor is not within descriptor table limits
    THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) := ReturnCodeSegmentSelector(RPL);
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP := ESP + SRC;
                    ELSE
                        IF StackAddressSize = 16
                            THEN
                                SP := SP + SRC;
                            ELSE (* StackAddressSize = 64 *)

```



```

                RSP := RSP + SRC;
            FI;
        FI;
    FI;
    tempESP := Pop();
    tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
ELSE
    IF OperandSize = 16
        THEN
            EIP := Pop();
            EIP := EIP AND 0000FFFFH;
            CS := Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL) := ReturnCodeSegmentSelector(RPL);
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP := ESP + SRC;
                        ELSE
                            IF StackAddressSize = 16
                                THEN
                                    SP := SP + SRC;
                                ELSE (* StackAddressSize = 64 *)
                                    RSP := RSP + SRC;
                            FI;
                        FI;
                    FI;
                FI;
            tempESP := Pop();
            tempSS := Pop(); (* 16-bit pop; segment descriptor loaded *)
        ELSE (* OperandSize = 64 *)
            RIP := Pop();
            CS := Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
            CS(RPL) := ReturnCodeSegmentSelector(RPL);
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    RSP := RSP + SRC;
            FI;
            tempESP := Pop();
            tempSS := Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
        FI;
    FI;
    IF ShadowStackEnabled(CPL)
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
        IF ReturnCodeSegmentSelector(RPL) != 3
            THEN
                tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                tempSSP = shadow_stack_load 8 bytes from SSP;
                SSP = SSP + 24;
                (* Do 64 bit compare to detect bits beyond 15 being set *)
                tempCS = CS; (* zero padded to 64 bit *)

```

```

    IF tempCS != tempSsCS
        THEN #CP(FAR-RET/IRET); FI;
    (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
    IF CSBASE + RIP != tempSsLIP
        THEN #CP(FAR-RET/IRET); FI;
    (* check if 4 byte aligned *)
    IF tempSSP AND 0x3 != 0
        THEN #CP(FAR-RET/IRET); FI;
FI;
FI;
tempOldCPL = CPL;
CPL := ReturnCodeSegmentSelector(RPL);
ESP := tempESP;
SS := tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
    IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
        (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
        THEN #GP(0); FI;
    SSP := tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT      (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                    (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN SegmentSelector := 0; (* SegmentSelector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP := SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP := RSP + SRC;
                FI;
            FI;
        FI;
FI;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If the return code or stack segment selector is NULL. If the return instruction pointer is not within the return code segment limit. If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
#GP(selector)	If the RPL of the return code segment selector is less than the CPL. If the return code or stack segment selector index is not within its descriptor table limits. If the return code segment descriptor does not indicate a code segment. If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector If the stack segment is not a writable data segment. If the stack segment selector RPL is not equal to the RPL of the return code segment selector. If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits. If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.
#CP(Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned. If return instruction pointer from stack and shadow stack do not match.

**Real-Address Mode Exceptions**

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

**Virtual-8086 Mode Exceptions**

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

**Compatibility Mode Exceptions**

Same as 64-bit mode exceptions.

**64-Bit Mode Exceptions**

#GP(0)	<p>If the return instruction pointer is non-canonical.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If the stack segment selector is NULL going back to compatibility mode.</p> <p>If the stack segment selector is NULL going back to CPL3 64-bit mode.</p> <p>If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.</p> <p>If the return code segment selector is NULL.</p> <p>If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL &lt;3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.</p>
#GP(selector)	<p>If the proposed segment descriptor for a code segment does not indicate it is a code segment.</p> <p>If the proposed new code segment descriptor has both the D-bit and L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p> <p>If CPL is greater than the RPL of the code segment selector.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If a segment selector index is outside its descriptor table limits.</p> <p>If a segment descriptor memory address is non-canonical.</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If an attempt to pop a value off the stack violates the SS limit.</p> <p>If an attempt to pop a value off the stack causes a non-canonical address to be referenced.</p>
#NP(selector)	<p>If the return code or stack segment is not present.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</p>
#CP(Far-RET/IRET)	<p>If the previous SSP from shadow stack (when returning to CPL &lt;3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.</p> <p>If return instruction pointer from stack and shadow stack do not match.</p>

## RSTORSSP—Restore Saved Shadow Stack Pointer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 /5 (modl=11, /5, memory only) RSTORSSP m64	M	V/V	CET_SS	Restore SSP.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

### Description

Restores SSP from the shadow-stack-restore token pointed to by m64. If the SSP restore was successful then the instruction replaces the shadow-stack-restore token with a previous-ssp token. The instruction sets the CF flag to indicate whether the SSP address recorded in the shadow-stack-restore token that was processed was 4 byte aligned, i.e., whether an alignment hole was created when the restore-shadow-stack token was pushed on this shadow stack.

Following RSTORSSP if a restore-shadow-stack token needs to be saved on the previous shadow stack, use the SAVEPREVSSP instruction.

If pushing a restore-shadow-stack token on the previous shadow stack is not required, the previous-ssp token can be popped using the INCSSPQ instruction. If the CF flag was set to indicate presence of an alignment hole, an additional INCSSPD instruction is needed to advance the SSP past the alignment hole.

### Operation

IF CPL = 3

IF (CR4.CET & IA32\_U\_CET.SH\_STK\_EN) = 0  
THEN #UD; FI;

ELSE

IF (CR4.CET & IA32\_S\_CET.SH\_STK\_EN) = 0  
THEN #UD; FI;

FI;

SSP\_LA = Linear\_Address(mem operand)

IF SSP\_LA not aligned to 8 bytes

THEN #GP(0); FI;

previous\_ssp\_token = SSP | (IA32\_EFER.LMA AND CS.L) | 0x02

Start Atomic Execution

restore\_ssp\_token = Locked shadow\_stack\_load 8 bytes from SSP\_LA

fault = 0

IF ((restore\_ssp\_token & 0x03) != (IA32\_EFER.LMA & CS.L))  
THEN fault = 1; FI; (\* If L flag in token does not match IA32\_EFER.LMA & CS.L or bit 1 is not 0 \*)

IF ((IA32\_EFER.LMA AND CS.L) = 0 AND restore\_ssp\_token[63:32] != 0)  
THEN fault = 1; FI; (\* If compatibility/legacy mode and SSP to be restored not below 4G \*)

TMP = restore\_ssp\_token & ~0x01

TMP = (TMP - 8)

TMP = TMP & ~0x07

```
IF TMP != SSP_LA
    THEN fault = 1; FI;    (* If address in token does not match the requested top of stack *)
```

```
TMP = (fault == 0) ? previous_ssp_token : restore_ssp_token
shadow_stack_store 8 bytes of TMP to SSP_LA and release lock
End Atomic Execution
```

```
IF fault == 1
    THEN #CP(RSTORSSP); FI;
```

```
SSP = SSP_LA
```

```
// Set the CF if the SSP in the restore token was 4 byte aligned, i.e., there is an alignment hole
RFLAGS.CF = (restore_ssp_token & 0x04) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF := 0;
```

### Flags Affected

CF is set to indicate if the shadow stack pointer in the restore token was 4 byte aligned, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

### C/C++ Compiler Intrinsic Equivalent

```
RSTORSSP void _rstorssp(void *);
```

### Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. If CPL = 3 and IA32_U_CET.SH_STK_EN = 0. If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If destination is located in a non-writable segment. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#CP(rstorssp)	If L bit in token does not match (IA32_EFER.LMA & CS.L). If address in token does not match linear address of memory operand. If in 32-bit or compatibility mode and the address in token is not below 4G.
#PF(fault-code)	If a page fault occurs.

### Real-Address Mode Exceptions

#UD	The RSTORSSP instruction is not recognized in real-address mode.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The RSTORSSP instruction is not recognized in virtual-8086 mode.
-----	--

### Compatibility Mode Exceptions

Same as protected mode exceptions.

### 64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. If CPL = 3 and IA32_U_CET.SH_STK_EN = 0. If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned. If a memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#CP(rstorssp)	If L bit in token does not match (IA32_EFER.LMA & CS.L). If address in token does not match linear address of memory operand.
#PF(fault-code)	If a page fault occurs.

## UD—Undefined Instruction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF FF /r	UD0 <sup>1</sup> <i>r32, r/m32</i>	RM	Valid	Valid	Raise invalid opcode exception.
OF B9 /r	UD1 <i>r32, r/m32</i>	RM	Valid	Valid	Raise invalid opcode exception.
OF 0B	UD2	ZO	Valid	Valid	Raise invalid opcode exception.

### NOTES:

1. Some processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid-opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

### Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

#UD (\* Generates invalid opcode exception \*);

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD                      Raises an invalid opcode exception in all operating modes.



## 11. Updates to Chapter 5, Volume 2C

Change bars and green text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V-Z*.

---

Changes to this chapter:

- VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD: Corrected typo (replaced "gathered" with "scattered").
- XRSTOR and XRSTORS: Updated pseudocode.

Additionally, there are three updates made to various instructions that are categorized as either typo corrections or cross-reference updates. These instruction pages are not included in this change document as their impact is evident.

- Updated instructions that were missing "(r)" after EVEX.vvvv or VEX.vvvvv: VALIGND/VALIGNQ, VBLENDMPD/VBLENDMPS, VCVTUSI2SD, VCVTUSI2SS, VDBPSADBW, VFIXUPIMMPD, VFIXUPIMMPS, VFIXUPIMMSD, VFIXUPIMMSS, VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4, VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4, VPBLEND, VPBLENDMB/VPBLENDMW, VPBLENDMD/VPBLENDMQ, VPDPWSSDS, VPERM2I128, VPERMD/VPERMW, VPMASKMOV, VPSHLD, VPSHLDV, VPSHRD, VPSHRDV, VRSQRT14SS, and XORPS.
- Updated instructions incorrectly using "EVEX.LIG" to use "EVEX.LLIG": VCVTSD2USI, VCVTSS2USI, VCVTSD2USI, VCVTSS2USI, VCVTUSI2SD, VCVTUSI2SS, VFIXUPIMMSD, VFIXUPIMMSS, VFMADD132SD/VFMADD213SD/VFMADD231SD, VFMADD132SS/VFMADD213SS/VFMADD231SS, VFMSUB132SD/VFMSUB213SD/VFMSUB231SD, VFMSUB132SS/VFMSUB213SS/VFMSUB231SS, VFNMADD132SD/VFNMADD213SD/VFNMADD231SD, VFNMADD132SS/VFNMADD213SS/VFNMADD231SS, VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD, VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS, VFPCLASSSD, VFPCLASSSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VRANGESD, VRANGESS, VRCP14SD, VRCP14SS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VRSQRT14SD, VRSQRT14SS, VSCALEFSD, and VSCALEFSS.
- Updated all instructions that list a specific exceptions type to use a cross-reference in order to allow users to jump directly to the applicable exception section for the instruction.

## VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, ymm1	A	V/V	AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1	A	V/V	AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

### Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be **scattered** before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be **scattered**.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1 or 4 byte displacement

#### VSCATTERDPS (EVEX encoded versions)

(KL, VL)= (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN MEM[BASE\_ADDR + SignExtend(VINDEX[i+31:i]) \* SCALE + DISP] :=

      SRC[i+31:i]

      k1[j] := 0

  FI;

ENDFOR

k1[MAX\_KL-1:KL] := 0

#### VSCATTERDPD (EVEX encoded versions)

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

  i := j \* 64

  k := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP] :=

      SRC[i+63:i]

      k1[j] := 0

  FI;

ENDFOR

k1[MAX\_KL-1:KL] := 0

**VSCATTERQPS (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 32

k := j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEK[k+63:k]) \* SCALE + DISP] :=

SRC[i+31:i]

k1[j] := 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] := 0

**VSCATTERQPD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEK[i+63:i]) \* SCALE + DISP] :=

SRC[i+63:i]

k1[j] := 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERDPD void \_\_mm512\_i32scatter\_pd(void \* base, \_\_m256i vdx, \_\_m512d a, int scale);

VSCATTERDPD void \_\_mm512\_mask\_i32scatter\_pd(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m512d a, int scale);

VSCATTERDPS void \_\_mm512\_i32scatter\_ps(void \* base, \_\_m512i vdx, \_\_m512 a, int scale);

VSCATTERDPS void \_\_mm512\_mask\_i32scatter\_ps(void \* base, \_\_mmask16 k, \_\_m512i vdx, \_\_m512 a, int scale);

VSCATTERQPD void \_\_mm512\_i64scatter\_pd(void \* base, \_\_m512i vdx, \_\_m512d a, int scale);

VSCATTERQPD void \_\_mm512\_mask\_i64scatter\_pd(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m512d a, int scale);

VSCATTERQPS void \_\_mm512\_i64scatter\_ps(void \* base, \_\_m512i vdx, \_\_m256 a, int scale);

VSCATTERQPS void \_\_mm512\_mask\_i64scatter\_ps(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m256 a, int scale);

VSCATTERDPD void \_\_mm256\_i32scatter\_pd(void \* base, \_\_m128i vdx, \_\_m256d a, int scale);

VSCATTERDPD void \_\_mm256\_mask\_i32scatter\_pd(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m256d a, int scale);

VSCATTERDPS void \_\_mm256\_i32scatter\_ps(void \* base, \_\_m256i vdx, \_\_m256 a, int scale);

VSCATTERDPS void \_\_mm256\_mask\_i32scatter\_ps(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256 a, int scale);

VSCATTERQPD void \_\_mm256\_i64scatter\_pd(void \* base, \_\_m256i vdx, \_\_m256d a, int scale);

VSCATTERQPD void \_\_mm256\_mask\_i64scatter\_pd(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256d a, int scale);

VSCATTERQPS void \_\_mm256\_i64scatter\_ps(void \* base, \_\_m256i vdx, \_\_m128 a, int scale);

VSCATTERQPS void \_\_mm256\_mask\_i64scatter\_ps(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m128 a, int scale);

VSCATTERDPD void \_\_mm\_i32scatter\_pd(void \* base, \_\_m128i vdx, \_\_m128d a, int scale);

VSCATTERDPD void \_\_mm\_mask\_i32scatter\_pd(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128d a, int scale);

VSCATTERDPS void \_\_mm\_i32scatter\_ps(void \* base, \_\_m128i vdx, \_\_m128 a, int scale);

VSCATTERDPS void \_\_mm\_mask\_i32scatter\_ps(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128 a, int scale);

VSCATTERQPD void \_\_mm\_i64scatter\_pd(void \* base, \_\_m128i vdx, \_\_m128d a, int scale);

VSCATTERQPD void \_\_mm\_mask\_i64scatter\_pd(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128d a, int scale);

VSCATTERQPS void \_\_mm\_i64scatter\_ps(void \* base, \_\_m128i vdx, \_\_m128 a, int scale);

VSCATTERQPS void \_\_mm\_mask\_i64scatter\_ps(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128 a, int scale);

**SIMD Floating-Point Exceptions**

Invalid, Overflow, Underflow, Precision, Denormal

**Other Exceptions**

See Table 2-61, “Type E12 Class Exception Conditions”.

## XRSTOR—Restore Processor Extended States

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF AE /5 XRSTOR <i>mem</i>	M	V/V	XSAVE	Restore state components specified by EDX:EAX from <i>mem</i> .
NP REX.W + OF AE /5 XRSTOR64 <i>mem</i>	M	V/N.E.	XSAVE	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.8, “Operation of XRSTOR,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP\_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If  $RFBM[i] = 0$ , XRSTOR does not update state component  $i$ .<sup>1</sup>
- If  $RFBM[i] = 1$  and bit  $i$  is clear in the XSTATE\_BV field in the XSAVE header, XRSTOR initializes state component  $i$ .
- If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ , XRSTOR loads state component  $i$  from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component  $i$  for which  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ ; it tracks as modified any state component  $i$  for which  $RFBM[i] = 0$ .

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR\_INFO.

1. There is an exception if  $RFBM[1] = 0$  and  $RFBM[2] = 1$ . In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

## Operation

```
RFBM := XCRO AND EDX:EAX; /* bitwise logical AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;
```

```
IF COMPMASK[63] = 0
  THEN
    /* Standard form of XRSTOR */
    TO_BE_RESTORED := RFBM AND RSTORMASK;
    TO_BE_INITIALIZED := RFBM AND NOT RSTORMASK;

    IF TO_BE_RESTORED[0] = 1
      THEN
        XINUSE[0] := 1;
        load x87 state from legacy region of XSAVE area;
      ELSIF TO_BE_INITIALIZED[0] = 1
        THEN
          XINUSE[0] := 0;
          initialize x87 state;
    FI;

    IF RFBM[1] = 1 OR RFBM[2] = 1
      THEN load MXCSR from legacy region of XSAVE area;
    FI;

    IF TO_BE_RESTORED[1] = 1
      THEN
        XINUSE[1] := 1;
        load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
      ELSIF TO_BE_INITIALIZED[1] = 1
        THEN
          XINUSE[1] := 0;
          set all XMM registers to 0; // this step does not initialize MXCSR
    FI;

    FOR i := 2 TO 62
      IF TO_BE_RESTORED[i] = 1
        THEN
          XINUSE[i] := 1;
          load XSAVE state component i at offset n from base of XSAVE area;
          // n enumerated by CPUID(EAX=ODH,ECX=i):EBX
        ELSIF TO_BE_INITIALIZED[i] = 1
          THEN
            XINUSE[i] := 0;
            initialize XSAVE state component i;
      FI;
    ENDFOR;

  ELSE
    /* Compacted form of XRSTOR */
    IF CPUID.(EAX=ODH,ECX=1):EAX.XSAVEC[bit 1] = 0
      THEN /* compacted form not supported */
        #GP(0);
```

```

FI;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        XINUSE[0] := 1;
        load x87 state from legacy region of XSAVE area;
    ELSIF TO_BE_INITIALIZED[0] = 1
        THEN
            XINUSE[0] := 0;
            initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        XINUSE[1] := 1;
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
    ELSIF TO_BE_INITIALIZED[1] = 1
        THEN
            set all XMM registers to 0;
            XINUSE[1] := 0;
            MXCSR := 1F80H;
FI;

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                FI;
                NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    XINUSE[i] := 0;
                    initialize XSAVE state component i;
            FI;
        ENDFOR;
FI;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;

```



LAXA := linear address of XSAVE area;  
 XRSTOR\_INFO := <CPL,VMXNR,LAXA,COMPMASK>;

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: void\_xrstor( void \*, unsigned \_\_int64);  
 XRSTOR: void\_xrstor64( void \*, unsigned \_\_int64);

## Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p>
-----	---

If the compacted form is executed and a bit in the XCOMP\_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE\_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	<p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XRSTORS—Restore Processor Extended States Supervisor

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C7 /3 XRSTORS <i>mem</i>	M	V/V	XSS	Restore state components specified by EDX:EAX from <i>mem</i> .
NP REX.W + OF C7 /3 XRSTORS64 <i>mem</i>	M	V/N.E.	XSS	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r</i> )	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32\_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.12, “Operation of XRSTORS,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32\_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE\_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR\_INFO.

**Operation**

```
RFBM := (XCRO OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;
```

```
FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;
```

```
IF TO_BE_RESTORED[0] = 1
    THEN
```

```
    XINUSE[0] := 1;
    load x87 state from legacy region of XSAVE area;
```

```
ELSIF TO_BE_INITIALIZED[0] = 1
```

```
    THEN
    XINUSE[0] := 0;
    initialize x87 state;
```

```
FI;
```

```
IF TO_BE_RESTORED[1] = 1
```

```
    THEN
    XINUSE[1] := 1;
    load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
```

```
ELSIF TO_BE_INITIALIZED[1] = 1
```

```
    THEN
    set all XMM registers to 0;
    XINUSE[1] := 0;
    MXCSR := 1F80H;
```

```
FI;
```

```
NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
```

```
FOR i := 2 TO 62
```

```
    IF FORMAT[i] = 1
        THEN
```

```
            IF TO_BE_RESTORED[i] = 1
                THEN
```

```
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
```

```
            FI;
```

```
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
```

```
        FI;
```

```
        IF TO_BE_INITIALIZED[i] = 1
```

```
            THEN
            XINUSE[i] := 0;
            initialize XSAVE state component i;
```

```
        FI;
```

```
ENDFOR;
```

```
XMODIFIED := NOT RFBM;
```

```
IF in VMX non-root operation
```

```

    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := <CPL,VMXNR,LAXA,COMPMASK>;

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XRSTORS:    void _xrstors( void *, unsigned __int64);
XRSTORS64: void _xrstors64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	<p>If CPL &gt; 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

### Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #GP(0)
  - If CPL > 0.
  - If a memory address is in a non-canonical form.
  - If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
  - If bit 63 of the XCOMP\_BV field of the XSAVE header is 0.
  - If a bit in XCR0 is 0 and the corresponding bit in the XCOMP\_BV field of the XSAVE header is 1.
  - If a bit in the XCOMP\_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE\_BV field is 1.
  - If bytes 63:16 of the XSAVE header are not all zero.
  - If attempting to write any reserved bits of the MXCSR register with 1.
- #SS(0)
  - If a memory address referencing the SS segment is in a non-canonical form.
- #PF(fault-code)
  - If a page fault occurs.
- #NM
  - If CR0.TS[bit 3] = 1.
- #UD
  - If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.

## 12. Updates to Chapter 6, Volume 2D

Change bars and green text show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference*.

-----  
Changes to this chapter: Update to table 6-4, "Register State Initialization after GETSEC[ENTERACCS]".

## GETSEC[ENTERACCS] – Execute Authenticated Chipset Code

Opcode	Instruction	Description
NP OF 37 (EAX = 2)	GETSEC[ENTERACCS]	Enter authenticated code execution mode. EBX holds the authenticated code module physical base address. ECX holds the authenticated code module size (bytes).

### Description

The GETSEC[ENTERACCS] function loads, authenticates and executes an authenticated code module using an Intel® TXT platform chipset's public key. The ENTERACCS leaf of GETSEC is selected with EAX set to 2 at entry.

There are certain restrictions enforced by the processor for the execution of the GETSEC[ENTERACCS] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled, that is, CR0.CD and CR0.NW bits must be 0.
- For processor packages containing more than one logical processor, CR0.CD is checked to ensure consistency between enabled logical processors.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not already be in authenticated code execution mode as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENDER] instruction without a subsequent exiting using GETSEC[EXITAC]).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by IA32\_APIC\_BASE.BSP (Bit 8).

Failure to conform to the above conditions results in the processor signaling a general protection exception.

Prior to execution of the ENTERACCS leaf, other logical processors, i.e., RLPs, in the platform must be:

- Idle in a wait-for-SIPI state (as initiated by an INIT assertion or through reset for non-BSP designated processors), or
- In the SENTER sleep state as initiated by a GETSEC[SENDER] from the initiating logical processor (ILP).

If other logical processor(s) in the same package are not idle in one of these states, execution of ENTERACCS signals a general protection exception. The same requirement and action applies if the other logical processor(s) of the same package do not have CR0.CD = 0.

A successful execution of ENTERACCS results in the ILP entering an authenticated code execution mode. Prior to reaching this point, the processor performs several checks. These include:

- Establish and check the location and size of the specified authenticated code module to be executed by the processor.
- Inhibit the ILP's response to the external events: INIT, A20M, NMI and SMI.
- Broadcast a message to enable protection of memory and I/O from other processor agents.
- Load the designated code module into an authenticated code execution area.
- Isolate the contents of the authenticated code execution area from further state modification by external agents.
- Authenticate the authenticated code module.
- Initialize the initiating logical processor state based on information contained in the authenticated code module header.
- Unlock the Intel® TXT-capable chipset private configuration space and TPM locality 3 space.



- Begin execution in the authenticated code module at the defined entry point.

The GETSEC[ENTERACCS] function requires two additional input parameters in the general purpose registers EBX and ECX. EBX holds the authenticated code (AC) module physical base address (the AC module must reside below 4 GBytes in physical address space) and ECX holds the AC module size (in bytes). The physical base address and size are used to retrieve the code module from system memory and load it into the internal authenticated code execution area. The base physical address is checked to verify it is on a modulo-4096 byte boundary. The size is verified to be a multiple of 64, that it does not exceed the internal authenticated code execution area capacity (as reported by GETSEC[CAPABILITIES]), and that the top address of the AC module does not exceed 32 bits. An error condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception.

As an integrity check for proper processor hardware operation, execution of GETSEC[ENTERACCS] will also check the contents of all the machine check status registers (as reported by the MSRs IA32\_MCi\_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32\_MCG\_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must not be asserted, indicating that no machine check exception processing is currently in progress. These checks are performed prior to initiating the load of the authenticated code module. Any outstanding valid uncorrectable machine check error condition present in these status registers at this point will result in the processor signaling a general protection violation.

The ILP masks the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. This masking remains active until optionally unmasked by GETSEC[EXITAC] (this defined unmasking behavior assumes GETSEC[ENTERACCS] was not executed by a prior GETSEC[SENDER]). The purpose of this masking control is to prevent exposure to existing external event handlers that may not be under the control of the authenticated code module.

The ILP sets an internal flag to indicate it has entered authenticated code execution mode. The state of the A20M pin is likewise masked and forced internally to a de-asserted state so that any external assertion is not recognized during authenticated code execution mode.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) access and I/O originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Prior to launching the authenticated execution module using GETSEC[ENTERACCS] or GETSEC[SENDER], the processor's MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (writeback). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. If the processor detected this requirement is not met, it will signal an Intel® TXT reset condition with an error code during the loading of the authenticated code module.

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default).

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

At the successful completion of GETSEC[ENTERACCS], the architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDTBasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDTLimit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.

A summary of the register state initialization after successful completion of GETSEC[ENTERACCS] is given for the processor in Table 6-4. The paging is disabled upon entry into authenticated code execution mode. The authenticated code module is loaded and initially executed using physical addresses. It is up to the system software after execution of GETSEC[ENTERACCS] to establish a new (or restore its previous) paging environment with an appropriate mapping to meet new protection requirements. EBP is initialized to the authenticated code module base physical address for initial execution in the authenticated environment. As a result, the authenticated code can reference EBP for relative address based references, given that the authenticated code module must be position independent.

**Table 6-4. Register State Initialization after GETSEC[ENTERACCS]**

Register State	Initialization Status	Comment
CRO	PG←0, AM←0, WP←0: Others unchanged	Paging, Alignment Check, Write-protection are disabled.
CR4	MCE←0, CET←0, PCIDE←0: Others unchanged	Machine Check Exceptions, Control-flow Enforcement Technology, and Process-context Identifiers disabled.
EFLAGS	00000002H	
IA32_EFER	0H	IA-32e mode disabled.
EIP	AC.base + EntryPoint	AC.base is in EBX as input to GETSEC[ENTERACCS].
[E R]BX	Pre-ENTERACCS state: Next [E R]IP prior to GETSEC[ENTERACCS]	Carry forward 64-bit processor state across GETSEC[ENTERACCS].
ECX	Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel	Carry forward processor state across GETSEC[ENTERACCS].
[E R]DX	Pre-ENTERACCS state: GDTR base	Carry forward 64-bit processor state across GETSEC[ENTERACCS].
EBP	AC.base	
CS	Sel=[SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	
DS	Sel=[SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	
GDTR	Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]	
DR7	00000400H	
IA32_DEBUGCTL	0H	
IA32_MISC_ENABLE	See Table 6-5 for example.	The number of initialized fields may change due to processor implementation.
Performance counters and counter control registers	0H	

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.

The MSR IA32\_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGS, and the MSR IA32\_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

Performance related counters and counter control registers are cleared as part of execution of ENTERACCS. This implies any active performance counters at any time of ENTERACCS execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

The IA32\_MISC\_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (See the footnote for Table 6-5.). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general-purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[SEXIT].

**Table 6-5. IA32\_MISC\_ENABLE MSR Initialization<sup>1</sup> by ENTERACCS and SENTER**

Field	Bit position	Description
Fast strings enable	0	Clear to 0.
FOPCODE compatibility mode enable	2	Clear to 0.
Thermal monitor enable	3	Set to 1 if other thermal monitor capability is not enabled. <sup>2</sup>
Split-lock disable	4	Clear to 0.
Bus lock on cache line splits disable	8	Clear to 0.
Hardware prefetch disable	9	Clear to 0.
GV1/2 legacy enable	15	Clear to 0.
MONITOR/MWAIT s/m enable	18	Clear to 0.
Adjacent sector prefetch disable	19	Clear to 0.

**NOTES:**

1. The number of IA32\_MISC\_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended

that a new IDT context be established before this point. Until such a new IDT context is established, the programmer must take care in not executing an INT n instruction or any other operation that would result in an exception or trap signaling.

Prior to completion of the GETSEC[ENTERACCS] instruction and after successful authentication of the AC module, the private configuration space of the Intel TXT chipset is unlocked. The authenticated code module alone can gain access to this normally restricted chipset state for the purpose of securing the platform.

Once the authenticated code module is launched at the completion of GETSEC[ENTERACCS], it is free to enable interrupts by setting EFLAGS.IF and enable NMI by execution of IRET. This presumes that it has re-established interrupt handling support through initialization of the IDT, GDT, and corresponding interrupt handling code.

### Operation in a Uni-Processor Platform

(\* The state of the internal flag ACMODEFLAG persists across instruction boundary \*)

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF (IA32_MCG[I]_STATUS = uncorrectable error)
            THEN #GP(0);
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE := EBX;
ACSIZE := ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0) or (ACSIZE < minimum module size) OR (ACSIZE > authenticated RAM
capacity)) or ((ACBASE+ACSIZE) > (2^32 - 1)))
    THEN #GP(0);
IF (secondary thread(s) CR0.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE := (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M := 0;
IA32_DEBUGCTL := 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;
ACMODEFLAG := 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);

```

```

IF (AC module header version isnot supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
(* Authenticate the AC Module and shutdown with an error if it fails *)
KEY := GETKEY(ACRAM, ACBASE);
KEYHASH := HASH(KEY);
CSKEYHASH := READ(TXT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE := DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] := SIGNATURE[I];
COMPUTEDSIGNATURE := HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] := COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL := ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN ACEntryPoint := ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint := ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size))) THEN TXT-SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
CRO.[PG.AM.WP] := 0;
CR4.MCE := 0;
EFLAGS := 00000002h;
IA32_EFER := 0h;
[E|R]BX := [E|R]IP of the instruction after GETSEC[ENTERACCS];
ECX := Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[E|R]DX := Pre-GETSEC[ENTERACCS] GDT.base;
EBP := ACBASE;
GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT := ACRAM[GDTLimit];
CS.SEL := ACRAM[SegSel];
CS.BASE := 0;
CS.LIMIT := FFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+8;
DS.BASE := 0;

```

```

DS.LIMIT := FFFFFFFh;
DS.G := 1;
DS.D := 1;
DS.AR := 93h;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP := ACEntryPoint;
END;

```

### Flags Affected

All flags are cleared.

### Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

### Protected Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If a Intel® TXT-capable chipset is not present.</p> <p>If in VMX root operation.</p> <p>If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.</p> <p>If the processor is already in authenticated code execution mode.</p> <p>If the processor is in SMM.</p> <p>If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.</p> <p>If the authenticated code base is not on a 4096 byte boundary.</p> <p>If the authenticated code size &gt; processor internal authenticated code area capacity.</p> <p>If the authenticated code size is not modulo 64.</p> <p>If other enabled logical processor(s) of the same package CR0.CD = 1.</p> <p>If other enabled logical processor(s) of the same package are not in the wait-for-SIPI or SENTER sleep state.</p>

### Real-Address Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[ENTERACCS] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

- #UD If CR4.SMXE = 0.  
If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].
- #GP(0) GETSEC[ENTERACCS] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

- #GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

- #GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

### VM-exit Condition

- Reason (GETSEC) IF in VMX non-root operation.

### 13. Updates to Chapter 7, Volume 2D

Changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D: Instruction Set Reference*.

-----

Changes to this chapter:

There are three updates made to various instructions that are categorized as either typo corrections or cross-reference updates. These instruction pages are not included in this change document as their impact is evident.

- Updated instructions that were missing “(r)” after EVEX.vvvv or VEX.vvvv: VRCP28SD and VRCP28SS.
- Updated instructions incorrectly using “EVEX.LIG” to use “EVEX.LLIG”: VRCP28SD, VRCP28SS, VRSQRT28SD, and VRSQRT28SS.
- Updated all instructions that list a specific exceptions type to use a cross-reference in order to allow users to jump directly to the applicable exception section for the instruction.



## 14. Updates to Chapter 1, Volume 3A

Change bars and green text show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Changes to this chapter: Updated section 1.1 "Intel® 64 and IA-32 Processors Covered in this Manual". Updated Section 1.2 "Overview of The SYSTEM PROGRAMMING GUIDE" to delete the chapters that have been removed and relabeled the existing chapters that followed. The chapters removed are: Chapter 19, "Performance Monitoring Events", Chapter 31, "Virtual-Machine Monitor Programming Considerations", Chapter 32, "Virtualization of System Resources", and Chapter 33, "Handling Boundary Conditions in a Virtual Machine Monitor".

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1* (order number 253668), the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2* (order number 253669), the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3* (order number 326019), and the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4* (order number 332831) are part of a set that describes the architecture and programming environment of Intel 64 and IA-32 Architecture processors. The other volumes in this set are:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (order number 253665).
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- *The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* address the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

## 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series

## ABOUT THIS MANUAL

- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Core™2 Extreme QX9000 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes.
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family
- 8th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series

- Intel® Xeon® E processors
- 9th generation Intel® Core™ processors
- 2nd generation Intel® Xeon® Processor Scalable Family
- 10th generation Intel® Core™ processors
- 11th generation Intel® Core™ processors
- 3rd generation Intel® Xeon® Processor Scalable Family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Airmont microarchitecture.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor C series, the Intel® Atom™ processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Processor Scalable Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Processor Scalable Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual's content follows<sup>1</sup>:

**Chapter 1 — About This Manual.** Gives an overview of all eight volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — System Architecture Overview.** Describes the modes of operation used by Intel 64 and IA-32 processors and the mechanisms provided by the architectures to support operating systems and executives, including the system-oriented registers and data structures and the system-oriented instructions. The steps necessary for switching between real-address and protected modes are also identified.

**Chapter 3 — Protected-Mode Memory Management.** Describes the data structures, registers, and instructions that support segmentation and paging. The chapter explains how they can be used to implement a "flat" (unsegmented) memory model or a segmented memory model.

**Chapter 4 — Paging.** Describes the paging modes supported by Intel 64 and IA-32 processors.

---

1. Model-Specific Registers have been moved out of this volume and into a separate volume: *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

**Chapter 5 — Protection.** Describes the support for page and segment protection provided in the Intel 64 and IA-32 architectures. This chapter also explains the implementation of privilege rules, stack switching, pointer validation, user and supervisor modes.

**Chapter 6 — Interrupt and Exception Handling.** Describes the basic interrupt mechanisms defined in the Intel 64 and IA-32 architectures, shows how interrupts and exceptions relate to protection, and describes how the architecture handles each exception type. Reference information for each exception is given in this chapter. Includes programming the LINT0 and LINT1 inputs and gives an example of how to program the LINT0 and LINT1 pins for specific interrupt vectors.

**Chapter 7 — Task Management.** Describes mechanisms the Intel 64 and IA-32 architectures provide to support multitasking and inter-task protection.

**Chapter 8 — Multiple-Processor Management.** Describes the instructions and flags that support multiple processors with shared memory, memory ordering, and Intel® Hyper-Threading Technology. Includes MP initialization for P6 family processors and gives an example of how to use the MP protocol to boot P6 family processors in an MP system.

**Chapter 9 — Processor Management and Initialization.** Defines the state of an Intel 64 or IA-32 processor after reset initialization. This chapter also explains how to set up an Intel 64 or IA-32 processor for real-address mode operation and protected- mode operation, and how to switch between modes.

**Chapter 10 — Advanced Programmable Interrupt Controller (APIC).** Describes the programming interface to the local APIC and gives an overview of the interface between the local APIC and the I/O APIC. Includes APIC bus message formats and describes the message formats for messages transmitted on the APIC bus for P6 family and Pentium processors.

**Chapter 11 — Memory Cache Control.** Describes the general concept of caching and the caching mechanisms supported by the Intel 64 or IA-32 architectures. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory. Information on using the new cache control and memory streaming instructions introduced with the Pentium III, Pentium 4, and Intel Xeon processors is also given.

**Chapter 12 — Intel® MMX™ Technology System Programming.** Describes those aspects of the Intel® MMX™ technology that must be handled and considered at the system programming level, including: task switching, exception handling, and compatibility with existing system environments.

**Chapter 13 — System Programming For Instruction Set Extensions And Processor Extended States.** Describes the operating system requirements to support SSE/SSE2/SSE3/SSSE3/SSE4 extensions, including task switching, exception handling, and compatibility with existing system environments. The latter part of this chapter describes the extensible framework of operating system requirements to support processor extended states. Processor extended state may be required by instruction set extensions beyond those of SSE/SSE2/SSE3/SSSE3/SSE4 extensions.

**Chapter 14 — Power and Thermal Management.** Describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

**Chapter 15 — Machine-Check Architecture.** Describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, and P6 family processors. Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

**Chapter 16 — Interpreting Machine-Check Error Codes.** Gives an example of how to interpret the error codes for a machine-check error that occurred on a P6 family processor.

**Chapter 17 — Debug, Branch Profile, TSC, and Resource Monitoring Features.** Describes the debugging registers and other debug mechanism provided in Intel 64 or IA-32 processors. This chapter also describes the time-stamp counter.

**Chapter 18 — Performance Monitoring.** Describes the Intel 64 and IA-32 architectures' facilities for monitoring performance.

**Chapter 19 — 8086 Emulation.** Describes the real-address and virtual-8086 modes of the IA-32 architecture.

**Chapter 20 — Mixing 16-Bit and 32-Bit Code.** Describes how to mix 16-bit and 32-bit code modules within the same program or task.

**Chapter 21 — IA-32 Architecture Compatibility.** Describes architectural compatibility among IA-32 processors.

- **Chapter 22 — Introduction to Virtual Machine Extensions.** Describes the basic elements of virtual machine architecture and the virtual machine extensions for Intel 64 and IA-32 Architectures.
- **Chapter 23 — Virtual Machine Control Structures.** Describes components that manage VMX operation. These include the working-VMCS pointer and the controlling-VMCS pointer.
- **Chapter 24 — VMX Non-Root Operation.** Describes the operation of a VMX non-root operation. Processor operation in VMX non-root mode can be restricted programmatically such that certain operations, events or conditions can cause the processor to transfer control from the guest (running in VMX non-root mode) to the monitor software (running in VMX root mode).
- **Chapter 25 — VM Entries.** Describes VM entries. VM entry transitions the processor from the VMM running in VMX root-mode to a VM running in VMX non-root mode. VM-Entry is performed by the execution of VMLAUNCH or VMRESUME instructions.
- **Chapter 26 — VM Exits.** Describes VM exits. Certain events, operations or situations while the processor is in VMX non-root operation may cause VM-exit transitions. In addition, VM exits can also occur on failed VM entries.
- **Chapter 27 — VMX Support for Address Translation.** Describes virtual-machine extensions that support address translation and the virtualization of physical memory.
- **Chapter 28 — APIC Virtualization and Virtual Interrupts.** Describes the VMCS including controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).
- **Chapter 29 — VMX Instruction Reference.** Describes the virtual-machine extensions (VMX). VMX is intended for a system executive to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.
- **Chapter 30 — System Management Mode.** Describes Intel 64 and IA-32 architectures' system management mode (SMM) facilities.
- **Chapter 31 — Intel® Processor Trace.** Describes details of Intel® Processor Trace.
- **Chapter 32 — Introduction to Intel® Software Guard Extensions.** Provides an overview of the Intel® Software Guard Extensions (Intel® SGX) set of instructions.
- **Chapter 33 — Enclave Access Control and Data Structures.** Describes Enclave Access Control procedures and defines various Intel SGX data structures.
- **Chapter 34 — Enclave Operation.** Describes enclave creation and initialization, adding pages and measuring an enclave, and enclave entry and exit.
- **Chapter 35 — Enclave Exiting Events.** Describes enclave-exiting events (EEE) and asynchronous enclave exit (AEX).
- **Chapter 36 — SGX Instruction References.** Describes the supervisor and user level instructions provided by Intel SGX.
- **Chapter 37 — Intel® SGX Interactions with IA32 and Intel® 64 Architecture.** Describes the Intel SGX collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel 64 architectures.
- **Chapter 38 — Enclave Code Debug and Profiling.** Describes enclave code debug processes and options.
- **Appendix A — VMX Capability Reporting Facility.** Describes the VMX capability MSRs. Support for specific VMX features is determined by reading capability MSRs.
- **Appendix B — Field Encoding in VMCS.** Enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).
- **Appendix C — VM Basic Exit Reasons.** Describes the 32-bit fields that encode reasons for a VM exit. Examples of exit reasons include, but are not limited to: software interrupts, processor exceptions, software traps, NMIs, external interrupts, and triple faults.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.



### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

### 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

#### NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

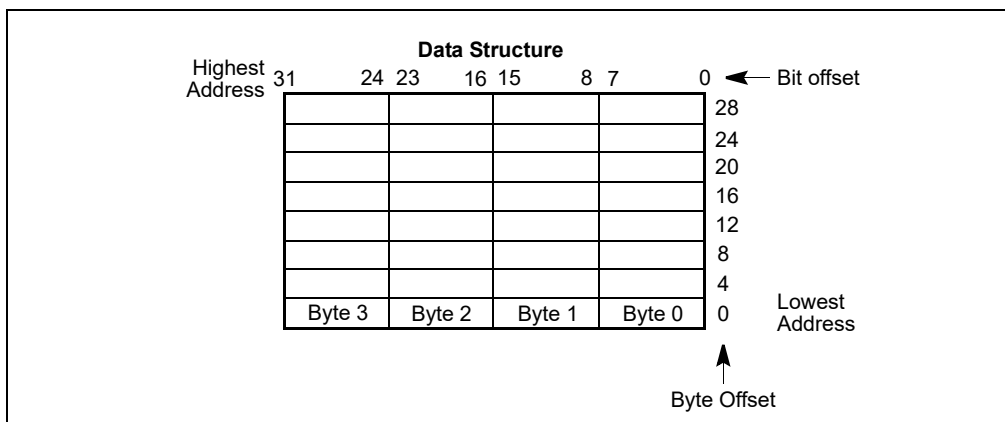


Figure 1-1. Bit and Byte Order



### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

```
CS:EIP
```

### 1.3.6 Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a single syntax to represent this type of information. See Figure 1-2.

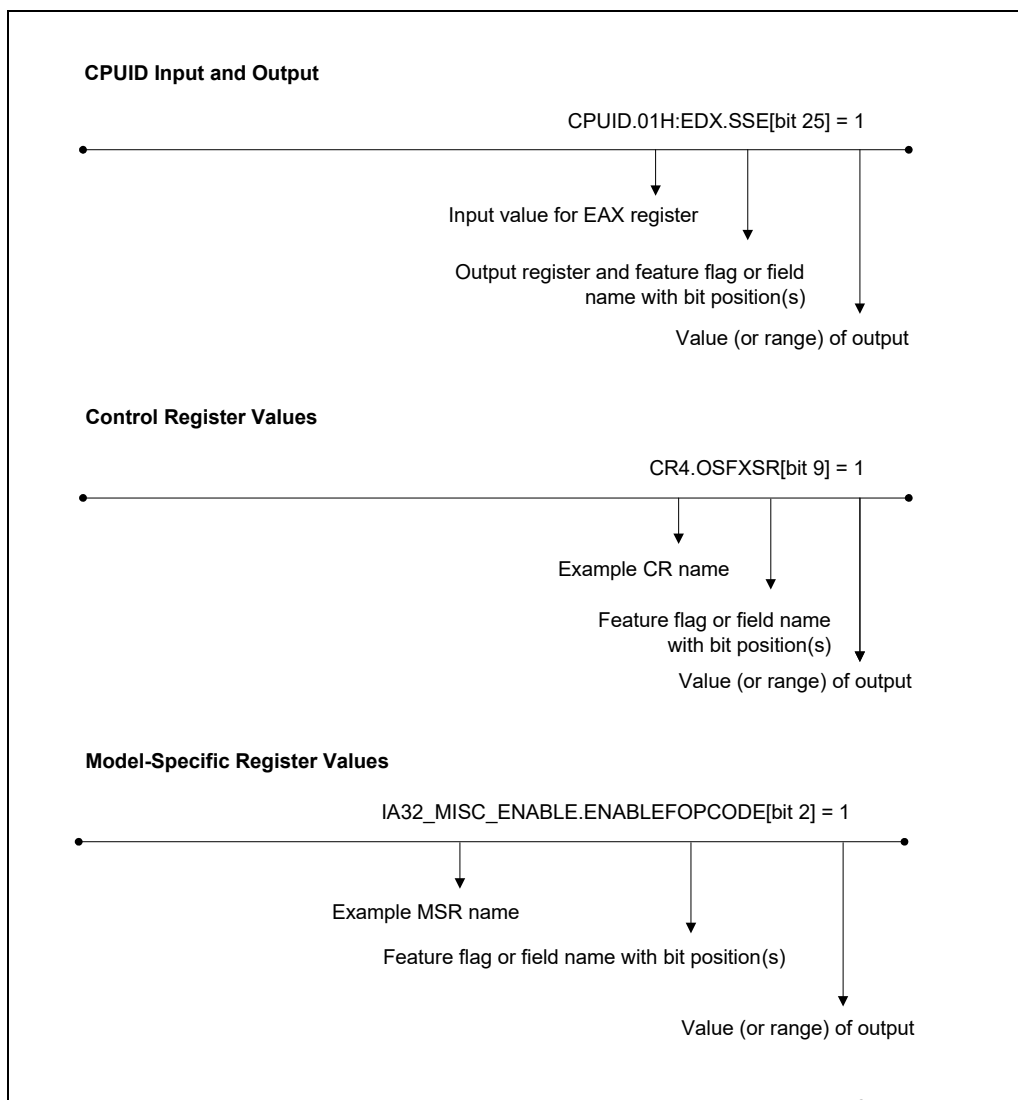


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

### 1.3.7 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

<https://software.intel.com/en-us/articles/intel-sdm>

See also:

- The latest security information on Intel® products:  
<https://www.intel.com/content/www/us/en/security-center/default.html>
- Software developer resources, guidance and insights for security advisories:  
<https://software.intel.com/security-software-guidance/>
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Software Development Tools:  
<https://software.intel.com/en-us/intel-sdp-home>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):  
<https://software.intel.com/en-us/articles/intel-sdm>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<https://software.intel.com/en-us/articles/intel-sdm#optimization>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Intel® Software Guard Extensions (Intel® SGX) Information  
<https://software.intel.com/en-us/isa-extensions/intel-sgx>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
<https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf>
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<https://software.intel.com/sites/default/files/22/30/25602>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference  
<https://software.intel.com/en-us/isa-extensions>

More relevant links are:

- Intel® Developer Zone:  
<https://software.intel.com/en-us>
- Developer centers:  
<http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>

## 15. Updates to Chapter 2, Volume 3A

Change bars and green text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Changes to this chapter: Updates to section 2.5, "Control Registers".

IA-32 architecture (beginning with the Intel386 processor family) provides extensive support for operating-system and system-development software. This support offers multiple modes of operation, which include:

- Real mode, protected mode, virtual 8086 mode, and system management mode. These are sometimes referred to as legacy modes.

Intel 64 architecture supports almost all the system programming facilities available in IA-32 architecture and extends them to a new operating mode (IA-32e mode) that supports a 64-bit programming environment. IA-32e mode allows software to operate in one of two sub-modes:

- 64-bit mode supports 64-bit OS and 64-bit applications
- Compatibility mode allows most legacy software to run; it co-exists with 64-bit applications under a 64-bit OS.

The IA-32 system-level architecture includes features to assist in the following operations:

- Memory management
- Protection of software modules
- Multitasking
- Exception and interrupt handling
- Multiprocessing
- Cache management
- Hardware resource and power management
- Debugging and performance monitoring

This chapter provides a description of each part of this architecture. It also describes the system registers that are used to set up and control the processor at the system level and gives a brief overview of the processor's system-level (operating system) instructions.

Many features of the system-level architecture are used only by system programmers. However, application programmers may need to read this chapter and the following chapters in order to create a reliable and secure environment for application programs.

This overview and most subsequent chapters of this book focus on protected-mode operation of the IA-32 architecture. IA-32e mode operation of the Intel 64 architecture, as it differs from protected mode operation, is also described.

All Intel 64 and IA-32 processors enter real-address mode following a power-up or reset (see Chapter 9, "Processor Management and Initialization"). Software then initiates the switch from real-address mode to protected mode. If IA-32e mode operation is desired, software also initiates a switch from protected mode to IA-32e mode.

## 2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE

System-level architecture consists of a set of registers, data structures, and instructions designed to support basic system-level operations such as memory management, interrupt and exception handling, task management, and control of multiple processors.

Figure 2-1 provides a summary of system registers and data structures that applies to 32-bit modes. System registers and data structures that apply to IA-32e mode are shown in Figure 2-2.

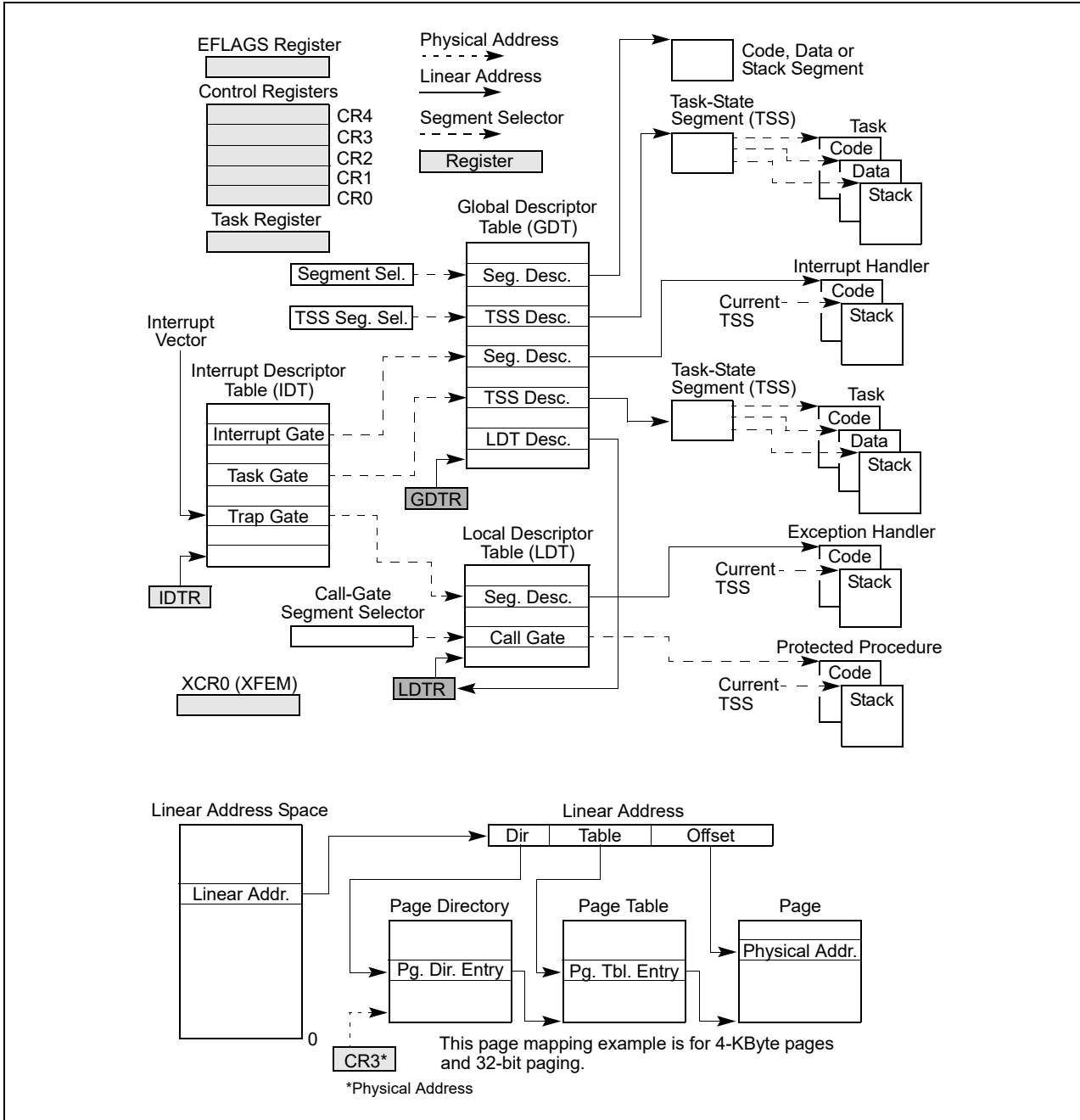


Figure 2-1. IA-32 System-Level Registers and Data Structures

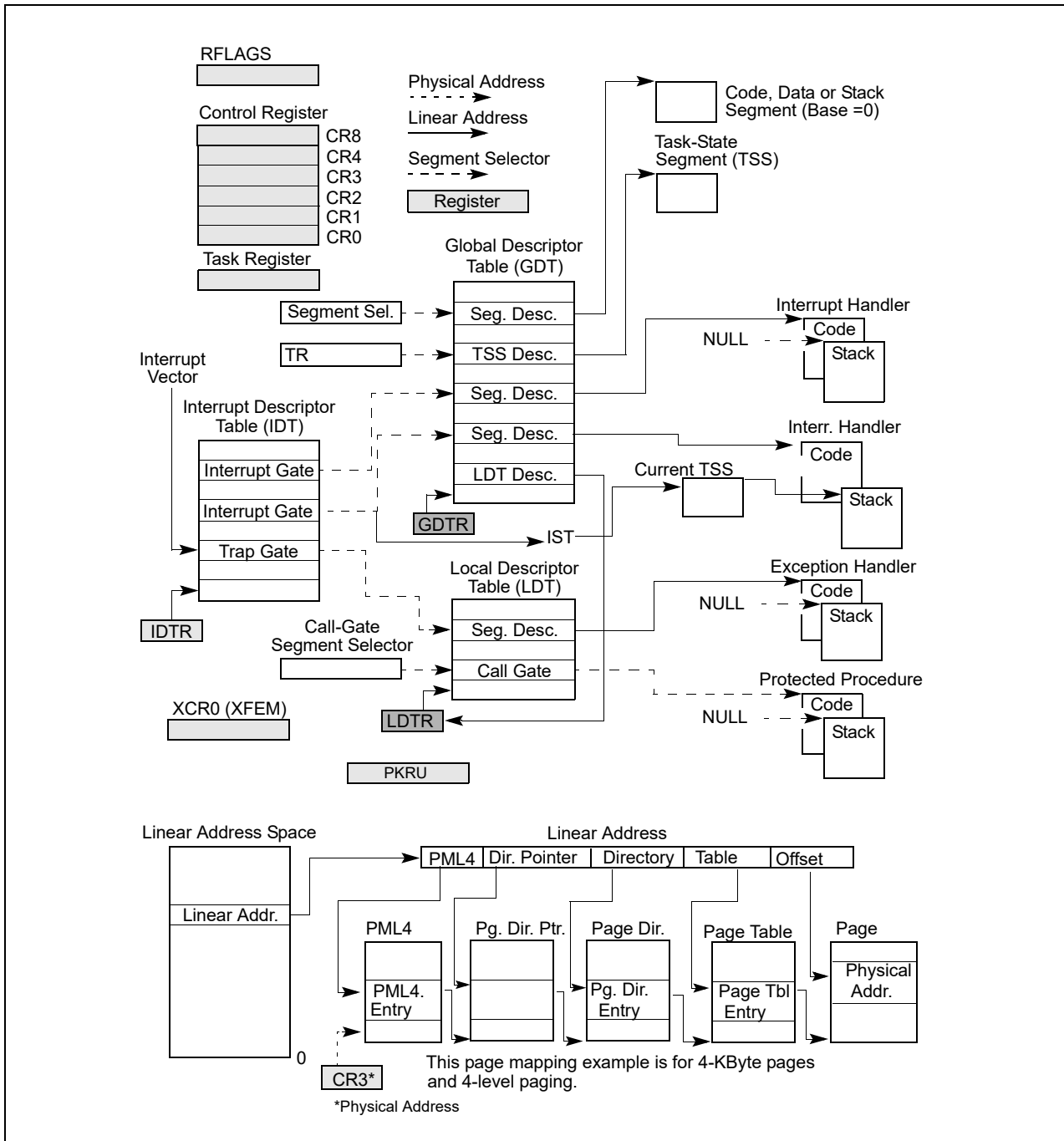


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging

### 2.1.1 Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

To access a byte in a segment, a segment selector and an offset must be supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

See Figure 2-1. The solid arrows in the figure indicate a linear address, dashed lines indicate a segment selector, and the dotted arrows indicate a physical address. For simplicity, many of the segment selectors are shown as direct pointers to a segment. However, the actual path from a segment selector to its associated segment is always through a GDT or LDT.

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR).

### 2.1.1.1 Global and Local Descriptor Tables in IA-32e Mode

GDTR and LDTR registers are expanded to 64-bits wide in both IA-32e sub-modes (64-bit mode and compatibility mode). For more information: see Section 3.5.2, "Segment Descriptor Tables in IA-32e Mode."

Global and local descriptor tables are expanded in 64-bit mode to support 64-bit base addresses, (16-byte LDT descriptors hold a 64-bit base address and various attributes). In compatibility mode, descriptors are not expanded.

## 2.1.2 System Segments, Segment Descriptors, and Gates

Besides code, data, and stack segments that make up the execution environment of a program or procedure, the architecture defines two system segments: the task-state segment (TSS) and the LDT. The GDT is not considered a segment because it is not accessed by means of a segment selector and segment descriptor. TSSs and LDTs have segment descriptors defined for them.

The architecture also defines a set of special descriptors called gates (call gates, interrupt gates, trap gates, and task gates). These provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures. For example, a CALL to a call gate can provide access to a procedure in a code segment that is at the same or a numerically lower privilege level (more privileged) than the current code segment. To access a procedure through a call gate, the calling procedure<sup>1</sup> supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate.

If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task. Gates also facilitate transitions between 16-bit and 32-bit code segments, and vice versa.

### 2.1.2.1 Gates in IA-32e Mode

In IA-32e mode, the following descriptors are 16-byte descriptors (expanded to allow a 64-bit base): LDT descriptors, 64-bit TSSs, call gates, interrupt gates, and trap gates.

Call gates facilitate transitions between 64-bit mode and compatibility mode. Task gates are not supported in IA-32e mode. On privilege level changes, stack segment selectors are not read from the TSS. Instead, they are set to NULL.

---

1. The word "procedure" is commonly used in this document as a general term for a logical unit or block of code (such as a program, procedure, function, or routine).



### 2.1.3 Task-State Segments and Task Gates

The TSS (see Figure 2-1) defines the state of the execution environment for a task. It includes the state of general-purpose registers, segment registers, the EFLAGS register, the EIP register, and segment selectors with stack pointers for three stack segments (one stack for each privilege level). The TSS also includes the segment selector for the LDT associated with the task and the base address of the paging-structure hierarchy.

All program execution in protected mode happens within the context of a task (called the current task). The segment selector for the TSS for the current task is stored in the task register. The simplest method for switching to a task is to make a call or jump to the new task. Here, the segment selector for the TSS of the new task is given in the CALL or JMP instruction. In switching tasks, the processor performs the following actions:

1. Stores the state of the current task in the current TSS.
2. Loads the task register with the segment selector for the new task.
3. Accesses the new TSS through a segment descriptor in the GDT.
4. Loads the state of the new task from the new TSS into the general-purpose registers, the segment registers, the LDTR, control register CR3 (base address of the paging-structure hierarchy), the EFLAGS register, and the EIP register.
5. Begins execution of the new task.

A task can also be accessed through a task gate. A task gate is similar to a call gate, except that it provides access (through a segment selector) to a TSS rather than a code segment.

#### 2.1.3.1 Task-State Segments in IA-32e Mode

Hardware task switches are not supported in IA-32e mode. However, TSSs continue to exist. The base address of a TSS is specified by its descriptor.

A 64-bit TSS holds the following information that is important to 64-bit operation:

- Stack pointer addresses for each privilege level
- Pointer addresses for the interrupt stack table
- Offset address of the IO-permission bitmap (from the TSS base)

The task register is expanded to hold 64-bit base addresses in IA-32e mode. See also: Section 7.7, “Task Management in 64-bit Mode.”

### 2.1.4 Interrupt and Exception Handling

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. Like the GDT, the IDT is not a segment. The linear address for the base of the IDT is contained in the IDT register (IDTR).

Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector from internal hardware, an external interrupt controller, or from software by means of an INT *n*, INTO, INT3, INT1, or BOUND instruction. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

#### 2.1.4.1 Interrupt and Exception Handling IA-32e Mode

In IA-32e mode, interrupt gate descriptors are expanded to 16 bytes to support 64-bit base addresses. This is true for 64-bit mode and compatibility mode.

The IDTR register is expanded to hold a 64-bit base address. Task gates are not supported.

## 2.1.5 Memory Management

System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address. When paging is used: all code, data, stack, and system segments (including the GDT and IDT) can be paged with only the most recently accessed pages being held in physical memory.

The location of pages (sometimes called page frames) in physical memory is contained in the paging structures. These structures reside in physical memory (see Figure 2-1 for the case of 32-bit paging).

The base physical address of the paging-structure hierarchy is contained in control register CR3. The entries in the paging structures determine the physical address of the base of a page frame, access rights and memory management information.

To use this paging mechanism, a linear address is broken into parts. The parts provide separate offsets into the paging structures and the page frame. A system can have a single hierarchy of paging structures or several. For example, each task can have its own hierarchy.

### 2.1.5.1 Memory Management in IA-32e Mode

In IA-32e mode, physical memory pages are managed by a set of system data structures. In both compatibility mode and 64-bit mode, four or five levels of system data structures are used (see Chapter 4, "Paging"). These include the following:

- **The page map level 5 (PML5)** — An entry in the PML5 table contains the physical address of the base of a PML4 table, access rights, and memory management information. The base physical address of the PML5 table is stored in CR3. The PML5 table is used only with 5-level paging.
- **A page map level 4 (PML4)** — An entry in a PML4 table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. With 4-level paging, there is only one PML4 table and its base physical address is stored in CR3.
- **A set of page directory pointer tables** — An entry in a page directory pointer table contains the physical address of the base of a page directory table, access rights, and memory management information.
- **Sets of page directories** — An entry in a page directory table contains the physical address of the base of a page table, access rights, and memory management information.
- **Sets of page tables** — An entry in a page table contains the physical address of a page frame, access rights, and memory management information.

## 2.1.6 System Registers

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the EFLAGS register and several system registers:

- The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights. See also: Section 2.3, "System Flags and Fields in the EFLAGS Register."
- The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system or executive. See also: Section 2.5, "Control Registers" and Section 2.6, "Extended Control Registers (Including XCR0)."
- The debug registers (not shown in Figure 2-1) allow the setting of breakpoints for use in debugging programs and systems software. See also: Chapter 17, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features."
- The GDTR, LDTR, and IDTR registers contain the linear addresses and sizes (limits) of their respective tables. See also: Section 2.4, "Memory-Management Registers."
- The task register contains the linear address and size of the TSS for the current task. See also: Section 2.4, "Memory-Management Registers."
- Model-specific registers (not shown in Figure 2-1).

The model-specific registers (MSRs) are a group of registers available primarily to operating-system or executive procedures (that is, code running at privilege level 0). These registers control items such as the debug extensions, the performance-monitoring counters, the machine-check architecture, and the memory type ranges (MTRRs).

The number and function of these registers varies among different members of the Intel 64 and IA-32 processor families. See also: Section 9.4, “Model-Specific Registers (MSRs),” and Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

Most systems restrict access to system registers (other than the EFLAGS register) by application programs. Systems can be designed, however, where all programs and procedures run at the most privileged level (privilege level 0). In such a case, application programs would be allowed to modify the system registers.

### 2.1.6.1 System Registers in IA-32e Mode

In IA-32e mode, the four system-descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded in hardware to hold 64-bit base addresses. EFLAGS becomes the 64-bit RFLAGS register. CR0–CR4 are expanded to 64 bits. CR8 becomes available. CR8 provides read-write access to the task priority register (TPR) so that the operating system can control the priority classes of external interrupts.

In 64-bit mode, debug registers DR0–DR7 are 64 bits. In compatibility mode, address-matching in DR0–DR3 is also done at 64-bit granularity.

On systems that support IA-32e mode, the extended feature enable register (IA32\_EFER) is available. This model-specific register controls activation of IA-32e mode and other IA-32e mode operations. In addition, there are several model-specific registers that govern IA-32e mode instructions:

- **IA32\_KERNEL\_GS\_BASE** — Used by SWAPGS instruction.
- **IA32\_LSTAR** — Used by SYSCALL instruction.
- **IA32\_FMASK** — Used by SYSCALL instruction.
- **IA32\_STAR** — Used by SYSCALL and SYSRET instruction.

### 2.1.7 Other System Resources

Besides the system registers and data structures described in the previous sections, system architecture provides the following additional resources:

- Operating system instructions (see also: Section 2.8, “System Instruction Summary”).
- Performance-monitoring counters (not shown in Figure 2-1).
- Internal caches and buffers (not shown in Figure 2-1).

Performance-monitoring counters are event counters that can be programmed to count processor events such as the number of instructions decoded, the number of interrupts received, or the number of cache loads.

The processor provides several internal caches and buffers. The caches are used to store both data and instructions. The buffers are used to store things like decoded addresses to system and application segments and write operations waiting to be performed. See also: Chapter 11, “Memory Cache Control.”

## 2.2 MODES OF OPERATION

The IA-32 architecture supports three operating modes and one quasi-operating mode:

- **Protected mode** — This is the native operating mode of the processor. It provides a rich set of architectural features, flexibility, high performance and backward compatibility to existing software base.
- **Real-address mode** — This operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode).
- **System management mode (SMM)** — SMM is a standard architectural feature in all IA-32 processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management

interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.

- **Virtual-8086 mode** — In protected mode, the processor supports a quasi-operating mode known as virtual-8086 mode. This mode allows the processor execute 8086 software in a protected, multitasking environment.

Intel 64 architecture supports all operating modes of IA-32 architecture and IA-32e modes:

- **IA-32e mode** — In IA-32e mode, the processor supports two sub-modes: compatibility mode and 64-bit mode. 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GBytes. Compatibility mode allows most legacy protected-mode applications to run unchanged.

Figure 2-3 shows how the processor moves between operating modes.

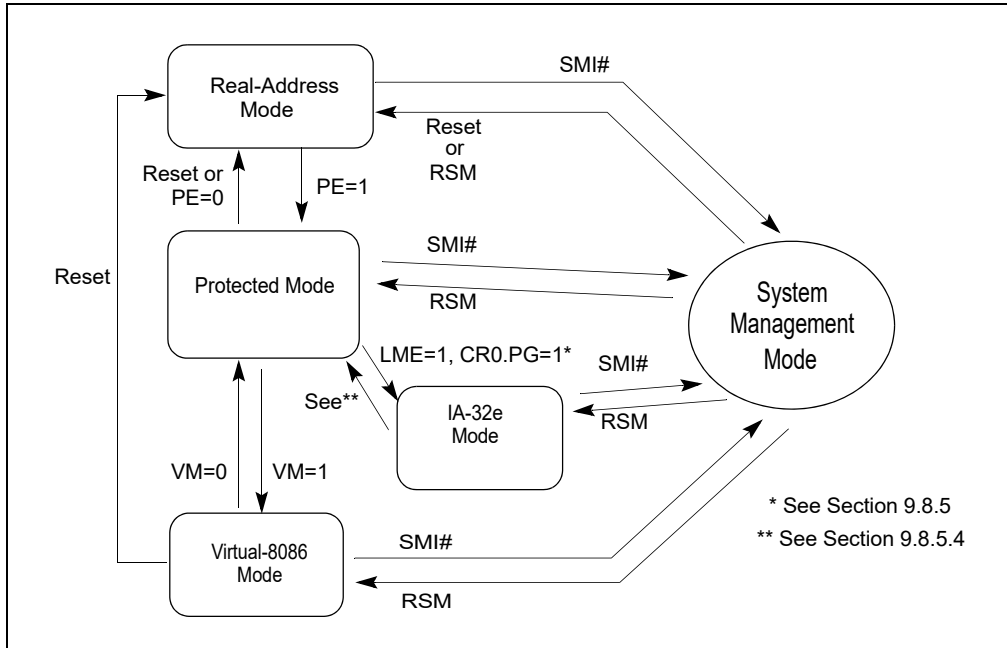


Figure 2-3. Transitions Among the Processor's Operating Modes

The processor is placed in real-address mode following power-up or a reset. The PE flag in control register CR0 then controls whether the processor is operating in real-address or protected mode. See also: Section 9.9, "Mode Switching." and Section 4.1.2, "Paging-Mode Enabling."

The VM flag in the EFLAGS register determines whether the processor is operating in protected mode or virtual-8086 mode. Transitions between protected mode and virtual-8086 mode are generally carried out as part of a task switch or a return from an interrupt or exception handler. See also: Section 19.2.5, "Entering Virtual-8086 Mode."

The LMA bit (IA32\_EFER.LMA[bit 10]) determines whether the processor is operating in IA-32e mode. When running in IA-32e mode, 64-bit or compatibility sub-mode operation is determined by CS.L bit of the code segment. The processor enters into IA-32e mode from protected mode by enabling paging and setting the LME bit (IA32\_EFER.LME[bit 8]). See also: Chapter 9, "Processor Management and Initialization."

The processor switches to SMM whenever it receives an SMI while the processor is in real-address, protected, virtual-8086, or IA-32e modes. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

## 2.2.1 Extended Feature Enable Register

The IA32\_EFER MSR provides several fields related to IA-32e mode enabling and operation. It also provides one field that relates to page-access right modification (see Section 4.6, “Access Rights”). The layout of the IA32\_EFER MSR is shown in Figure 2-4.

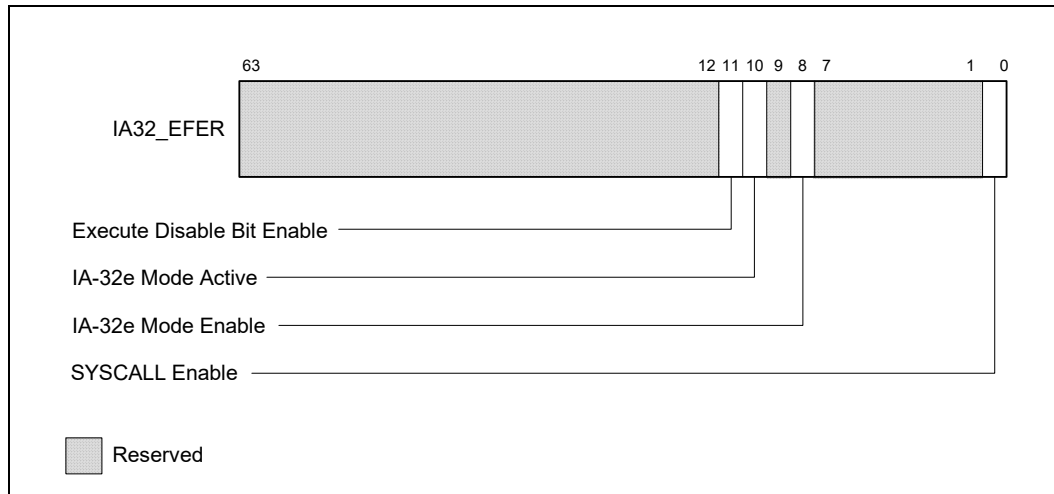


Figure 2-4. IA32\_EFER MSR Layout

Table 2-1. IA32\_EFER MSR Information

Bit	Description
0	<b>SYSCALL Enable: IA32_EFER.SCE (R/W)</b> Enables SYSCALL/SYSRET instructions in 64-bit mode.
7:1	Reserved.
8	<b>IA-32e Mode Enable: IA32_EFER.LME (R/W)</b> Enables IA-32e mode operation.
9	Reserved.
10	<b>IA-32e Mode Active: IA32_EFER.LMA (R)</b> Indicates IA-32e mode is active when set.
11	<b>Execute Disable Bit Enable: IA32_EFER.NXE (R/W)</b> Enables page access restriction by preventing instruction fetches from PAE pages with the XD bit set (See Section 4.6).
63:12	Reserved.

## 2.3 SYSTEM FLAGS AND FIELDS IN THE EFLAGS REGISTER

The system flags and IOPL field of the EFLAGS register control I/O, maskable hardware interrupts, debugging, task switching, and the virtual-8086 mode (see Figure 2-5). Only privileged code (typically operating system or executive code) should be allowed to modify these bits.

The system flags and IOPL are:

TF **Trap (bit 8)** — Set to enable single-step mode for debugging; clear to disable single-step mode. In single-step mode, the processor generates a debug exception after each instruction. This allows the execution state of a program to be inspected after each instruction. If an application program sets the TF flag using a

POPF, POPFD, or IRET instruction, a debug exception is generated after the instruction that follows the POPF, POPFD, or IRET.

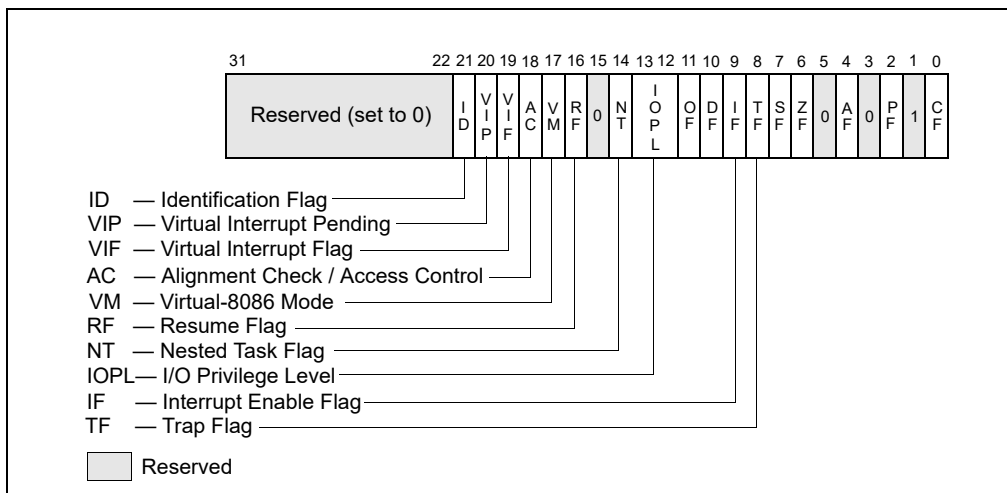


Figure 2-5. System Flags in the EFLAGS Register

**IF Interrupt enable (bit 9)** — Controls the response of the processor to maskable hardware interrupt requests (see also: Section 6.3.2, “Maskable Hardware Interrupts”). The flag is set to respond to maskable hardware interrupts; cleared to inhibit maskable hardware interrupts. The IF flag does not affect the generation of exceptions or nonmaskable interrupts (NMI interrupts). The CPL, IOPL, and the state of the VME flag in control register CR4 determine whether the IF flag can be modified by the CLI, STI, POPF, POPFD, and IRET.

**IOPL I/O privilege level field (bits 12 and 13)** — Indicates the I/O privilege level (IOPL) of the currently running program or task. The CPL of the currently running program or task must be less than or equal to the IOPL to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.

The IOPL is also one of the mechanisms that controls the modification of the IF flag and the handling of interrupts in virtual-8086 mode when virtual mode extensions are in effect (when CR4.VME = 1). See also: Chapter 19, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

**NT Nested task (bit 14)** — Controls the chaining of interrupted and called tasks. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. It examines and modifies this flag on returns from a task initiated with the IRET instruction. The flag can be explicitly set or cleared with the POPF/POPFD instructions; however, changing to the state of this flag can generate unexpected exceptions in application programs.

See also: Section 7.4, “Task Linking.”

**RF Resume (bit 16)** — Controls the processor’s response to instruction-breakpoint conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints (although other exception conditions can cause an exception to be generated). When clear, instruction breakpoints will generate debug exceptions.

The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debug software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with IRETD (to prevent the instruction breakpoint from causing another debug exception). The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.

See also: Section 17.3.1.1, “Instruction-Breakpoint Exception Condition.”

**VM Virtual-8086 mode (bit 17)** — Set to enable virtual-8086 mode; clear to return to protected mode.

See also: Section 19.2.1, “Enabling Virtual-8086 Mode.”

- AC Alignment check or access control (bit 18)** — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. An alignment-check exception is generated when reference is made to an unaligned operand, such as a word at an odd byte address or a doubleword at an address which is not an integral multiple of four. Alignment-check exceptions are generated only in user mode (privilege level 3). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate this exception even when caused by instructions executed in user-mode.
- The alignment-check exception can be used to check alignment of data. This is useful when exchanging data with processors which require all data to be aligned. The alignment-check exception can also be used by interpreters to flag some pointers as special by misaligning the pointer. This eliminates overhead of checking each pointer and only handles the special pointer when used.
- If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1. See Section 4.6, “Access Rights.”
- VIF Virtual Interrupt (bit 19)** — Contains a virtual image of the IF flag. This flag is used in conjunction with the VIP flag. The processor only recognizes the VIF flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. (The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.)
- See also: Section 19.3.3.5, “Method 6: Software Interrupt Handling,” and Section 19.4, “Protected-Mode Virtual Interrupts.”
- VIP Virtual interrupt pending (bit 20)** — Set by software to indicate that an interrupt is pending; cleared to indicate that no interrupt is pending. This flag is used in conjunction with the VIF flag. The processor reads this flag but never modifies it. The processor only recognizes the VIP flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.
- See Section 19.3.3.5, “Method 6: Software Interrupt Handling,” and Section 19.4, “Protected-Mode Virtual Interrupts.”
- ID Identification (bit 21)** — The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

### 2.3.1 System Flags and Fields in IA-32e Mode

In 64-bit mode, the RFLAGS register expands to 64 bits with the upper 32 bits reserved. System flags in RFLAGS (64-bit mode) or EFLAGS (compatibility mode) are shown in Figure 2-5.

In IA-32e mode, the processor does not allow the VM bit to be set because virtual-8086 mode is not supported (attempts to set the bit are ignored). Also, the processor will not set the NT bit. The processor does, however, allow software to set the NT bit (note that an IRET causes a general protection fault in IA-32e mode if the NT bit is set).

In IA-32e mode, the SYSCALL/SYCRET instructions have a programmable method of specifying which bits are cleared in RFLAGS/EFLAGS. These instructions save/restore EFLAGS/RFLAGS.

## 2.4 MEMORY-MANAGEMENT REGISTERS

The processor provides four memory-management registers (GDTR, LDTR, IDTR, and TR) that specify the locations of the data structures which control segmented memory management (see Figure 2-6). Special instructions are provided for loading and storing these registers.



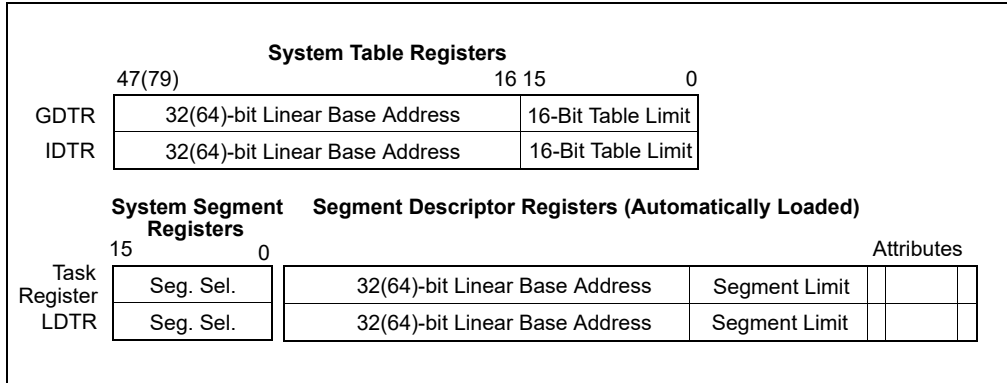


Figure 2-6. Memory Management Registers

### 2.4.1 Global Descriptor Table Register (GDTR)

The GDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and the 16-bit table limit for the GDT. The base address specifies the linear address of byte 0 of the GDT; the table limit specifies the number of bytes in the table.

The LGDT and SGDT instructions load and store the GDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. A new base address must be loaded into the GDTR as part of the processor initialization process for protected-mode operation.

See also: Section 3.5.1, "Segment Descriptor Tables."

### 2.4.2 Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the LDT. The base address specifies the linear address of byte 0 of the LDT segment; the segment limit specifies the number of bytes in the segment. See also: Section 3.5.1, "Segment Descriptor Tables."

The LLDT and SLDT instructions load and store the segment selector part of the LDTR register, respectively. The segment that contains the LDT must have a segment descriptor in the GDT. When the LLDT instruction loads a segment selector in the LDTR: the base address, limit, and descriptor attributes from the LDT descriptor are automatically loaded in the LDTR.

When a task switch occurs, the LDTR is automatically loaded with the segment selector and descriptor for the LDT for the new task. The contents of the LDTR are not automatically saved prior to writing the new LDT information into the register.

On power up or reset of the processor, the segment selector and base address are set to the default value of 0 and the limit is set to 0FFFFH.

### 2.4.3 IDTR Interrupt Descriptor Table Register

The IDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and 16-bit table limit for the IDT. The base address specifies the linear address of byte 0 of the IDT; the table limit specifies the number of bytes in the table. The LIDT and SIDT instructions load and store the IDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. The base address and limit in the register can then be changed as part of the processor initialization process.

See also: Section 6.10, "Interrupt Descriptor Table (IDT)."



## 2.4.4 Task Register (TR)

The task register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the TSS of the current task. The selector references the TSS descriptor in the GDT. The base address specifies the linear address of byte 0 of the TSS; the segment limit specifies the number of bytes in the TSS. See also: Section 7.2.4, “Task Register.”

The LTR and STR instructions load and store the segment selector part of the task register, respectively. When the LTR instruction loads a segment selector in the task register, the base address, limit, and descriptor attributes from the TSS descriptor are automatically loaded into the task register. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH.

When a task switch occurs, the task register is automatically loaded with the segment selector and descriptor for the TSS for the new task. The contents of the task register are not automatically saved prior to writing the new TSS information into the register.

## 2.5 CONTROL REGISTERS

Control registers (CR0, CR1, CR2, CR3, and CR4; see Figure 2-7) determine operating mode of the processor and the characteristics of the currently executing task. These registers are 32 bits in all 32-bit modes and compatibility mode.

In 64-bit mode, control registers are expanded to 64 bits. The MOV CRn instructions are used to manipulate the register bits. Operand-size prefixes for these instructions are ignored. The following is also true:

- The control registers can be read and loaded (or modified) using the move-to-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating-system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.
- Some of the bits in CR0 and CR4 are reserved and must be written with zeros. Attempting to set any reserved bits in CR0[31:0] is ignored. Attempting to set any reserved bits in CR0[63:32] results in a general-protection exception, #GP(0). Attempting to set any reserved bits in CR4 results in a general-protection exception, #GP(0).
- All 64 bits of CR2 are writable by software.
- Reserved bits in CR3[63:MAXPHYADDR] must be zero. Attempting to set any of them results in #GP(0).
- The MOV CR2 instruction does not check that address written to CR2 is canonical.
- A 64-bit capable processor will retain the upper 32 bits of each control register when transitioning out of IA-32e mode.
- On a 64-bit capable processor, an execution of MOV to CR outside of 64-bit mode zeros the upper 32 bits of the control register.
- Register CR8 is available in 64-bit mode only.

The control registers are summarized below, and each architecturally defined control field in these control registers is described individually. In Figure 2-7, the width of the register in 64-bit mode is indicated in parenthesis (except for CR0).

- **CR0** — Contains system control flags that control operating mode and states of the processor.
- **CR1** — Reserved.
- **CR2** — Contains the page-fault linear address (the linear address that caused a page fault).
- **CR3** — Contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor’s internal data caches (they do not control TLB caching of page-directory information).

When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table. With 4-level paging and 5-level paging, the CR3 register contains the base address of the PML4

table and PML5 table, respectively. If PCIDs are enabled, CR3 has a format different from that illustrated in Figure 2-7. See Section 4.5, “4-Level Paging and 5-Level Paging.”

See also: Chapter 4, “Paging.”

- CR4** — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. Bits CR4[63:32] can only be used for IA-32e mode only features that are enabled after entering 64-bit mode. Bits CR4[63:32] do not have any effect outside of IA-32e mode.
- CR8** — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

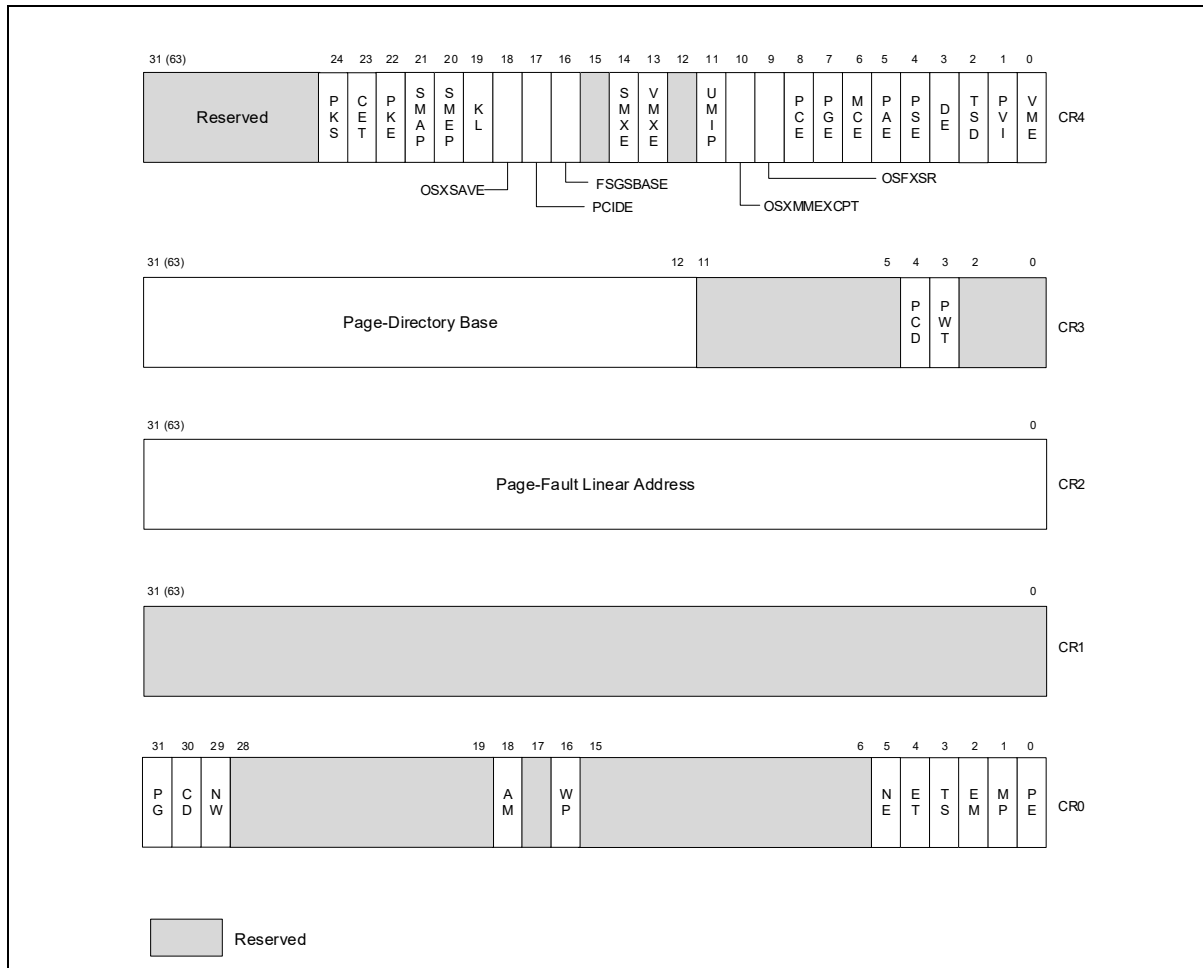


Figure 2-7. Control Registers

The flags in control registers are:

CR0.PG

**Paging (bit 31 of CR0)** — Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP). See also: Chapter 4, “Paging.”

On Intel 64 processors, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.

## CR0.CD

**Cache Disable (bit 30 of CR0)** — When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor's internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 11-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur.

See also: Section 11.5.3, "Preventing Caching," and Section 11.5, "Cache Control."

## CR0.NW

**Not Write-through (bit 29 of CR0)** — When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 11-5 for detailed information about the effect of the NW flag on caching for other settings of the CD and NW flags.

## CR0.AM

**Alignment Mask (bit 18 of CR0)** — Enables automatic alignment checking when set; disables alignment checking when clear. Alignment checking is performed only when the AM flag is set, the AC flag in the EFLAGS register is set, CPL is 3, and the processor is operating in either protected or virtual-8086 mode.

## CR0.WP

**Write Protect (bit 16 of CR0)** — When set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX. This flag must be set before software can set CR4.CET, and it cannot be cleared as long as CR4.CET = 1 (see below).

## CR0.NE

**Numeric Error (bit 5 of CR0)** — Enables the native (internal) mechanism for reporting x87 FPU errors when set; enables the PC-style x87 FPU error reporting mechanism when clear. When the NE flag is clear and the IGNNE# input is asserted, x87 FPU errors are ignored. When the NE flag is clear and the IGNNE# input is deasserted, an unmasked x87 FPU error causes the processor to assert the FERR# pin to generate an external interrupt and to stop instruction execution immediately before executing the next waiting floating-point instruction or WAIT/FWAIT instruction.

The FERR# pin is intended to drive an input to an external interrupt controller (the FERR# pin emulates the ERROR# pin of the Intel 287 and Intel 387 DX math coprocessors). The NE flag, IGNNE# pin, and FERR# pin are used with external logic to implement PC-style error reporting. Using FERR# and IGNNE# to handle floating-point exceptions is deprecated by modern operating systems; this non-native approach also limits newer processors to operate with one logical processor active.

See also: Section 8.7, "Handling x87 FPU Exceptions in Software" in Chapter 8, "Programming with the x87 FPU," and Appendix A, "EFLAGS Cross-Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## CR0.ET

**Extension Type (bit 4 of CR0)** — Reserved in the Pentium 4, Intel Xeon, P6 family, and Pentium processors. In the Pentium 4, Intel Xeon, and P6 family processors, this flag is hardcoded to 1. In the Intel386 and Intel486 processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.

## CR0.TS

**Task Switched (bit 3 of CR0)** — Allows the saving of the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction is actually executed by the new task. The processor sets this flag on every task switch and tests it when executing x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

- If the TS flag is set and the EM flag (bit 2 of CR0) is clear, a device-not-available exception (#NM) is raised prior to the execution of any x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction; with the exception of PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. See the paragraph below for the special case of the WAIT/FWAIT instructions.
- If the TS flag is set and the MP flag (bit 1 of CR0) and EM flag are clear, an #NM exception is not raised prior to the execution of an x87 FPU WAIT/FWAIT instruction.

- If the EM flag is set, the setting of the TS flag has no effect on the execution of x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

Table 2-2 shows the actions taken when the processor encounters an x87 FPU instruction based on the settings of the TS, EM, and MP flags. Table 12-1 and 13-1 show the actions taken when the processor encounters an MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction.

The processor does not automatically save the context of the x87 FPU, XMM, and MXCSR registers on a task switch. Instead, it sets the TS flag, which causes the processor to raise an #NM exception whenever it encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction in the instruction stream for the new task (with the exception of the instructions listed above).

The fault handler for the #NM exception can then be used to clear the TS flag (with the CLTS instruction) and save the context of the x87 FPU, XMM, and MXCSR registers. If the task never encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction, the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context is never saved.

**Table 2-2. Action Taken By x87 FPU Instructions for Different Combinations of EM, MP, and TS**

CRO Flags			x87 FPU Instruction Type	
EM	MP	TS	Floating-Point	WAIT/FWAIT
0	0	0	Execute	Execute.
0	0	1	#NM Exception	Execute.
0	1	0	Execute	Execute.
0	1	1	#NM Exception	#NM exception.
1	0	0	#NM Exception	Execute.
1	0	1	#NM Exception	Execute.
1	1	0	#NM Exception	Execute.
1	1	1	#NM Exception	#NM exception.

**CRO.EM**

**Emulation (bit 2 of CRO)** — Indicates that the processor does not have an internal or external x87 FPU when set; indicates an x87 FPU is present when clear. This flag also affects the execution of MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

When the EM flag is set, execution of an x87 FPU instruction generates a device-not-available exception (#NM). This flag must be set when the processor does not have an internal x87 FPU or is not connected to an external math coprocessor. Setting this flag forces all floating-point instructions to be handled by software emulation. Table 9-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the EM, MP, and TS flags.

Also, when the EM flag is set, execution of an MMX instruction causes an invalid-opcode exception (#UD) to be generated (see Table 12-1). Thus, if an IA-32 or Intel 64 processor incorporates MMX technology, the EM flag must be set to 0 to enable execution of MMX instructions.

Similarly for SSE/SSE2/SSE3/SSSE3/SSE4 extensions, when the EM flag is set, execution of most SSE/SSE2/SSE3/SSSE3/SSE4 instructions causes an invalid opcode exception (#UD) to be generated (see Table 13-1). If an IA-32 or Intel 64 processor incorporates the SSE/SSE2/SSE3/SSSE3/SSE4 extensions, the EM flag must be set to 0 to enable execution of these extensions. SSE/SSE2/SSE3/SSSE3/SSE4 instructions not affected by the EM flag include: PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

**CRO.MP**

**Monitor Coprocessor (bit 1 of CRO)** — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CRO). If the MP flag is set, a WAIT instruction generates a device-not-available exception (#NM) if the TS flag is also set. If the MP flag is clear, the WAIT instruction ignores the setting of the TS flag.

Table 9-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the MP, EM, and TS flags.

## CR0.PE

**Protection Enable (bit 0 of CR0)** — Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set.

See also: Section 9.9, “Mode Switching.”

## CR3.PCD

**Page-level Cache Disable (bit 4 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, “Paging and Memory Typing”. This bit is not used if paging is disabled, with PAE paging, or with 4-level paging<sup>2</sup> or 5-level paging if CR4.PCIDE=1.

## CR3.PWT

**Page-level Write-Through (bit 3 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, “Paging and Memory Typing”. This bit is not used if paging is disabled, with PAE paging, or with 4-level paging or 5-level paging if CR4.PCIDE=1.

## CR4.VME

**Virtual-8086 Mode Extensions (bit 0 of CR4)** — Enables interrupt- and exception-handling extensions in virtual-8086 mode when set; disables the extensions when clear. Use of the virtual mode extensions can improve the performance of virtual-8086 applications by eliminating the overhead of calling the virtual-8086 monitor to handle interrupts and exceptions that occur while executing an 8086 program and, instead, redirecting the interrupts and exceptions back to the 8086 program’s handlers. It also provides hardware support for a virtual interrupt flag (VIF) to improve reliability of running 8086 programs in multi-tasking and multiple-processor environments.

See also: Section 19.3, “Interrupt and Exception Handling in Virtual-8086 Mode.”

## CR4.PVI

**Protected-Mode Virtual Interrupts (bit 1 of CR4)** — Enables hardware support for a virtual interrupt flag (VIF) in protected mode when set; disables the VIF flag in protected mode when clear.

See also: Section 19.4, “Protected-Mode Virtual Interrupts.”

## CR4.TSD

**Time Stamp Disable (bit 2 of CR4)** — Restricts the execution of the RDTSC instruction to procedures running at privilege level 0 when set; allows RDTSC instruction to be executed at any privilege level when clear. This bit also applies to the RDTSCP instruction if supported (if CPUID.8000001H:EDX[27] = 1).

## CR4.DE

**Debugging Extensions (bit 3 of CR4)** — References to debug registers DR4 and DR5 cause an undefined opcode (#UD) exception to be generated when set; when clear, processor aliases references to registers DR4 and DR5 for compatibility with software written to run on earlier IA-32 processors.

See also: Section 17.2.2, “Debug Registers DR4 and DR5.”

## CR4.PSE

**Page Size Extensions (bit 4 of CR4)** — Enables 4-MByte pages with 32-bit paging when set; restricts 32-bit paging to pages of 4 KBytes when clear.

See also: Section 4.3, “32-Bit Paging.”

## CR4.PAE

**Physical Address Extension (bit 5 of CR4)** — When set, enables paging to produce physical addresses with more than 32 bits. When clear, restricts physical addresses to 32 bits. PAE must be set before entering IA-32e mode.

See also: Chapter 4, “Paging.”

## CR4.MCE

**Machine-Check Enable (bit 6 of CR4)** — Enables the machine-check exception when set; disables the machine-check exception when clear.

---

2. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

See also: Chapter 15, “Machine-Check Architecture.”

## CR4.PGE

**Page Global Enable (bit 7 of CR4)** — (Introduced in the P6 family processors.) Enables the global page feature when set; disables the global page feature when clear. The global page feature allows frequently used or shared pages to be marked as global to all users (done with the global flag, bit 8, in a page-directory-pointer-table entry, a page-directory entry, or a page-table entry). Global pages are not flushed from the translation-lookaside buffer (TLB) on a task switch or a write to register CR3.

When enabling the global page feature, paging must be enabled (by setting the PG flag in control register CR0) before the PGE flag is set. Reversing this sequence may affect program correctness, and processor performance will be impacted.

See also: Section 4.10, “Caching Translation Information.”

## CR4.PCE

**Performance-Monitoring Counter Enable (bit 8 of CR4)** — Enables execution of the RDPMC instruction for programs or procedures running at any protection level when set; RDPMC instruction can be executed only at protection level 0 when clear.

## CR4.OSFXSR

**Operating System Support for FXSAVE and FXRSTOR instructions (bit 9 of CR4)** — When set, this flag: (1) indicates to software that the operating system supports the use of the FXSAVE and FXRSTOR instructions, (2) enables the FXSAVE and FXRSTOR instructions to save and restore the contents of the XMM and MXCSR registers along with the contents of the x87 FPU and MMX registers, and (3) enables the processor to execute SSE/SSE2/SSE3/SSSE3/SSE4 instructions, with the exception of the PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

If this flag is clear, the FXSAVE and FXRSTOR instructions will save and restore the contents of the x87 FPU and MMX registers, but they may not save and restore the contents of the XMM and MXCSR registers. Also, the processor will generate an invalid opcode exception (#UD) if it attempts to execute any SSE/SSE2/SSE3 instruction, with the exception of PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. The operating system or executive must explicitly set this flag.

## NOTE

CPUID feature flag FXSR indicates availability of the FXSAVE/FXRSTOR instructions. The OSFXSR bit provides operating system software with a means of enabling FXSAVE/FXRSTOR to save/restore the contents of the X87 FPU, XMM and MXCSR registers. Consequently OSFXSR bit indicates that the operating system provides context switch support for SSE/SSE2/SSE3/SSSE3/SSE4.

## CR4.OSXMMEXCPT

**Operating System Support for Unmasked SIMD Floating-Point Exceptions (bit 10 of CR4)** — When set, indicates that the operating system supports the handling of unmasked SIMD floating-point exceptions through an exception handler that is invoked when a SIMD floating-point exception (#XM) is generated. SIMD floating-point exceptions are only generated by SSE/SSE2/SSE3/SSE4.1 SIMD floating-point instructions.

The operating system or executive must explicitly set this flag. If this flag is not set, the processor will generate an invalid opcode exception (#UD) whenever it detects an unmasked SIMD floating-point exception.

## CR4.UMIP

**User-Mode Instruction Prevention (bit 11 of CR4)** — When set, the following instructions cannot be executed if CPL > 0: SGDT, SIDT, SLDT, SMSW, and STR. An attempt at such execution causes a general-protection exception (#GP).

## CR4.LA57

**57-bit linear addresses (bit 12 of CR4)** — When set in IA-32e mode, the processor uses 5-level paging to translate 57-bit linear addresses. When clear in IA-32e mode, the processor uses 4-level paging to translate 48-bit linear addresses. This bit cannot be modified in IA-32e mode.

See also: Chapter 4, “Paging.”



## CR4.VMXE

**VMX-Enable Bit (bit 13 of CR4)** — Enables VMX operation when set. See Chapter 22, “Introduction to Virtual Machine Extensions.”

## CR4.SMXE

**SMX-Enable Bit (bit 14 of CR4)** — Enables SMX operation when set. See Chapter 6, “Safer Mode Extensions Reference” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*.

## CR4.FSGSBASE

**FSGSBASE-Enable Bit (bit 16 of CR4)** — Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE.

## CR4.PCIDE

**PCID-Enable Bit (bit 17 of CR4)** — Enables process-context identifiers (PCIDs) when set. See Section 4.10.1, “Process-Context Identifiers (PCIDs)”. Applies only in IA-32e mode (if IA32\_EFER.LMA = 1).

## CR4.OSXSAVE

**XSAVE and Processor Extended States-Enable Bit (bit 18 of CR4)** — When set, this flag: (1) indicates (via CPUID.01H:ECX.OSXSAVE[bit 27]) that the operating system supports the use of the XGETBV, XSAVE and XRSTOR instructions by general software; (2) enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), the SSE state (XMM registers and MXCSR), along with other processor extended states enabled in XCR0; (3) enables the processor to execute XGETBV and XSETBV instructions in order to read and write XCR0. See Section 2.6 and Chapter 13, “System Programming for Instruction Set Extensions and Processor Extended States”.

## CR4.KL

**Key-Locker-Enable Bit (bit 19 of CR4)** — When set, the `LOADIWKEY` instruction is enabled; in addition, if support for the AES Key Locker instructions has been activated by system firmware, CPUID.19H:EBX.AESKLE[bit 0] is enumerated as 1 and the AES Key Locker instructions are enabled.<sup>3</sup> When clear, CPUID.19H:EBX.AESKLE[bit 0] is enumerated as 0 and execution of any Key Locker instruction causes an invalid-opcode exception (#UD).

## CR4.SMEP

**SMEP-Enable Bit (bit 20 of CR4)** — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, “Access Rights”.

## CR4.SMAPP

**SMAPP-Enable Bit (bit 21 of CR4)** — Enables supervisor-mode access prevention (SMAPP) when set. See Section 4.6, “Access Rights”.

## CR4.PKE

**Enable protection keys for user-mode pages (bit 22 of CR4)** — 4-level paging and 5-level paging associate each user-mode linear address with a protection key. When set, this flag indicates (via CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4]) that the operating system supports use of the PKRU register to specify, for each protection key, whether user-mode linear addresses with that protection key can be read or written. This bit also enables access to the PKRU register using the `RDPKRU` and `WRPKRU` instructions.

## CR4.CET

**Control-flow Enforcement Technology (bit 23 of CR4)** — Enables control-flow enforcement technology when set. See Chapter 18, “Control-flow Enforcement Technology (CET)” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1*. This flag can be set only if CR0.WP is set, and it must be clear before CR0.WP can be cleared (see below).

## CR4.PKS

**Enable protection keys for supervisor-mode pages (bit 24 of CR4)** — 4-level paging and 5-level paging associate each supervisor-mode linear address with a protection key. When set, this flag allows use of the IA32\_PKRS MSR to specify, for each protection key, whether supervisor-mode linear addresses with that protection key can be read or written.

---

3. Software can check CPUID.19H:EBX.AESKLE[bit 0] after setting CR4.KL to determine whether the AES Key Locker instructions have been enabled. Note that some processors may allow enabling of those instructions without activation by system firmware. Some processors may not support use of the AES Key Locker instructions in system-management mode (SMM). Those processors enumerate CPUID.19H:EBX.AESKLE[bit 0] as 0 in SMM regardless of the setting of CR4.KL.

CR8.TPL

**Task Priority Level (bit 3:0 of CR8)** — This sets the threshold value corresponding to the highest-priority interrupt to be blocked. A value of 0 means all interrupts are enabled. This field is available in 64-bit mode. A value of 15 means all interrupts will be disabled.

### 2.5.1 CPUID Qualification of Control Register Flags

Not all flags in control register CR4 are implemented on all processors. With the exception of the PCE flag, they can be qualified with the CPUID instruction to determine if they are implemented on the processor before they are used.

The CR8 register is available on processors that support Intel 64 architecture.

## 2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0)

If CPUID.01H:ECX.XSAVE[bit 26] is 1, the processor supports one or more **extended control registers** (XCRs). Currently, the only such register defined is XCR0. This register specifies the set of processor state components for which the operating system provides context management, e.g. x87 FPU state, SSE state, AVX state. The OS programs XCR0 to reflect the features for which it provides context management.

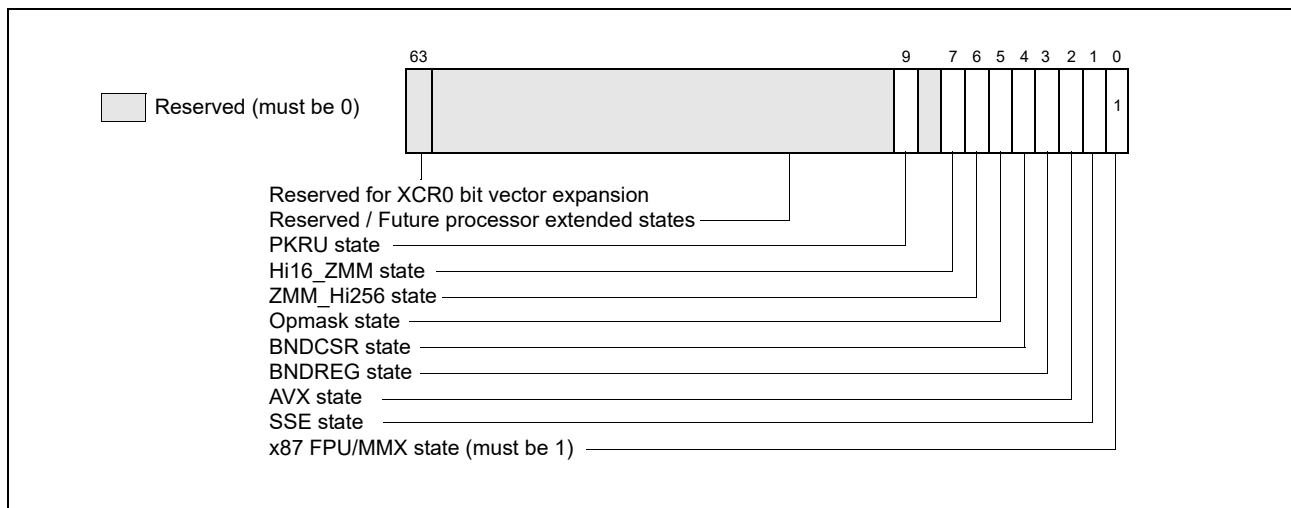


Figure 2-8. XCR0

Software can access XCR0 only if CR4.OSXSAVE[bit 18] = 1. (This bit is also readable as CPUID.01H:ECX.OSXSAVE[bit 27].) Software can use CPUID leaf function 0DH to enumerate the bits in XCR0 that the processor supports (see CPUID instruction in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). Each supported state component is represented by a bit in XCR0. System software enables state components by loading an appropriate bit mask value into XCR0 using the XSETBV instruction.

As each bit in XCR0 (except bit 63) corresponds to a processor state component, XCR0 thus provides support for up to 63 sets of processor state components. Bit 63 of XCR0 is reserved for future expansion and will not represent a processor state component.

Currently, XCR0 defines support for the following state components:

- XCR0.X87 (bit 0): This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.
- XCR0.SSE (bit 1): If 1, the XSAVE feature set can be used to manage MXCSR and the XMM registers (XMM0-XMM15 in 64-bit mode; otherwise XMM0-XMM7).
- XCR0.AVX (bit 2): If 1, AVX instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the YMM registers (YMM0-YMM15 in 64-bit mode; otherwise YMM0-YMM7).



- XCR0.BNDREG (bit 3): If 1, MPX instructions can be executed and the XSAVE feature set can be used to manage the bounds registers BND0–BND3.
- XCR0.BNDCSR (bit 4): If 1, MPX instructions can be executed and the XSAVE feature set can be used to manage the BNDCFGU and BNDSTATUS registers.
- XCR0.opmask (bit 5): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the opmask registers k0–k7.
- XCR0.ZMM\_Hi256 (bit 6): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the lower ZMM registers (ZMM0–ZMM15 in 64-bit mode; otherwise ZMM0–ZMM7).
- XCR0.Hi16\_ZMM (bit 7): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper ZMM registers (ZMM16–ZMM31, only in 64-bit mode).
- XCR0.PKRU (bit 9): If 1, the XSAVE feature set can be used to manage the PKRU register (see Section 2.7).

An attempt to use XSETBV to write to XCR0 results in general-protection exceptions (#GP) if it would do any of the following:

- Set a bit reserved in XCR0 for a given processor (as determined by the contents of EAX and EDX after executing CPUID with EAX=0DH, ECX= 0H).
- Clear XCR0.x87.
- Clear XCR0.SSE and set XCR0.AVX.
- Clear XCR0.AVX and set any of XCR0.opmask, XCR0.ZMM\_Hi256, and XCR0.Hi16\_ZMM.
- Set either XCR0.BNDREG and XCR0.BNDCSR while not setting the other.
- Set any of XCR0.opmask, XCR0.ZMM\_Hi256, and XCR0.Hi16\_ZMM while not setting all of them.

After reset, all bits (except bit 0) in XCR0 are cleared to zero; XCR0[0] is set to 1.

## 2.7 PROTECTION-KEY RIGHTS REGISTERS (PKRU AND IA32\_PKRS)

Processors may support either or both of two protection-key rights registers: PKRU for user-mode pages and the IA32\_PKRS MSR (MSR index 6E1H) for supervisor-mode pages. 4-level paging and 5-level paging associate a 4-bit **protection key** with each page. The protection-key rights registers determine accessibility based on a page's protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, the processor supports the protection-key feature for user-mode pages. When CR4.PKE = 1, software can use the **protection-key rights register for user pages** (PKRU) to specify the access rights for user-mode pages for each protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, the processor supports the protection-key feature for supervisor-mode pages. When CR4.PKS = 1, software can use the **protection-key rights register for supervisor pages** (the IA32\_PKRS MSR) to specify the access rights for supervisor-mode pages for each protection key.

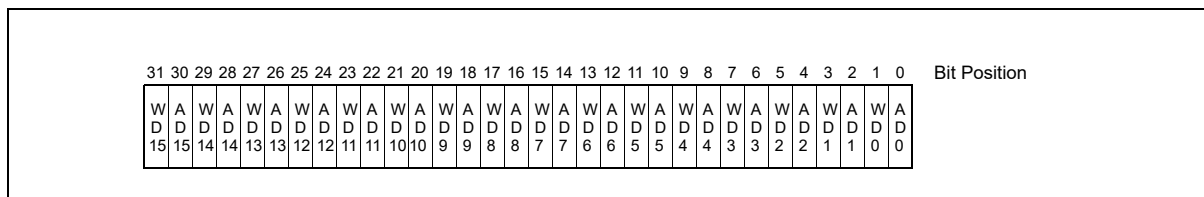


Figure 2-9. Format of Protection-Key Rights Registers

The format of each protection-key rights register is given in Figure 2-9. Each contains 16 pairs of disable controls to prevent data accesses to linear addresses (user-mode or supervisor-mode, depending on the register) based on their protection keys. Each protection key  $i$  ( $0 \leq i \leq 15$ ) is associated with two bits in each protection-key rights register:

- Bit  $2i$ , shown as "AD $i$ " (access disable): if set, the processor prevents any data accesses to linear addresses (user-mode or supervisor-mode, depending on the register) with protection key  $i$ .

- Bit  $2i+1$ , shown as “WDI” (write disable): if set, the processor prevents write accesses to linear addresses (user-mode or supervisor-mode, depending on the register) with protection key  $i$ .

(Bits 63:32 of the IA32\_PKRS MSR are reserved and must be zero.)

See Section 4.6.2, “Protection Keys,” for details of how the processor uses the protection-key rights registers to control accesses to linear addresses.

Software can read and write PKRU using the RDPKRU and WRPKRU instructions. The IA32\_PKRS MSR can be read and written with the RDMSR and WRMSR instructions. Writes to the IA32\_PKRS MSR using WRMSR are not serializing.

## 2.8 SYSTEM INSTRUCTION SUMMARY

System instructions handle system-level functions such as loading system registers, managing the cache, managing interrupts, or setting up the debug registers. Many of these instructions can be executed only by operating-system or executive procedures (that is, procedures running at privilege level 0). Others can be executed at any privilege level and are thus available to application programs.

Table 2-3 lists the system instructions and indicates whether they are available and useful for application programs. These instructions are described in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.

**Table 2-3. Summary of System Instructions**

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	If CR4.UMIP = 1
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	If CR4.UMIP = 1
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	If CR4.UMIP = 1
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	If CR4.UMIP = 1
MOV CR <i>n</i>	Load and store control registers	No	Yes
SMSW	Store MSW	Yes	If CR4.UMIP = 1
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CRO	No	Yes
ARPL	Adjust RPL	Yes <sup>1,5</sup>	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DR <i>n</i>	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No

Table 2-3. Summary of System Instructions (Contd.)

Instruction	Description	Useful to Application?	Protected from Application?
RSM	Return from system management mode	No	Yes
RDMSR <sup>3</sup>	Read Model-Specific Registers	No	Yes
WRMSR <sup>3</sup>	Write Model-Specific Registers	No	Yes
RDPMC <sup>4</sup>	Read Performance-Monitoring Counter	Yes	Yes <sup>2</sup>
RDTSC <sup>3</sup>	Read Time-Stamp Counter	Yes	Yes <sup>2</sup>
RDTSCP <sup>7</sup>	Read Serialized Time-Stamp Counter	Yes	Yes <sup>2</sup>
XGETBV	Return the state of XCRO	Yes	No
XSETBV	Enable one or more processor extended states	No <sup>6</sup>	Yes

**NOTES:**

- Useful to application programs running at a CPL of 1 or 2.
- The TSD and PCE flags in control register CR4 control access to these instructions by application programs running at a CPL of 3.
- These instructions were introduced into the IA-32 Architecture with the Pentium processor.
- This instruction was introduced into the IA-32 Architecture with the Pentium Pro processor and the Pentium processor with MMX technology.
- This instruction is not supported in 64-bit mode.
- Application uses XGETBV to query which set of processor extended states are enabled.
- RDTSCP is introduced in Intel Core i7 processor.

## 2.8.1 Loading and Storing System Registers

The GDTR, LDTR, IDTR, and TR registers each have a load and store instruction for loading data into and storing data from the register:

- **LGDT (Load GDTR Register)** — Loads the GDT base address and limit from memory into the GDTR register.
- **SGDT (Store GDTR Register)** — Stores the GDT base address and limit from the GDTR register into memory.
- **LIDT (Load IDTR Register)** — Loads the IDT base address and limit from memory into the IDTR register.
- **SIDT (Store IDTR Register)** — Stores the IDT base address and limit from the IDTR register into memory.
- **LLDT (Load LDTR Register)** — Loads the LDT segment selector and segment descriptor from memory into the LDTR. (The segment selector operand can also be located in a general-purpose register.)
- **SLDT (Store LDTR Register)** — Stores the LDT segment selector from the LDTR register into memory or a general-purpose register.
- **LTR (Load Task Register)** — Loads segment selector and segment descriptor for a TSS from memory into the task register. (The segment selector operand can also be located in a general-purpose register.)
- **STR (Store Task Register)** — Stores the segment selector for the current task TSS from the task register into memory or a general-purpose register.

The LMSW (load machine status word) and SMSW (store machine status word) instructions operate on bits 0 through 15 of control register CR0. These instructions are provided for compatibility with the 16-bit Intel 286 processor. Programs written to run on 32-bit IA-32 processors should not use these instructions. Instead, they should access the control register CR0 using the MOV CR instruction.

The CLTS (clear TS flag in CR0) instruction is provided for use in handling a device-not-available exception (#NM) that occurs when the processor attempts to execute a floating-point instruction when the TS flag is set. This instruction allows the TS flag to be cleared after the x87 FPU context has been saved, preventing further #NM exceptions. See Section 2.5, “Control Registers,” for more information on the TS flag.

The control registers (CR0, CR1, CR2, CR3, CR4, and CR8) are loaded using the MOV instruction. The instruction loads a control register from a general-purpose register or stores the content of a control register in a general-purpose register.

## 2.8.2 Verifying of Access Privileges

The processor provides several instructions for examining segment selectors and segment descriptors to determine if access to their associated segments is allowed. These instructions duplicate some of the automatic access rights and type checking done by the processor, thus allowing operating-system or executive software to prevent exceptions from being generated.

The ARPL (adjust RPL) instruction adjusts the RPL (requestor privilege level) of a segment selector to match that of the program or procedure that supplied the segment selector. See Section 5.10.4, “Checking Caller Access Privileges (ARPL Instruction)” for a detailed explanation of the function and use of this instruction. Note that ARPL is not supported in 64-bit mode.

The LAR (load access rights) instruction verifies the accessibility of a specified segment and loads access rights information from the segment’s segment descriptor into a general-purpose register. Software can then examine the access rights to determine if the segment type is compatible with its intended use. See Section 5.10.1, “Checking Access Rights (LAR Instruction)” for a detailed explanation of the function and use of this instruction.

The LSL (load segment limit) instruction verifies the accessibility of a specified segment and loads the segment limit from the segment’s segment descriptor into a general-purpose register. Software can then compare the segment limit with an offset into the segment to determine whether the offset lies within the segment. See Section 5.10.3, “Checking That the Pointer Offset Is Within Limits (LSL Instruction)” for a detailed explanation of the function and use of this instruction.

The VERR (verify for reading) and VERW (verify for writing) instructions verify if a selected segment is readable or writable, respectively, at a given CPL. See Section 5.10.2, “Checking Read/Write Rights (VERR and VERW Instructions)” for a detailed explanation of the function and use of these instructions.

## 2.8.3 Loading and Storing Debug Registers

Internal debugging facilities in the processor are controlled by a set of 8 debug registers (DR0-DR7). The MOV instruction allows setup data to be loaded to and stored from these registers.

On processors that support Intel 64 architecture, debug registers DR0-DR7 are 64 bits. In 32-bit modes and compatibility mode, writes to a debug register fill the upper 32 bits with zeros. Reads return the lower 32 bits. In 64-bit mode, the upper 32 bits of DR6-DR7 are reserved and must be written with zeros. Writing one to any of the upper 32 bits causes an exception, #GP(0).

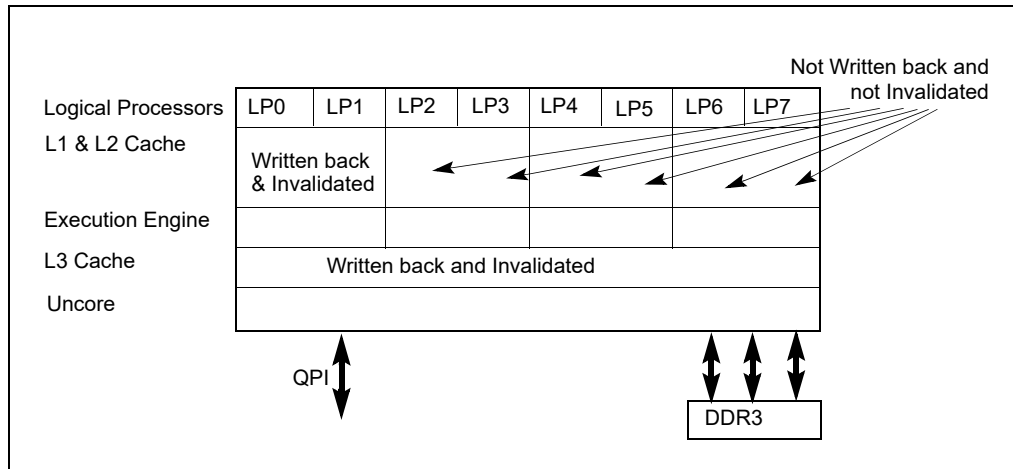
In 64-bit mode, MOV DRn instructions read or write all 64 bits of a debug register (operand-size prefixes are ignored). All 64 bits of DR0-DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0-DR3 are in the limits of the implementation. Address matching is supported only on valid addresses generated by the processor implementation.

## 2.8.4 Invalidating Caches and TLBs

The processor provides several instructions for use in explicitly invalidating its caches and TLB entries. The INVD (invalidate cache with no writeback) instruction invalidates all data and instruction entries in the internal caches and sends a signal to the external caches indicating that they should also be invalidated.

The WBINVD (invalidate cache with writeback) instruction performs the same function as the INVD instruction, except that it writes back modified lines in its internal caches to memory before it invalidates the caches. After invalidating the caches local to the executing logical processor or processor core, WBINVD signals caches higher in the cache hierarchy (caches shared with the invalidating logical processor or core) to write back any data they have in modified state at the time of instruction execution and to invalidate their contents.

Note, non-shared caches may not be written back nor invalidated. In Figure 2-10 below, if code executing on either LP0 or LP1 were to execute a WBINVD, the shared L1 and L2 for LP0/LP1 will be written back and invalidated as will the shared L3. However, the L1 and L2 caches not shared with LP0 and LP1 will not be written back nor invalidated.



**Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy**

The INVLPG (invalidate TLB entry) instruction invalidates (flushes) the TLB entry for a specified page.

## 2.8.5 Controlling the Processor

The HLT (halt processor) instruction stops the processor until an enabled interrupt (such as NMI or SMI, which are normally enabled), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal is received. The processor generates a special bus cycle to indicate that the halt mode has been entered.

Hardware may respond to this signal in a number of ways. An indicator light on the front panel may be turned on. An NMI interrupt for recording diagnostic information may be generated. Reset initialization may be invoked (note that the BINIT# pin was introduced with the Pentium Pro processor). If any non-wake events are pending during shutdown, they will be handled after the wake event from shutdown is processed (for example, A20M# interrupts).

The LOCK prefix invokes a locked (atomic) read-modify-write operation when modifying a memory operand. This mechanism is used to allow reliable communications between processors in multiprocessor systems, as described below:

- In the Pentium processor and earlier IA-32 processors, the LOCK prefix causes the processor to assert the LOCK# signal during the instruction. This always causes an explicit bus lock to occur.
- In the Pentium 4, Intel Xeon, and P6 family processors, the locking operation is handled with either a cache lock or bus lock. If a memory access is cacheable and affects only a single cache line, a cache lock is invoked and the system bus and the actual memory location in system memory are not locked during the operation. Here, other Pentium 4, Intel Xeon, or P6 family processors on the bus write-back any modified data and invalidate their caches as necessary to maintain system memory coherency. If the memory access is not cacheable and/or it crosses a cache line boundary, the processor's LOCK# signal is asserted and the processor does not respond to requests for bus control during the locked operation.

The RSM (return from SMM) instruction restores the processor (from a context dump) to the state it was in prior to a system management mode (SMM) interrupt.

## 2.8.6 Reading Performance-Monitoring and Time-Stamp Counters

The RDPMC (read performance-monitoring counter) and RDTSC (read time-stamp counter) instructions allow application programs to read the processor's performance-monitoring and time-stamp counters, respectively. Processors based on Intel NetBurst® microarchitecture have eighteen 40-bit performance-monitoring counters; P6 family processors have two 40-bit counters. Intel® Atom™ processors and most of the processors based on the Intel Core microarchitecture support two types of performance monitoring counters: programmable performance counters similar to those available in the P6 family, and three fixed-function performance monitoring counters.

Details of programmable and fixed-function performance monitoring counters for each processor generation are described in Chapter 18, “Performance Monitoring”.

The programmable performance counters can support counting either the occurrence or duration of events. Events that can be monitored on programmable counters generally are model specific (except for architectural performance events enumerated by CPUID leaf 0AH); they may include the number of instructions decoded, interrupts received, or the number of cache loads. Individual counters can be set up to monitor different events. Use the system instruction WRMSR to set up values in one of the IA32\_PERFEVTSELx MSR, in one of the 45 ESCRs and one of the 18 CCCR MSRs (for Pentium 4 and Intel Xeon processors); or in the PerfEvtSel0 or the PerfEvtSel1 MSR (for the P6 family processors). The RDPMC instruction loads the current count from the selected counter into the EDX:EAX registers.

Fixed-function performance counters record only specific events that are defined at: <https://perfmon-events.intel.com/>, and the width/number of fixed-function counters are enumerated by CPUID leaf 0AH.

The time-stamp counter is a model-specific 64-bit counter that is reset to zero each time the processor is reset. If not reset, the counter will increment  $\sim 9.5 \times 10^{16}$  times per year when the processor is operating at a clock rate of 3GHz. At this clock frequency, it would take over 190 years for the counter to wrap around. The RDTSC instruction loads the current count of the time-stamp counter into the EDX:EAX registers.

See Section 18.1, “Performance Monitoring Overview,” and Section 17.17, “Time-Stamp Counter,” for more information about the performance monitoring and time-stamp counters.

The RDTSC instruction was introduced into the IA-32 architecture with the Pentium processor. The RDPMC instruction was introduced into the IA-32 architecture with the Pentium Pro processor and the Pentium processor with MMX technology. Earlier Pentium processors have two performance-monitoring counters, but they can be read only with the RDMSR instruction, and only at privilege level 0.

### 2.8.6.1 Reading Counters in 64-Bit Mode

In 64-bit mode, RDTSC operates the same as in protected mode. The count in the time-stamp counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

RDPMC requires an index to specify the offset of the performance-monitoring counter. In 64-bit mode for Pentium 4 or Intel Xeon processor families, the index is specified in ECX[30:0]. The current count of the performance-monitoring counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

## 2.8.7 Reading and Writing Model-Specific Registers

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions allow a processor’s 64-bit model-specific registers (MSRs) to be read and written, respectively. The MSR to be read or written is specified by the value in the ECX register.

RDMSR reads the value from the specified MSR to the EDX:EAX registers; WRMSR writes the value in the EDX:EAX registers to the specified MSR. RDMSR and WRMSR were introduced into the IA-32 architecture with the Pentium processor.

See Section 9.4, “Model-Specific Registers (MSRs),” for more information.

### 2.8.7.1 Reading and Writing Model-Specific Registers in 64-Bit Mode

RDMSR and WRMSR require an index to specify the address of an MSR. In 64-bit mode, the index is 32 bits; it is specified using ECX.

## 2.8.8 Enabling Processor Extended States

The XSETBV instruction is required to enable OS support of individual processor extended states in XCR0 (see Section 2.6).

## 16. Updates to Chapter 6, Volume 3A

Change bars and green text show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Changes to this chapter: Updates to section 6.9, "Prioritization of Concurrent Events".

# CHAPTER 6

## INTERRUPT AND EXCEPTION HANDLING

---

This chapter describes the interrupt and exception-handling mechanism when operating in protected mode on an Intel 64 or IA-32 processor. Most of the information provided here also applies to interrupt and exception mechanisms used in real-address, virtual-8086 mode, and 64-bit mode.

Chapter 19, “8086 Emulation,” describes information specific to interrupt and exception mechanisms in real-address and virtual-8086 mode. Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” describes information specific to interrupt and exception mechanisms in IA-32e mode and 64-bit sub-mode.

### 6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the `INT n` instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor’s interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

### 6.2 EXCEPTION AND INTERRUPT VECTORS

To aid in handling exceptions and interrupts, each architecturally defined exception and each interrupt condition requiring special handling by the processor is assigned a unique identification number, called a vector number. The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT). The table provides the entry point to an exception or interrupt handler (see Section 6.10, “Interrupt Descriptor Table (IDT)”).

The allowable range for vector numbers is 0 to 255. Vector numbers in the range 0 through 31 are reserved by the Intel 64 and IA-32 architectures for architecture-defined exceptions and interrupts. Not all of the vector numbers in this range have a currently defined function. The unassigned vector numbers in this range are reserved. Do not use the reserved vector numbers.

Vector numbers in the range 32 to 255 are designated as user-defined interrupts and are not reserved by the Intel 64 and IA-32 architecture. These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt mechanisms (see Section 6.3, “Sources of Interrupts”).



Table 6-1 shows vector number assignments for architecturally defined exceptions and for the NMI interrupt. This table gives the exception type (see Section 6.5, "Exception Classifications") and indicates whether an error code is saved on the stack for the exception. The source of each predefined exception and the NMI interrupt is also given.

### 6.3 SOURCES OF INTERRUPTS

The processor receives interrupts from two sources:

- External (hardware generated) interrupts.
- Software-generated interrupts.

#### 6.3.1 External Interrupts

External interrupts are received through pins on the processor or through the local APIC. The primary interrupt pins on Pentium 4, Intel Xeon, P6 family, and Pentium processors are the LINT[1:0] pins, which are connected to the local APIC (see Chapter 10, "Advanced Programmable Interrupt Controller (APIC)"). When the local APIC is enabled, the LINT[1:0] pins can be programmed through the APIC's local vector table (LVT) to be associated with any of the processor's exception or interrupt vectors.

When the local APIC is global/hardware disabled, these pins are configured as INTR and NMI pins, respectively. Asserting the INTR pin signals the processor that an external interrupt has occurred. The processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A (see Section 6.2, "Exception and Interrupt Vectors"). Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt vector 2.

**Table 6-1. Protected-Mode Exceptions and Interrupts**

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. <sup>1</sup>
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

**Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)**

Vector	Mnemonic	Description	Type	Error Code	Source
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory. <sup>2</sup>
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent. <sup>3</sup>
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions <sup>4</sup>
20	#VE	Virtualization Exception	Fault	No	EPT violations <sup>5</sup>
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT <i>n</i> instruction.

**NOTES:**

- Processors after the Intel386 processor do not generate this exception.
- This exception was introduced in the Intel486 processor.
- This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
- This exception was introduced in the Pentium III processor.
- This exception can occur only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.

The processor’s local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC’s pins can be directed to the local APIC through the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom®, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors). The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC. These processors have dedicated NMI and INTR pins. With these processors, external interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the INTR pin.

Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 30, “System Management Mode.”

### 6.3.2 Maskable Hardware Interrupts

Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a maskable hardware interrupt. Maskable hardware interrupts that can be delivered through the INTR pin include

all IA-32 architecture defined interrupt vectors from 0 through 255; those that can be delivered through the local APIC include interrupt vectors 16 through 255.

The IF flag in the EFLAGS register permits all maskable hardware interrupts to be masked as a group (see Section 6.8.1, “Masking Maskable Hardware Interrupts”). Note that when interrupts 0 through 15 are delivered through the local APIC, the APIC indicates the receipt of an illegal vector.

### 6.3.3 Software-Generated Interrupts

The INT *n* instruction permits interrupts to be generated from within software by supplying an interrupt vector number as an operand. For example, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.

Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction. If the processor’s predefined NMI vector is used, however, the response of the processor will not be the same as it would be from an NMI interrupt generated in the normal manner. If vector number 2 (the NMI vector) is used in this instruction, the NMI interrupt handler is called, but the processor’s NMI-handling hardware is not activated.

Interrupts generated in software with the INT *n* instruction cannot be masked by the IF flag in the EFLAGS register.

## 6.4 SOURCES OF EXCEPTIONS

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

### 6.4.1 Program-Error Exceptions

The processor generates one or more exceptions when it detects program errors during the execution in an application program or the operating system or executive. Intel 64 and IA-32 architectures define a vector number for each processor-detectable exception. Exceptions are classified as **faults**, **traps**, and **aborts** (see Section 6.5, “Exception Classifications”).

### 6.4.2 Software-Generated Exceptions

The INTO, INT1, INT3, and BOUND instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, INT3 causes a breakpoint exception to be generated.

The INT *n* instruction can be used to emulate exceptions in software; but there is a limitation.<sup>1</sup> If INT *n* provides a vector for one of the architecturally-defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware-generated exception normally produces an error code. The exception handler will still attempt to pop an error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

### 6.4.3 Machine-Check Exceptions

The P6 family and Pentium processors provide both internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions. These mechanisms are implementation depen-

---

1. The INT *n* instruction has opcode CD following by an immediate byte encoding the value of *n*. In contrast, INT1 has opcode F1 and INT3 has opcode CC.

dent. When a machine-check error is detected, the processor signals a machine-check exception (vector 18) and returns an error code.

See Chapter 6, “Interrupt 18—Machine-Check Exception (#MC)” and Chapter 15, “Machine-Check Architecture,” for more information about the machine-check mechanism.

## 6.5 EXCEPTION CLASSIFICATIONS

Exceptions are classified as **faults**, **traps**, or **aborts** depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

- **Faults** — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- **Aborts** — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

### NOTE

One exception subset normally reported as a fault is not restartable. Such exceptions result in loss of some processor state. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. In this situation, the exception handler sees that the instruction pointer (CS:EIP) has been restored as if the POPAD instruction had not been executed. However, internal processor state (the general-purpose registers) will have been modified. Such cases are considered programming errors. An application causing this class of exceptions should be terminated by the operating system.

## 6.6 PROGRAM OR TASK RESTART

To allow the restarting of program or task following the handling of an exception or an interrupt, all exceptions (except aborts) are guaranteed to report exceptions on an instruction boundary. All interrupts are guaranteed to be taken on an instruction boundary.

For fault-class exceptions, the return instruction pointer (saved when the processor generates an exception) points to the faulting instruction. So, when a program or task is restarted following the handling of a fault, the faulting instruction is restarted (re-executed). Restarting the faulting instruction is commonly used to handle exceptions that are generated when access to an operand is blocked. The most common example of this type of fault is a page-fault exception (#PF) that occurs when a program or task references an operand located on a page that is not in memory. When a page-fault exception occurs, the exception handler can load the page into memory and resume execution of the program or task by restarting the faulting instruction. To ensure that the restart is handled transparently to the currently executing program or task, the processor saves the necessary registers and stack pointers to allow a restart to the state prior to the execution of the faulting instruction.

For trap-class exceptions, the return instruction pointer points to the instruction following the trapping instruction. If a trap is detected during an instruction which transfers execution, the return instruction pointer reflects the transfer. For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction. All trap exceptions allow program or task restart with no loss of continuity. For example, the overflow exception is a trap exception. Here, the return instruction pointer points to the instruction following the INTO instruction that tested EFLAGS.OF (overflow) flag. The trap handler for this exception resolves the overflow condition. Upon return from the trap handler, program or task execution continues at the instruction following the INTO instruction.

The abort-class exceptions do not support reliable restarting of the program or task. Abort handlers are designed to collect diagnostic information about the state of the processor when the abort exception occurred and then shut down the application and system as gracefully as possible.

Interrupts rigorously support restarting of interrupted programs and tasks without loss of continuity. The return instruction pointer saved for an interrupt points to the next instruction to be executed at the instruction boundary where the processor took the interrupt. If the instruction just executed has a repeat prefix, the interrupt is taken at the end of the current iteration with the registers set to execute the next iteration.

The ability of a P6 family processor to speculatively execute instructions does not affect the taking of interrupts by the processor. Interrupts are taken at instruction boundaries located during the retirement phase of instruction execution; so they are always taken in the “in-order” instruction stream. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the P6 family processors’ microarchitecture and its support for out-of-order instruction execution.

Note that the Pentium processor and earlier IA-32 processors also perform varying amounts of prefetching and preliminary decoding. With these processors as well, exceptions and interrupts are not signaled until actual “in-order” execution of the instructions. For a given code sample, the signaling of exceptions occurs uniformly when the code is executed on any family of IA-32 processors (except where new exceptions or new opcodes have been defined).

## 6.7 NONMASKABLE INTERRUPT (NMI)

The nonmaskable interrupt (NMI) can be generated in either of two ways:

- External hardware asserts the NMI pin.
- The processor receives a message on the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors) with a delivery mode NMI.

When the processor receives a NMI from either of these sources, the processor handles it immediately by calling the NMI handler pointed to by interrupt vector number 2. The processor also invokes certain hardware conditions to ensure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 6.7.1, “Handling Multiple NMIs”).

Also, when an NMI is received from either of the above sources, it cannot be masked by the IF flag in the EFLAGS register.

It is possible to issue a maskable hardware interrupt (through the INTR pin) to vector 2 to invoke the NMI interrupt handler; however, this interrupt will not truly be an NMI interrupt. A true NMI interrupt that activates the processor’s NMI-handling hardware can only be delivered through one of the mechanisms listed above.

### 6.7.1 Handling Multiple NMIs

While an NMI interrupt handler is executing, the processor blocks delivery of subsequent NMIs until the next execution of the IRET instruction. This blocking of NMIs prevents nested execution of the NMI handler. It is recommended that the NMI interrupt handler be accessed through an interrupt gate to disable maskable hardware interrupts (see Section 6.8.1, “Masking Maskable Hardware Interrupts”).

An execution of the IRET instruction unblocks NMIs even if the instruction causes a fault. For example, if the IRET instruction executes with EFLAGS.VM = 1 and IOPL of less than 3, a general-protection exception is generated (see Section 19.2.7, “Sensitive Instructions”). In such a case, NMIs are unmasked before the exception handler is invoked.

## 6.8 ENABLING AND DISABLING INTERRUPTS

The processor inhibits the generation of some interrupts, depending on the state of the processor and of the IF and RF flags in the EFLAGS register, as described in the following sections.

## 6.8.1 Masking Maskable Hardware Interrupts

The IF flag can disable the servicing of maskable hardware interrupts received on the processor's INTR pin or through the local APIC (see Section 6.3.2, "Maskable Hardware Interrupts"). When the IF flag is clear, the processor inhibits interrupts delivered to the INTR pin or through the local APIC from generating an internal interrupt request; when the IF flag is set, interrupts delivered to the INTR or through the local APIC pin are processed as normal external interrupts.

The IF flag does not affect non-maskable interrupts (NMIs) delivered to the NMI pin or delivery mode NMI messages delivered through the local APIC, nor does it affect processor generated exceptions. As with the other flags in the EFLAGS register, the processor clears the IF flag in response to a hardware reset.

The fact that the group of maskable hardware interrupts includes the reserved interrupt and exception vectors 0 through 32 can potentially cause confusion. Architecturally, when the IF flag is set, an interrupt for any of the vectors from 0 through 32 can be delivered to the processor through the INTR pin and any of the vectors from 16 through 32 can be delivered through the local APIC. The processor will then generate an interrupt and call the interrupt or exception handler pointed to by the vector number. So for example, it is possible to invoke the page-fault handler through the INTR pin (by means of vector 14); however, this is not a true page-fault exception. It is an interrupt. As with the INT *n* instruction (see Section 6.4.2, "Software-Generated Exceptions"), when an interrupt is generated through the INTR pin to an exception vector, the processor does not push an error code on the stack, so the exception handler may not operate correctly.

The IF flag can be set or cleared with the STI (set interrupt-enable flag) and CLI (clear interrupt-enable flag) instructions, respectively. These instructions may be executed only if the CPL is equal to or less than the IOPL. A general-protection exception (#GP) is generated if they are executed when the CPL is greater than the IOPL.<sup>2</sup> If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI.<sup>3</sup> The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag is also affected by the following operations:

- The PUSHF instruction stores all flags on the stack, where they can be examined and modified. The POPF instruction can be used to load the modified flags back into the EFLAGS register.
- Task switches and the POPF and IRET instructions load the EFLAGS register; therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is handled through an interrupt gate, the IF flag is automatically cleared, which disables maskable hardware interrupts. (If an interrupt is handled through a trap gate, the IF flag is not cleared.)

See the descriptions of the CLI, STI, PUSHF, POPF, and IRET instructions in Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and Chapter 4, "Instruction Set Reference, M-U," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a detailed description of the operations these instructions are allowed to perform on the IF flag.

## 6.8.2 Masking Instruction Breakpoints

The RF (resume) flag in the EFLAGS register controls the response of the processor to instruction-breakpoint conditions (see the description of the RF flag in Section 2.3, "System Flags and Fields in the EFLAGS Register").

When set, it prevents an instruction breakpoint from generating a debug exception (#DB); when clear, instruction breakpoints will generate debug exceptions. The primary function of the RF flag is to prevent the processor from going into a debug exception loop on an instruction-breakpoint. See Section 17.3.1.1, "Instruction-Breakpoint Exception Condition," for more information on the use of this flag.

As noted in Section 6.8.3, execution of the MOV or POP instruction to load the SS register suppresses any instruction breakpoint on the next instruction (just as if EFLAGS.RF were 1).

2. The effect of the IOPL on these instructions is modified slightly when the virtual mode extension is enabled by setting the VME flag in control register CR4: see Section 19.3, "Interrupt and Exception Handling in Virtual-8086 Mode." Behavior is also impacted by the PVI flag: see Section 19.4, "Protected-Mode Virtual Interrupts."

3. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.

### 6.8.3 Masking Exceptions and Interrupts When Switching Stacks

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX
MOV ESP, StackTop
```

(Software might also use the POP instruction to load SS and ESP.)

If an interrupt or exception occurs after the new SS segment descriptor has been loaded but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler (assuming that delivery of the interrupt or exception does not itself load a new stack pointer).

To account for this situation, the processor prevents certain events from being delivered after execution of a MOV to SS instruction or a POP to SS instruction. The following items provide details:

- Any instruction breakpoint on the next instruction is suppressed (as if EFLAGS.RF were 1).
- Any data breakpoint on the MOV to SS instruction or POP to SS instruction is inhibited until the instruction boundary following the next instruction.
- Any single-step trap that would be delivered following the MOV to SS instruction or POP to SS instruction (because EFLAGS.TF is 1) is suppressed.
- The suppression and inhibition ends after delivery of an exception or the execution of the next instruction.
- If a sequence of consecutive instructions each loads the SS register (using MOV or POP), only the first is guaranteed to inhibit or suppress events in this way.

Intel recommends that software use the LSS instruction to load the SS register and ESP together. The problem identified earlier does not apply to LSS, and the LSS instruction does not inhibit events as detailed above.

## 6.9 PRIORITIZATION OF CONCURRENT EVENTS

If more than one event is pending at an instruction boundary (between execution of instructions), the processor services them in a predictable order. Table 6-2 shows the priority among classes of event sources.

Table 6-2. Priority Among Concurrent Events

Priority	Description
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check (#MC)
2	Trap on Task Switch - T flag in TSS is set (#DB)
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Trap-class Debug Exceptions (#DB due to TF flag set or data/I-O breakpoint)
5	Nonmaskable Interrupts (NMI) <sup>1</sup>
6	Maskable Hardware Interrupts <sup>1</sup>
7	Fault-class Debug Exceptions (#DB due to instruction breakpoint)



Table 6-2. Priority Among Concurrent Events (Contd.)

Priority	Description
8	Faults from Fetching Next Instruction - Code-Segment Limit Violation (#GP) - Code Page Fault (#PF) - Control protection exception due to missing ENDBRANCH at target of an indirect call or jump (#CP)
9 (Lowest)	Faults from Decoding the Next Instruction - Instruction length > 15 bytes (#GP) - Invalid Opcode (#UD) - Coprocessor Not Available (#NM)

**NOTE**

1. The Intel® 486 processor and earlier processors group nonmaskable and maskable interrupts in the same priority class.

The processor first services a pending **event** from the class which has the highest priority, transferring execution to the first instruction of the handler. Lower priority exceptions are discarded; lower priority interrupts are held pending. Discarded exceptions **may be** re-generated when the event handler returns execution to the point in the program or task where the **original event** occurred. While the priority among the classes listed in Table 6-2 is consistent **across processor implementations**, the **priority of events** within a class is implementation-dependent and may vary from processor to processor.

Table 6-2 specifies the prioritization of events that may be pending at an instruction boundary. It does not specify the prioritization of faults that arise during instruction execution or event delivery (these include #BR, #TS, #NP, #SS, #GP, #PF, #AC, #MF, #XM, #VE, or #CP). It also does not apply to the events generated by the "Call to Interrupt Procedure" instructions (INT n, INTO, INT3, and INT1), as these events are integral to the execution of those instructions and do not occur between instructions.

## 6.10 INTERRUPT DESCRIPTOR TABLE (IDT)

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors (in protected mode). Unlike the GDT, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight (the number of bytes in a gate descriptor). Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 descriptors, because descriptors are required only for the interrupt and exception vectors that may occur. All empty descriptor slots in the IDT should have the present flag for the descriptor set to 0.

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is,  $8N - 1$ ).

The IDT may reside anywhere in the linear address space. As shown in Figure 6-1, the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.

The LIDT (load IDT register) and SIDT (store IDT register) instructions load and store the contents of the IDTR register, respectively. The LIDT instruction loads the IDTR register with the base address and limit held in a memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another. The SIDT instruction copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

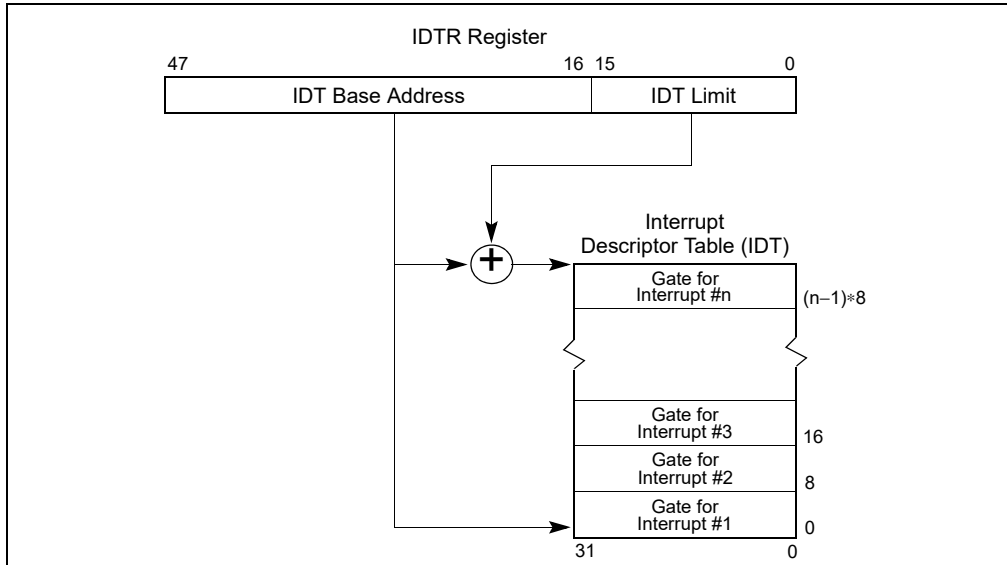
If a vector references a descriptor beyond the limit of the IDT, a general-protection exception (#GP) is generated.



**NOTE**

Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/or the blocking of interrupt delivery.

IA-32 architecture rules need to be followed for setting up IDTR base/limit/access fields and each field in the gate descriptors. The same apply for the Intel 64 architecture. This includes implicit referencing of the destination code segment through the GDT or LDT and accessing the stack.



**Figure 6-1. Relationship of the IDTR and IDT**

## 6.11 IDT DESCRIPTORS

The IDT may contain any of three kinds of gate descriptors:

- Task-gate descriptor
- Interrupt-gate descriptor
- Trap-gate descriptor

Figure 6-2 shows the formats for the task-gate, interrupt-gate, and trap-gate descriptors. The format of a task gate used in an IDT is the same as that of a task gate used in the GDT or an LDT (see Section 7.2.5, “Task-Gate Descriptor”). The task gate contains the segment selector for a TSS for an exception and/or interrupt handler task.

Interrupt and trap gates are very similar to call gates (see Section 5.8.3, “Call Gates”). They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution to a handler procedure in an exception- or interrupt-handler code segment. These gates differ in the way the processor handles the IF flag in the EFLAGS register (see Section 6.12.1.3, “Flag Usage By Exception- or Interrupt-Handler Procedure”).

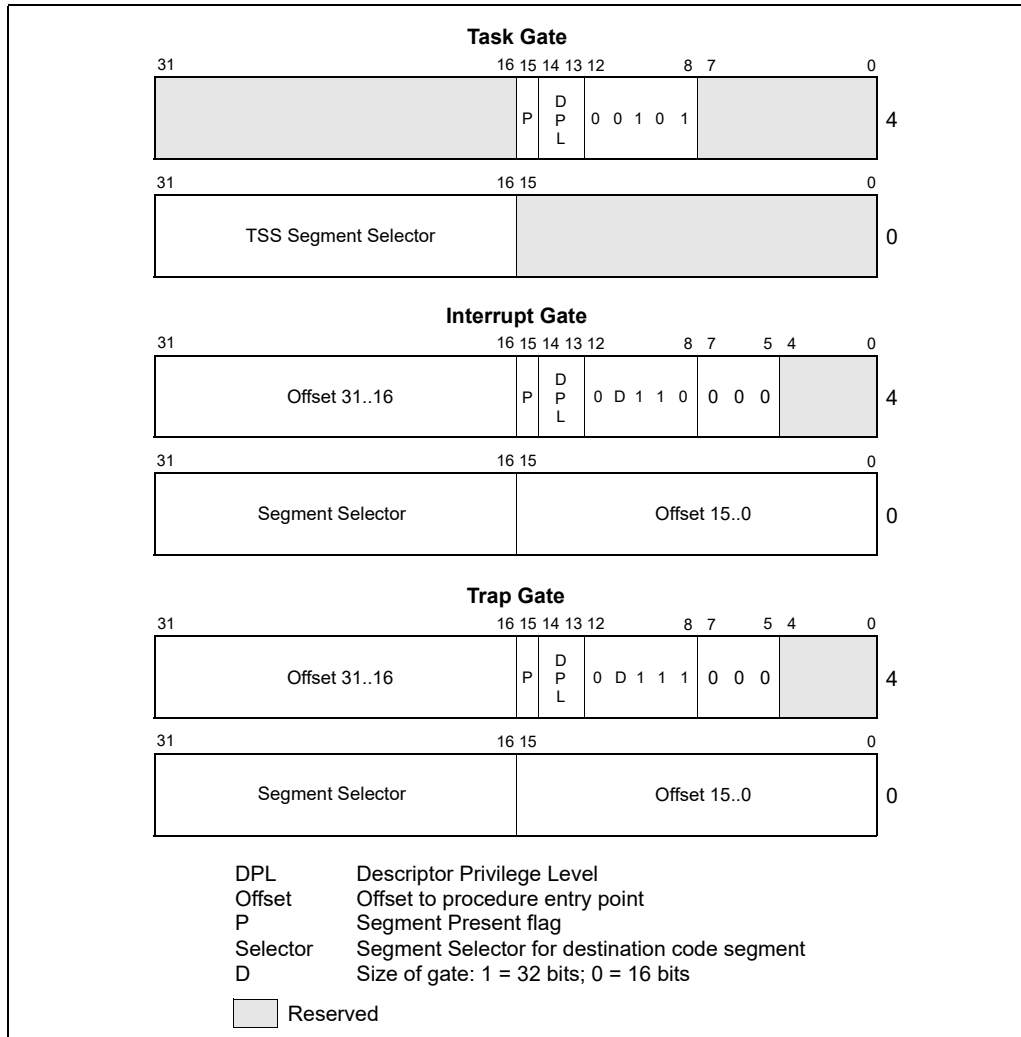


Figure 6-2. IDT Gate Descriptors

## 6.12 EXCEPTION AND INTERRUPT HANDLING

The processor handles calls to exception- and interrupt-handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate (see Section 5.8.2, "Gate Descriptors," through Section 5.8.6, "Returning from a Called Procedure"). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a task gate (see Section 7.3, "Task Switching").

### 6.12.1 Exception- or Interrupt-Handler Procedures

An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of the currently executing task (see Figure 6-3). The segment selector for the gate points to a segment descriptor for an executable code segment in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception- or interrupt-handling procedure.

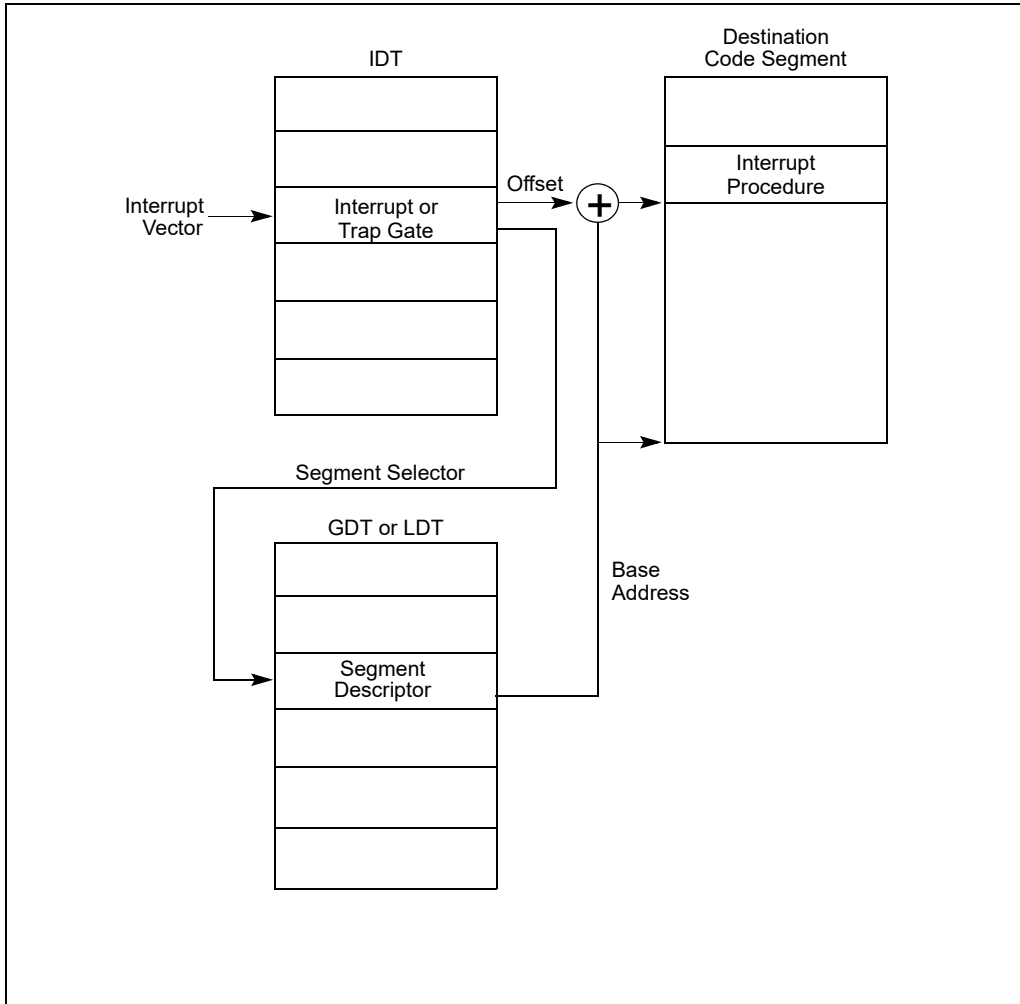


Figure 6-3. Interrupt Procedure Call

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  - a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  - b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figure 6-4).
  - c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  - a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figure 6-4).
  - b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

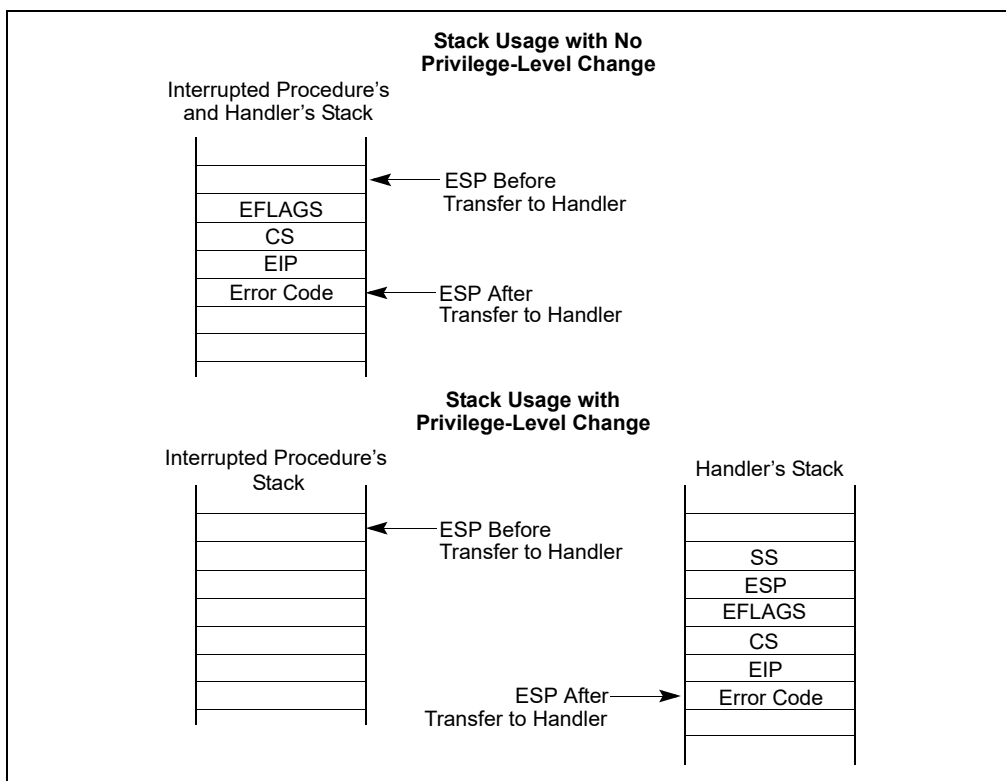


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL is less than or equal to the IOPL. See Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for a description of the complete operation performed by the IRET instruction.

If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure's stack on the return.

### 6.12.1.1 Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a shadow stack switch occurs. When the shadow stack switch occurs:
  - a. On a transfer from privilege level 3, if shadow stacks are enabled at privilege level 3 then the SSP is saved to the IA32\_PL3\_SSP MSR.
  - b. If shadow stacks are enabled at the privilege level where the handler will execute then the shadow stack for the handler is obtained from one of the following MSRs based on the privilege level at which the handler executes.
    - IA32\_PL2\_SSP if handler executes at privilege level 2.
    - IA32\_PL1\_SSP if handler executes at privilege level 1.
    - IA32\_PL0\_SSP if handler executes at privilege level 0.
  - c. The SSP obtained is then verified to ensure it points to a valid supervisory shadow stack that is not currently active by verifying a supervisor shadow stack token at the address pointed to by the SSP. The operations performed to verify and acquire the supervisor shadow stack token by making it busy are as described in Section 18.2.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
  - d. On this new shadow stack, the processor pushes the CS, LIP (CS.base + EIP), and SSP of the interrupted procedure if the interrupted procedure was executing at privilege level less than 3; see Figure 6-5.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure and shadow stacks are enabled at current privilege level:
  - a. The processor saves the current state of the CS, LIP (CS.base + EIP), and SSP registers on the current shadow stack; see Figure 6-5.

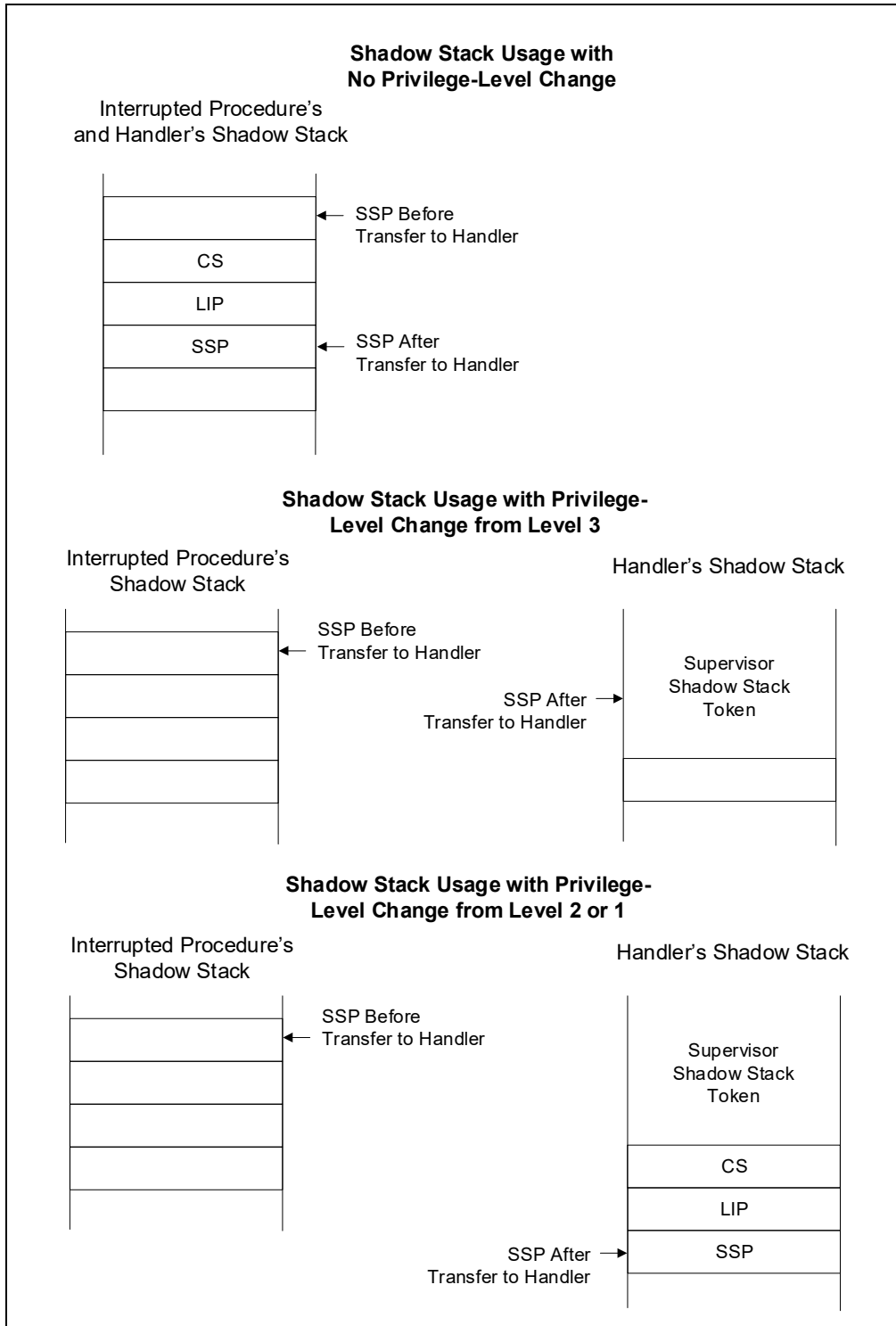


Figure 6-5. Shadow Stack Usage on Transfers to Interrupt and Exception-Handling Routines

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions to enforce return address protection:

- Restores the CS and EIP registers to their values prior to the interrupt or exception.

If shadow stack is enabled:

- Compares the values on shadow stack at address  $SSP+8$  (the LIP) and  $SSP+16$  (the CS) to the CS and  $(CS.base + EIP)$  popped from the stack and causes a control protection exception ( $\#CP(FAR-RET/IRET)$ ) if they do not match.
- Pops the top-of-stack value (the SSP prior to the interrupt or exception) from shadow stack into SSP register.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs the actions below.

- If shadow stack is enabled at current privilege level:
  - If SSP is not aligned to 8 bytes then causes a control protection exception ( $\#CP(FAR-RET/IRET)$ ).
  - If privilege level of the procedure being returned to is less than 3 (returning to supervisor mode):
    - Compares the values on shadow stack at address  $SSP+8$  (the LIP) and  $SSP+16$  (the CS) to the CS and  $(CS.base + EIP)$  popped from the stack and causes a control protection exception ( $\#CP(FAR-RET/IRET)$ ) if they do not match.
    - Temporarily saves the top-of-stack value (the SSP of the procedure being returned to) internally.
  - If a busy supervisor shadow stack token is present at address  $SSP+24$ , then marks the token free using operations described in section Section 18.2.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
  - If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode), restores the SSP register from the internally saved value.
  - If the privilege level of the procedure being returned to is 3 (returning to user mode) and shadow stack is enabled at privilege level 3, then restores the SSP register with value of IA32\_PL3\_SSP MSR.

### 6.12.1.2 Protection of Exception- and Interrupt-Handler Procedures

The privilege-level protection for exception- and interrupt-handler procedures is similar to that used for ordinary procedure calls when called through a call gate (see Section 5.8.4, "Accessing a Code Segment Through a Call Gate"). The processor does not permit transfer of execution to an exception- or interrupt-handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL.

An attempt to violate this rule results in a general-protection exception ( $\#GP$ ). The protection mechanism for exception- and interrupt-handler procedures is different in the following ways:

- Because interrupt and exception vectors have no RPL, the RPL is not checked on implicit calls to exception and interrupt handlers.
- The processor checks the DPL of the interrupt or trap gate only if an exception or interrupt is generated with an  $INT\ n$ ,  $INT3$ , or  $INTO$  instruction.<sup>4</sup> Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using a software interrupt to access critical exception handlers, such as the page-fault handler, providing that those handlers are placed in more privileged code segments (numerically lower privilege level). For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL of interrupt and trap gates.

Because exceptions and interrupts generally do not occur at predictable times, these privilege rules effectively impose restrictions on the privilege levels at which exception and interrupt- handling procedures can run. Either of the following techniques can be used to avoid privilege-level violations.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data from a data segment, the data segment needs to be accessible from privilege level 3, which would make it unprotected.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program or task is running at.

---

4. This check is not performed by execution of the  $INT1$  instruction (opcode  $F1$ ); it would be performed by execution of  $INT\ 1$  (opcode  $CD\ 01$ ).

### 6.12.1.3 Flag Usage By Exception- or Interrupt-Handler Procedure

When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the processor clears the TF flag in the EFLAGS register after it saves the contents of the EFLAGS register on the stack. (On calls to exception and interrupt handlers, the processor also clears the VM, RF, and NT flags in the EFLAGS register, after they are saved on the stack.) Clearing the TF flag prevents instruction tracing from affecting interrupt response and ensures that no single-step exception will be delivered after delivery to the handler. A subsequent IRET instruction restores the TF (and VM, RF, and NT) flags to the values in the saved contents of the EFLAGS register on the stack.

The only difference between an interrupt gate and a trap gate is the way the processor handles the IF flag in the EFLAGS register. When accessing an exception- or interrupt-handling procedure through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to its value in the saved contents of the EFLAGS register on the stack. Accessing a handler procedure through a trap gate does not affect the IF flag.

## 6.12.2 Interrupt Tasks

When an exception or interrupt handler is accessed through a task gate in the IDT, a task switch results. Handling an exception or interrupt with a separate task offers several advantages:

- The entire context of the interrupted program or task is saved automatically.
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

The disadvantage of handling an interrupt with a separate task is that the amount of machine state that must be saved on a task switch makes it slower than using an interrupt gate, resulting in increased interrupt latency.

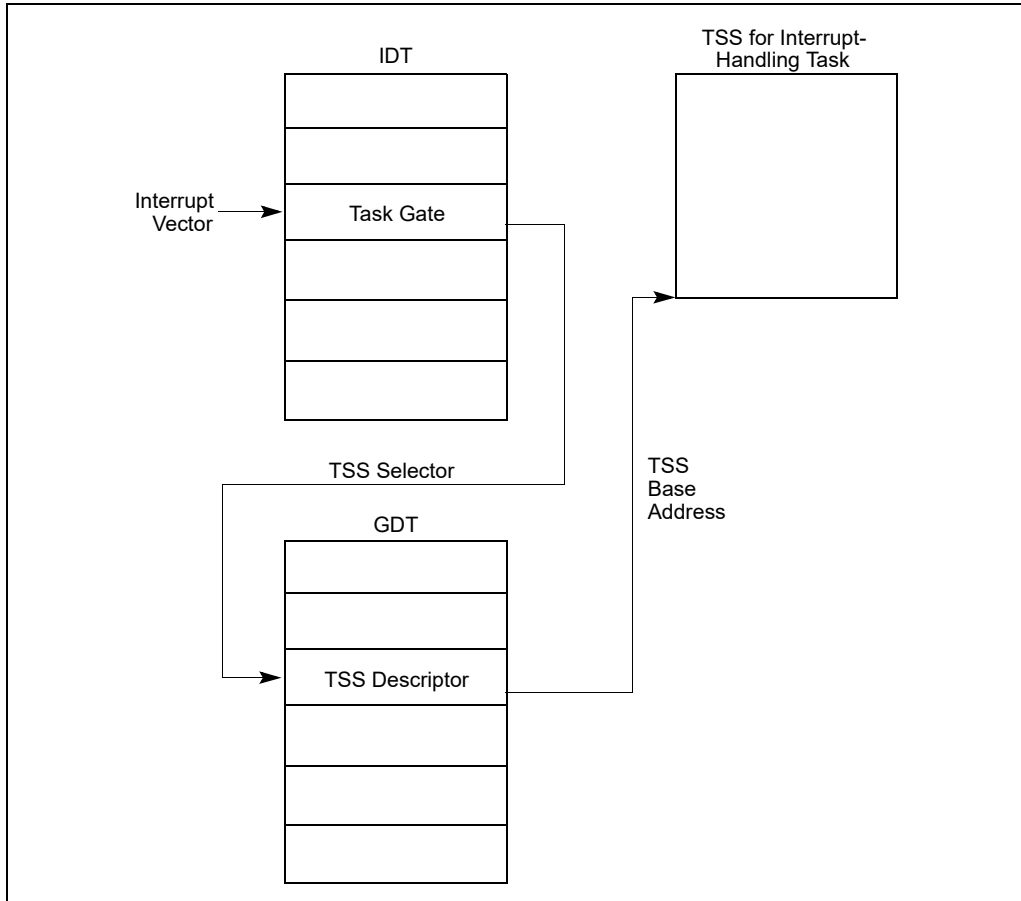
A task gate in the IDT references a TSS descriptor in the GDT (see Figure 6-6). A switch to the handler task is handled in the same manner as an ordinary task switch (see Section 7.3, "Task Switching"). The link back to the interrupted task is stored in the previous task link field of the handler task's TSS. If an exception caused an error code to be generated, this error code is copied to the stack of the new task.

When exception- or interrupt-handler tasks are used in an operating system, there are actually two mechanisms that can be used to dispatch tasks: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor's interrupt mechanism). The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.



**NOTE**

Because IA-32 architecture tasks are not re-entrant, an interrupt-handler task must disable interrupts between the time it completes handling the interrupt and the time it executes the IRET instruction. This action prevents another interrupt from occurring while the interrupt task's TSS is still marked busy, which would cause a general-protection (#GP) exception.



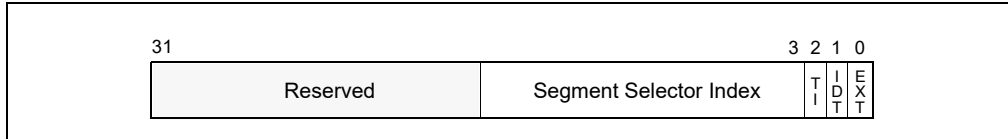
**Figure 6-6. Interrupt Task Switch**

### 6.13 ERROR CODE

When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the format shown in Figure 6-7. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

- EXT**      **External event (bit 0)** — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.<sup>5</sup> The bit is cleared if the exception occurred during delivery of a software interrupt (INT *n*, INT3, or INTO).
- IDT**      **Descriptor location (bit 1)** — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.
- TI**        **GDT/LDT (bit 2)** — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

5. The bit is also set if the exception occurred during delivery of INT1.



**Figure 6-7. Error Code**

The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment selector was referenced in an operation.

The format of the error code is different for page-fault exceptions (#PF). See the “Interrupt 14—Page-Fault Exception (#PF)” section in this chapter.

The format of the error code is different for control protection exceptions (#CP). See the “Interrupt 21—Control Protection Exception (#CP)” section in this chapter.

The error code is pushed on the stack as a doubleword or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for doubleword pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT[1:0] pins) or the INT *n* instruction, even if an error code is normally produced for those exceptions.

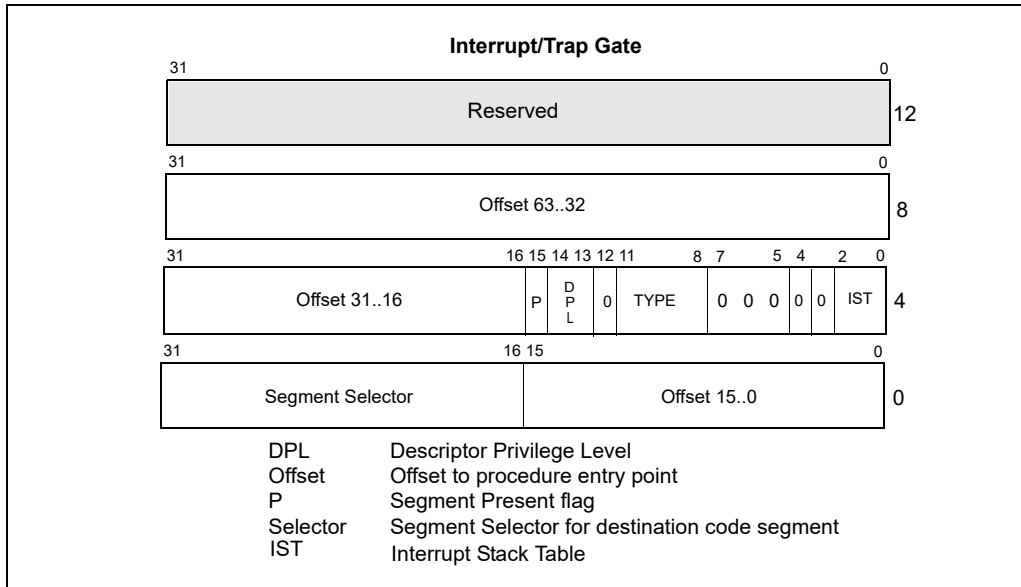
## 6.14 EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE

In 64-bit mode, interrupt and exception handling is similar to what has been described for non-64-bit modes. The following are the exceptions:

- All interrupt handlers pointed by the IDT are in 64-bit code (this does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits; and the processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy modes, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism and a new interrupt shadow stack-switch mechanism.
- The alignment of interrupt stack frame is different.

### 6.14.1 64-Bit Mode IDT

Interrupt and trap gates are 16 bytes in length to provide a 64-bit offset for the instruction pointer (RIP). The 64-bit RIP referenced by interrupt-gate descriptors allows an interrupt service routine to be located anywhere in the linear-address space. See Figure 6-8.



**Figure 6-8. 64-Bit IDT Gate Descriptors**

In 64-bit mode, the IDT index is formed by scaling the interrupt vector by 16. The first eight bytes (bytes 7:0) of a 64-bit mode interrupt gate are similar but not identical to legacy 32-bit interrupt gates. The type field (bits 11:8 in bytes 7:4) is described in Table 3-2. The Interrupt Stack Table (IST) field (bits 4:0 in bytes 7:4) is used by the stack switching mechanisms described in Section 6.14.5, "Interrupt Stack Table." Bytes 11:8 hold the upper 32 bits of the target RIP (interrupt segment offset) in canonical form. A general-protection exception (#GP) is generated if software attempts to reference an interrupt gate with a target RIP that is not in canonical form.

The target code segment referenced by the interrupt gate must be a 64-bit code segment (CS.L = 1, CS.D = 0). If the target is not a 64-bit code segment, a general-protection exception (#GP) is generated with the IDT vector number reported as the error code.

Only 64-bit interrupt and trap gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). Legacy 32-bit interrupt or trap gate types (0EH or 0FH) are redefined in IA-32e mode as 64-bit interrupt and trap gate types. No 32-bit interrupt or trap gate type exists in IA-32e mode. If a reference is made to a 16-bit interrupt or trap gate (06H or 07H), a general-protection exception (#GP(0)) is generated.

### 6.14.2 64-Bit Mode Stack Frame

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes. SS:ESP is pushed only on a CPL change. In 64-bit mode, the size of interrupt stack-frame pushes is fixed at eight bytes. This is because only 64-bit mode gates can be referenced. 64-bit mode also pushes SS:RSP unconditionally, rather than only on a CPL change.

When shadow stacks are enabled at the interrupt handler's privilege level and the interrupted procedure was not executing at a privilege level 3, then the processor pushes the CS:LIP:SSP of the interrupted procedure on the shadow stack of the interrupt handler (where LIP is the linear address of the return address).

Aside from error codes, pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stackframe size across all interrupts. Interrupt service-routine entry points that handle interrupts generated by the INTn instruction or external INTR# signal can push an additional error code place-holder to maintain consistency.

In legacy mode, the stack pointer may be at any alignment when an interrupt or exception causes a stack frame to be pushed. This causes the stack frame and succeeding pushes done by an interrupt handler to be at arbitrary alignments. In IA-32e mode, the RSP is aligned to a 16-byte boundary before pushing the stack frame. The stack frame itself is aligned on a 16-byte boundary when the interrupt handler is called. The processor can arbitrarily realign the new RSP on interrupts because the previous (possibly unaligned) RSP is unconditionally saved on the newly aligned stack. The previous RSP will be automatically restored by a subsequent IRET.

Aligning the stack permits exception and interrupt frames to be aligned on a 16-byte boundary before interrupts are re-enabled. This allows the stack to be formatted for optimal storage of 16-byte XMM registers, which enables the interrupt handler to use faster 16-byte aligned loads and stores (MOVAPS rather than MOVUPS) to save and restore XMM registers.

Although the RSP alignment is always performed when LMA = 1, it is only of consequence for the kernel-mode case where there is no stack switch or IST used. For a stack switch or IST, the OS would have presumably put suitably aligned RSP values in the TSS.

### 6.14.3 IRET in IA-32e Mode

In IA-32e mode, IRET executes with an 8-byte operand size. There is nothing that forces this requirement. The stack is formatted in such a way that for actions where IRET is required, the 8-byte IRET operand size works correctly.

Because interrupt stack-frame pushes are always eight bytes in IA-32e mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64-bit operand-size prefix. The size of the pop is determined by the address size of the instruction. The SS/ESP/RSP size adjustment is determined by the stack size.

IRET pops SS:RSP unconditionally off the interrupt stack frame only when it is executed in 64-bit mode. In compatibility mode, IRET pops SS:RSP off the stack only if there is a CPL change. This allows legacy applications to execute properly in compatibility mode when using the IRET instruction. 64-bit interrupt service routines that exit with an IRET unconditionally pop SS:RSP off of the interrupt stack frame, even if the target code segment is running in 64-bit mode or at CPL = 0. This is because the original interrupt always pushes SS:RSP.

When shadow stacks are enabled and the target privilege level is not 3, the CS:LIP from the shadow stack frame is compared to the return linear address formed by CS:EIP from the stack. If they do not match then the processor caused a control protection exception (#CP(FAR-RET/IRET)), else the processor pops the SSP of the interrupted procedure from the shadow stack. If the target privilege level is 3 and shadow stacks are enabled at privilege level 3, then the SSP for the interrupted procedure is restored from the IA32\_PL3\_SSP MSR.

In IA-32e mode, IRET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL  $\neq$  3, IRET allows SS to be loaded with a NULL selector. As part of the stack switch mechanism, an interrupt or exception sets the new SS to NULL, instead of fetching a new SS selector from the TSS and loading the corresponding descriptor from the GDT or LDT. The new SS selector is set to NULL in order to properly handle returns from subsequent nested far transfers. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent IRET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

### 6.14.4 Stack Switching in IA-32e Mode

The IA-32 architecture provides a mechanism to automatically switch stack frames in response to an interrupt. The 64-bit extensions of Intel 64 architecture implement a modified version of the legacy stack-switching mechanism and an alternative stack-switching mechanism called the interrupt stack table (IST).

In IA-32 modes, the legacy IA-32 stack-switch mechanism is unchanged. In IA-32e mode, the legacy stack-switch mechanism is modified. When stacks are switched as part of a 64-bit mode privilege-level change (resulting from an interrupt), a new SS descriptor is not loaded. IA-32e mode loads only an inner-level RSP from the TSS. The new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INT, interrupts and exceptions). The old SS and RSP are saved on the new stack (Figure 6-9). On the subsequent IRET, the old SS is popped from the stack and loaded into the SS register.

In summary, a stack switch in IA-32e mode works like the legacy stack switch, except that a new SS selector is not loaded from the TSS. Instead, the new SS is forced to NULL.

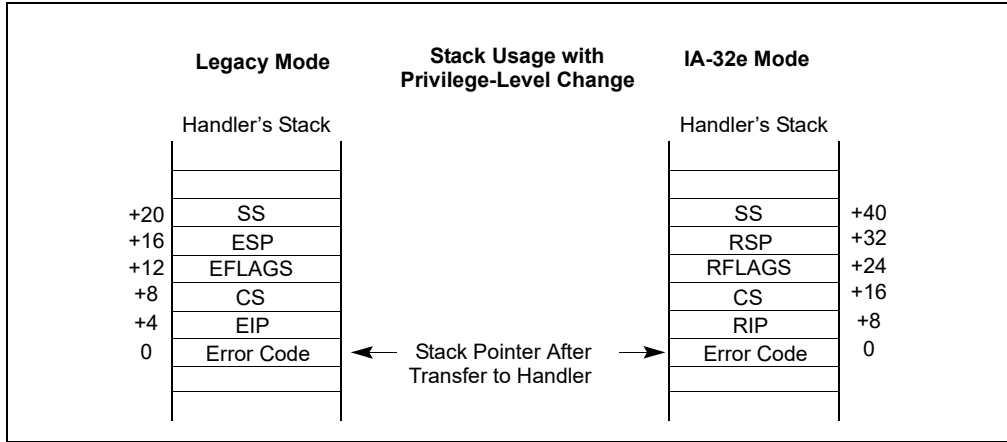


Figure 6-9. IA-32e Mode Stack Usage After Privilege Level Change

### 6.14.5 Interrupt Stack Table

In IA-32e mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack-switching mechanism described above. This mechanism unconditionally switches stacks when it is enabled. It can be enabled on an individual interrupt-vector basis using a field in the IDT entry. This means that some interrupt vectors can use the modified legacy mechanism and others can use the IST mechanism.

The IST mechanism is only available in IA-32e mode. It is part of the 64-bit mode TSS. The motivation for the IST mechanism is to provide a method for specific interrupts (such as NMI, double-fault, and machine-check) to always execute on a known good stack. In legacy mode, interrupts can use the task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the legacy task-switch mechanism is not supported in IA-32e mode.

The IST mechanism provides up to seven IST pointers in the TSS. The pointers are referenced by an interrupt-gate descriptor in the interrupt-descriptor table (IDT); see Figure 6-8. The gate descriptor contains a 3-bit IST index field that provides an offset into the IST section of the TSS. Using the IST mechanism, the processor loads the value pointed by an IST pointer into the RSP.

When an interrupt occurs, the new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal. If the IST index is zero, the modified legacy stack-switching mechanism described above is used.

To support this stack-switching mechanism with shadow stacks enabled, the processor provides an MSR, IA32\_INTERRUPT\_SSP\_TABLE, to program the linear address of a table of seven shadow stack pointers that are selected using the IST index from the gate descriptor. To switch to a shadow stack selected from the interrupt shadow stack table pointed to by the IA32\_INTERRUPT\_SSP\_TABLE, the processor requires that the shadow stack addresses programmed into this table point to a supervisor shadow stack token; see Figure 6-10.

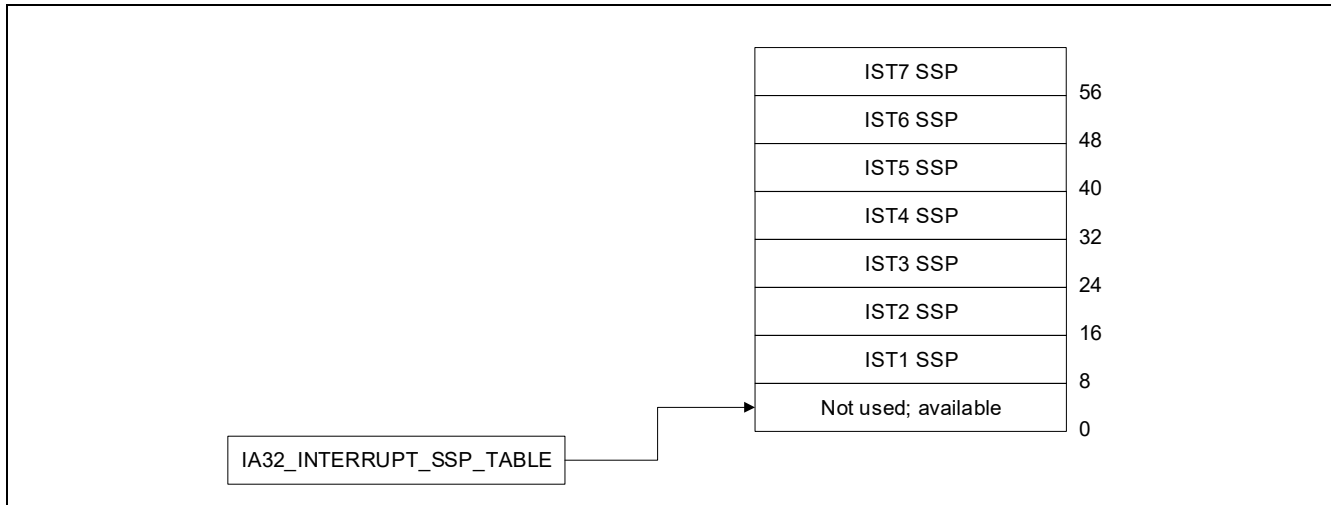


Figure 6-10. Interrupt Shadow Stack Table

## 6.15 EXCEPTION AND INTERRUPT REFERENCE

The following sections describe conditions which generate exceptions and interrupts. They are arranged in the order of vector numbers. The information contained in these sections are as follows:

- **Exception Class** — Indicates whether the exception class is a fault, trap, or abort type. Some exceptions can be either a fault or trap type, depending on when the error condition is detected. (This section is not applicable to interrupts.)
- **Description** — Gives a general description of the purpose of the exception or interrupt. It also describes how the processor handles the exception or interrupt.
- **Exception Error Code** — Indicates whether an error code is saved for the exception. If one is saved, the contents of the error code are described. (This section is not applicable to interrupts.)
- **Saved Instruction Pointer** — Describes which instruction the saved (or return) instruction pointer points to. It also indicates whether the pointer can be used to restart a faulting instruction.
- **Program State Change** — Describes the effects of the exception or interrupt on the state of the currently running program or task and the possibilities of restarting the program or task without loss of continuity.

## Interrupt 0—Divide Error Exception (#DE)

**Exception Class**     **Fault.**

### Description

Indicates the divisor operand for a DIV or IDIV instruction is 0 or that the result cannot be represented in the number of bits specified for the destination operand.

### Exception Error Code

None.

### Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state change does not accompany the divide error, because the exception occurs before the faulting instruction is executed.

## Interrupt 1—Debug Exception (#DB)

**Exception Class**     **Trap or Fault. The exception handler can distinguish between traps or faults by examining the contents of DR6 and the other debug registers.**

### Description

Indicates that one or more of several debug-exception conditions has been detected. Whether the exception is a fault or a trap depends on the condition (see Table 6-3). See Chapter 17, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” for detailed information about the debug exceptions.

**Table 6-3. Debug Exception Conditions and Corresponding Exception Classes**

Exception Condition	Exception Class
Instruction fetch breakpoint	Fault
Data read or write breakpoint	Trap
I/O read or write breakpoint	Trap
General detect condition (in conjunction with in-circuit emulation)	Fault
Single-step	Trap
Task-switch	Trap
Execution of INT1 <sup>1</sup>	Trap

### NOTES:

1. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

### Exception Error Code

None. An exception handler can examine the debug registers to determine which condition caused the exception.

### Saved Instruction Pointer

Fault — Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap — Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

### Program State Change

Fault — A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap — A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

The following items detail the treatment of debug exceptions on the instruction boundary following execution of the MOV or the POP instruction that loads the SS register:

- If EFLAGS.TF is 1, no single-step trap is generated.
- If the instruction encounters a data breakpoint, the resulting debug exception is delivered after completion of the instruction after the MOV or POP. This occurs even if the next instruction is INT *n*, INT3, or INTO.
- Any instruction breakpoint on the instruction after the MOV or POP is suppressed (as if EFLAGS.RF were 1).

Any debug exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a debug exception instead causes execution to roll back to just before the XBEGIN instruction



and then delivers a #DB. See Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Interrupt 2—NMI Interrupt

**Exception Class**     **Not applicable.**

### Description

The nonmaskable interrupt (NMI) is generated externally by asserting the processor's NMI pin or through an NMI request set by the I/O APIC to the local APIC. This interrupt causes the NMI interrupt handler to be called.

### Exception Error Code

Not applicable.

### Saved Instruction Pointer

The processor always takes an NMI interrupt on an instruction boundary. The saved contents of CS and EIP registers point to the next instruction to be executed at the point the interrupt is taken. See Section 6.5, "Exception Classifications," for more information about when the processor takes NMI interrupts.

### Program State Change

The instruction executing when an NMI interrupt is received is completed before the NMI is generated. A program or task can thus be restarted upon returning from an interrupt handler without loss of continuity, provided the interrupt handler saves the state of the processor before handling the interrupt and restores the processor's state prior to a return.

## Interrupt 3—Breakpoint Exception (#BP)

**Exception Class**    **Trap.**

### Description

Indicates that a breakpoint instruction (INT3, opcode CC) was executed, causing a breakpoint trap to be generated. Typically, a debugger sets a breakpoint by replacing the first opcode byte of an instruction with the opcode for the INT3 instruction. (The INT3 instruction is one byte long, which makes it easy to replace an opcode in a code segment in RAM with the breakpoint opcode.) The operating system or a debugging tool can use a data segment mapped to the same physical address space as the code segment to place an INT3 instruction in places where it is desired to call the debugger.

With the P6 family, Pentium, Intel486, and Intel386 processors, it is more convenient to set breakpoints with the debug registers. (See Section 17.3.2, “Breakpoint Exception (#BP)—Interrupt Vector 3,” for information about the breakpoint exception.) If more breakpoints are needed beyond what the debug registers allow, the INT3 instruction can be used.

Any breakpoint exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a break exception instead causes execution to roll back to just before the XBEGIN instruction and then delivers a **debug exception (#DB)** — **not** a breakpoint exception. See Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

A breakpoint exception can also be generated by executing the INT *n* instruction with an operand of 3. The action of this instruction (INT 3) is slightly different than that of the INT3 instruction (see “INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

### Exception Error Code

None.

### Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction following the INT3 instruction.

### Program State Change

Even though the EIP points to the instruction following the breakpoint instruction, the state of the program is essentially unchanged because the INT3 instruction does not affect any register or memory locations. The debugger can thus resume the suspended program by replacing the INT3 instruction that caused the breakpoint with the original opcode and decrementing the saved contents of the EIP register. Upon returning from the debugger, program execution resumes with the replaced instruction.

## Interrupt 4—Overflow Exception (#OF)

**Exception Class**      **Trap.**

### Description

Indicates that an overflow trap occurred when an INTO instruction was executed. The INTO instruction checks the state of the OF flag in the EFLAGS register. If the OF flag is set, an overflow trap is generated.

Some arithmetic instructions (such as the ADD and SUB) perform both signed and unsigned arithmetic. These instructions set the OF and CF flags in the EFLAGS register to indicate signed overflow and unsigned overflow, respectively. When performing arithmetic on signed operands, the OF flag can be tested directly or the INTO instruction can be used. The benefit of using the INTO instruction is that if the overflow exception is detected, an exception handler can be called automatically to handle the overflow condition.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction following the INTO instruction.

### Program State Change

Even though the EIP points to the instruction following the INTO instruction, the state of the program is essentially unchanged because the INTO instruction does not affect any register or memory locations. The program can thus resume normal execution upon returning from the overflow exception handler.

## Interrupt 5—BOUND Range Exceeded Exception (#BR)

**Exception Class**     **Fault.**

### Description

Indicates that a BOUND-range-exceeded fault occurred when a BOUND instruction was executed. The BOUND instruction checks that a signed array index is within the upper and lower bounds of an array located in memory. If the array index is not within the bounds of the array, a BOUND-range-exceeded fault is generated.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the BOUND instruction that generated the exception.

### Program State Change

A program-state change does not accompany the bounds-check fault, because the operands for the BOUND instruction are not modified. Returning from the BOUND-range-exceeded exception handler causes the BOUND instruction to be restarted.

## Interrupt 6—Invalid Opcode Exception (#UD)

**Exception Class**     **Fault.**

### Description

Indicates that the processor did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode; for example, the source operand for a LES instruction is not a memory location.
- Attempted to execute an MMX or SSE/SSE2/SSE3 instruction on an Intel 64 or IA-32 processor that does not support the MMX technology or SSE/SSE2/SSE3/SSSE3 extensions, respectively. CPUID feature flags MMX (bit 23), SSE (bit 25), SSE2 (bit 26), SSE3 (ECX, bit 0), SSSE3 (ECX, bit 9) indicate support for these extensions.
- Attempted to execute an MMX instruction or SSE/SSE2/SSE3/SSSE3 SIMD instruction (with the exception of the MOVNTI, PAUSE, PREFETCH $h$ , SFENCE, LFENCE, MFENCE, CLFLUSH, MONITOR, and MWAIT instructions) when the EM flag in control register CR0 is set (1).
- Attempted to execute an SSE/SE2/SSE3/SSSE3 instruction when the OSFXSR bit in control register CR4 is clear (0). Note this does not include the following SSE/SSE2/SSE3 instructions: MASKMOVQ, MOVNTQ, MOVNTI, PREFETCH $h$ , SFENCE, LFENCE, MFENCE, and CLFLUSH; or the 64-bit versions of the PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADB, PSHUFW, PADDQ, PSUBQ, PALIGNR, PABSB, PABSD, PABSW, PHADDD, PHADDSW, PHADDW, PHSUBD, PHSUBSW, PHSUBW, PMADDUSB, PMULHRW, PSHUFB, PSIGNB, PSIGND, and PSIGNW.
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction on an Intel 64 or IA-32 processor that caused a SIMD floating-point exception when the OSXMMEXCPT bit in control register CR4 is clear (0).
- Executed a UD0, UD1 or UD2 instruction. Note that even though it is the execution of the UD0, UD1 or UD2 instruction that causes the invalid opcode exception, the saved instruction pointer will still points at the UD0, UD1 or UD2 instruction.
- Detected a LOCK prefix that precedes an instruction that may not be locked or one that may be locked but the destination operand is not a memory location.
- Attempted to execute an LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW, or ARPL instruction while in real-address or virtual-8086 mode.
- Attempted to execute the RSM instruction when not in SMM mode.

In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 6.5, "Exception Classifications," for general rules for taking of interrupts and exceptions.)

The opcodes D6 and F1 are undefined opcodes reserved by the Intel 64 and IA-32 architectures. These opcodes, even though undefined, do not generate an invalid opcode exception.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state change does not accompany an invalid-opcode fault, because the invalid instruction is not executed.

## Interrupt 7—Device Not Available Exception (#NM)

**Exception Class**     **Fault.**

### Description

Indicates one of the following things:

The device-not-available exception is generated by either of three conditions:

- The processor executed an x87 FPU floating-point instruction while the EM flag in control register CR0 was set (1). See the paragraph below for the special case of the WAIT/FWAIT instruction.
- The processor executed a WAIT/FWAIT instruction while the MP and TS flags of register CR0 were set, regardless of the setting of the EM flag.
- The processor executed an x87 FPU, MMX, or SSE/SSE2/SSE3 instruction (with the exception of MOVNTI, PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, and CLFLUSH) while the TS flag in control register CR0 was set and the EM flag is clear.

The EM flag is set when the processor does not have an internal x87 FPU floating-point unit. A device-not-available exception is then generated each time an x87 FPU floating-point instruction is encountered, allowing an exception handler to call floating-point instruction emulation routines.

The TS flag indicates that a context switch (task switch) has occurred since the last time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction was executed; but that the context of the x87 FPU, XMM, and MXCSR registers were not saved. When the TS flag is set and the EM flag is clear, the processor generates a device-not-available exception each time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction is encountered (with the exception of the instructions listed above). The exception handler can then save the context of the x87 FPU, XMM, and MXCSR registers before it executes the instruction. See Section 2.5, "Control Registers," for more information about the TS flag.

The MP flag in control register CR0 is used along with the TS flag to determine if WAIT or FWAIT instructions should generate a device-not-available exception. It extends the function of the TS flag to the WAIT and FWAIT instructions, giving the exception handler an opportunity to save the context of the x87 FPU before the WAIT or FWAIT instruction is executed. The MP flag is provided primarily for use with the Intel 286 and Intel386 DX processors. For programs running on the Pentium 4, Intel Xeon, P6 family, Pentium, or Intel486 DX processors, or the Intel 487 SX coprocessors, the MP flag should always be set; for programs running on the Intel486 SX processor, the MP flag should be clear.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the floating-point instruction or the WAIT/FWAIT instruction that generated the exception.

### Program State Change

A program-state change does not accompany a device-not-available fault, because the instruction that generated the exception is not executed.

If the EM flag is set, the exception handler can then read the floating-point instruction pointed to by the EIP and call the appropriate emulation routine.

If the MP and TS flags are set or the TS flag alone is set, the exception handler can save the context of the x87 FPU, clear the TS flag, and continue execution at the interrupted floating-point or WAIT/FWAIT instruction.

## Interrupt 8—Double Fault Exception (#DF)

**Exception Class**     **Abort.**

### Description

Indicates that the processor detected a second exception while calling an exception handler for a prior exception. Normally, when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception. To determine when two faults need to be signalled as a double fault, the processor divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults (see Table 6-4).

**Table 6-4. Interrupt and Exception Classes**

Class	Vector Number	Description
Benign Exceptions and Interrupts	1	Debug
	2	NMI Interrupt
	3	Breakpoint
	4	Overflow
	5	BOUND Range Exceeded
	6	Invalid Opcode
	7	Device Not Available
	9	Coprocessor Segment Overrun
	16	Floating-Point Error
	17	Alignment Check
	18	Machine Check
	19	SIMD floating-point
	All	INT <i>n</i>
All	INTR	
Contributory Exceptions	0	Divide Error
	10	Invalid TSS
	11	Segment Not Present
	12	Stack Fault
	13	General Protection
	21	Control Protection
Page Faults	14	Page Fault
	20	Virtualization Exception

Table 6-5 shows the various combinations of exception classes that cause a double fault to be generated. A double-fault exception falls in the abort class of exceptions. The program or task cannot be restarted or resumed. The double-fault handler can be used to collect diagnostic information about the state of the machine and/or, when possible, to shut the application and/or system down gracefully or restart the system.



A segment or page fault may be encountered while prefetching instructions; however, this behavior is outside the domain of Table 6-5. Any further faults generated while the processor is attempting to transfer control to the appropriate fault handler could still lead to a double-fault sequence.

**Table 6-5. Conditions for Generating a Double Fault**

First Exception	Second Exception		
	Benign	Contributory	Page Fault
Benign	Handle Exceptions Serially	Handle Exceptions Serially	Handle Exceptions Serially
Contributory	Handle Exceptions Serially	Generate a Double Fault	Handle Exceptions Serially
Page Fault	Handle Exceptions Serially	Generate a Double Fault	Generate a Double Fault
Double Fault	Handle Exceptions Serially	Enter Shutdown Mode	Enter Shutdown Mode

If another contributory or page fault exception occurs while attempting to call the double-fault handler, the processor enters shutdown mode. This mode is similar to the state following execution of an HLT instruction. In this mode, the processor stops executing instructions until an NMI interrupt, SMI interrupt, hardware reset, or INIT# is received. The processor generates a special bus cycle to indicate that it has entered shutdown mode. Software designers may need to be aware of the response of hardware when it goes into shutdown mode. For example, hardware may turn on an indicator light on the front panel, generate an NMI interrupt to record diagnostic information, invoke reset initialization, generate an INIT initialization, or generate an SMI. If any events are pending during shutdown, they will be handled after a wake event from shutdown is processed (for example, A20M# interrupts).

If a shutdown occurs while the processor is executing an NMI interrupt handler, then only a hardware reset can restart the processor. Likewise, if the shutdown occurs while executing in SMM, a hardware reset must be used to restart the processor.

**Exception Error Code**

Zero. The processor always pushes an error code of 0 onto the stack of the double-fault handler.

**Saved Instruction Pointer**

The saved contents of CS and EIP registers are undefined.

**Program State Change**

A program-state following a double-fault exception is undefined. The program or task cannot be resumed or restarted. The only available action of the double-fault exception handler is to collect all possible context information for use in diagnostics and then close the application and/or shut down or reset the processor.

If the double fault occurs when any portion of the exception handling machine state is corrupted, the handler cannot be invoked and the processor must be reset.

## Interrupt 9—Coprocesor Segment Overrun

**Exception Class**     **Abort. (Intel reserved; do not use. Recent IA-32 processors do not generate this exception.)**

### Description

Indicates that an Intel386 CPU-based systems with an Intel 387 math coprocessor detected a page or segment violation while transferring the middle portion of an Intel 387 math coprocessor operand. The P6 family, Pentium, and Intel486 processors do not generate this exception; instead, this condition is detected with a general protection exception (`#GP`), interrupt 13.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state following a coprocessor segment-overrun exception is undefined. The program or task cannot be resumed or restarted. The only available action of the exception handler is to save the instruction pointer and reinitialize the x87 FPU using the `FNINIT` instruction.

## Interrupt 10—Invalid TSS Exception (#TS)

**Exception Class**     **Fault.**

### Description

Indicates that there was an error related to a TSS. Such an error might be detected during a task switch or during the execution of instructions that use information from a TSS. Table 6-6 shows the conditions that cause an invalid TSS exception to be generated.

**Table 6-6. Invalid TSS Conditions**

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment limit is less than 67H for 32-bit TSS or less than 2CH for 16-bit TSS.
TSS segment selector index	During an IRET task switch, the TI flag in the TSS segment selector indicates the LDT.
TSS segment selector index	During an IRET task switch, the TSS segment selector exceeds descriptor table limit.
TSS segment selector index	During an IRET task switch, the busy flag in the TSS descriptor indicates an inactive task.
TSS segment selector index	During a task switch, an attempt to access data in a TSS results in a limit violation or canonical fault.
TSS segment selector index	During an IRET task switch, the backlink is a NULL selector.
TSS segment selector index	During an IRET task switch, the backlink points to a descriptor which is not a busy TSS.
TSS segment selector index	The new TSS descriptor is beyond the GDT limit.
TSS segment selector index	The new TSS selector is null on an attempt to lock the new TSS.
TSS segment selector index	The new TSS selector has the TI bit set on an attempt to lock the new TSS.
TSS segment selector index	The new TSS descriptor is not an available TSS descriptor on an attempt to lock the new TSS.
LDT segment selector index	LDT not valid or not present.
Stack segment selector index	The stack segment selector exceeds descriptor table limit.
Stack segment selector index	The stack segment selector is NULL.
Stack segment selector index	The stack segment descriptor is a non-data segment.
Stack segment selector index	The stack segment is not writable.
Stack segment selector index	The stack segment DPL ≠ CPL.
Stack segment selector index	The stack segment selector RPL ≠ CPL.
Code segment selector index	The code segment selector exceeds descriptor table limit.
Code segment selector index	The code segment selector is NULL.
Code segment selector index	The code segment descriptor is not a code segment type.
Code segment selector index	The nonconforming code segment DPL ≠ CPL.
Code segment selector index	The conforming code segment DPL is greater than CPL.
Data segment selector index	The data segment selector exceeds the descriptor table limit.
Data segment selector index	The data segment descriptor is not a readable code or data type.
Data segment selector index	The data segment descriptor is a nonconforming code type and RPL > DPL.
Data segment selector index	The data segment descriptor is a nonconforming code type and CPL > DPL.
TSS segment selector index	The TSS segment descriptor/upper descriptor is beyond the GDT segment limit.
TSS segment selector index	The TSS segment descriptor is not an available TSS type.
TSS segment selector index	The TSS segment descriptor is an available 286 TSS type in IA-32e mode.

**Table 6-6. Invalid TSS Conditions (Contd.)**

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment upper descriptor is not the correct type.
TSS segment selector index	The TSS segment descriptor contains a non-canonical base.

This exception can be generated either in the context of the original task or in the context of the new task (see Section 7.3, “Task Switching”). Until the processor has completely verified the presence of the new TSS, the exception is generated in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete. Any invalid-TSS conditions detected after this point are handled in the context of the new task. (A task switch is considered complete when the task register is loaded with the segment selector for the new TSS and, if the switch is due to a procedure call or interrupt, the previous task link field of the new TSS references the old TSS.)

The invalid-TSS handler must be a task called using a task gate. Handling this exception inside the faulting TSS context is not recommended because the processor state may not be consistent.

### Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception was caused by an event external to the currently running program (for example, if an external interrupt handler using a task gate attempted a task switch to an invalid TSS).

### Saved Instruction Pointer

If the exception condition was detected before the task switch was carried out, the saved contents of CS and EIP registers point to the instruction that invoked the task switch. If the exception condition was detected after the task switch was carried out, the saved contents of CS and EIP registers point to the first instruction of the new task.

### Program State Change

The ability of the invalid-TSS handler to recover from the fault depends on the error condition that causes the fault. See Section 7.3, “Task Switching,” for more information on the task switch process and the possible recovery actions that can be taken.

If an invalid TSS exception occurs during a task switch, it can occur before or after the commit-to-new-task point. If it occurs before the commit point, no program state change occurs. If it occurs after the commit point (when the segment descriptor information for the new segment selectors have been loaded in the segment registers), the processor will load all the state information from the new TSS before it generates the exception. During a task switch, the processor first loads all the segment registers with segment selectors from the TSS, then checks their contents for validity. If an invalid TSS exception is discovered, the remaining segment registers are loaded but not checked for validity and therefore may not be usable for referencing memory. The invalid TSS handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should load all segment registers before trying to resume the new task; otherwise, general-protection exceptions (#GP) may result later under conditions that make diagnosis more difficult. The Intel recommended way of dealing with this situation is to use a task for the invalid TSS exception handler. The task switch back to the interrupted task from the invalid-TSS exception-handler task will then cause the processor to check the registers as it loads them from the TSS.

## Interrupt 11—Segment Not Present (#NP)

**Exception Class**     **Fault.**

### Description

Indicates that the present flag of a segment or gate descriptor is clear. The processor can generate this exception during any of the following operations:

- While attempting to load CS, DS, ES, FS, or GS registers. [Detection of a not-present segment while loading the SS register causes a stack fault exception (#SS) to be generated.] This situation can occur while performing a task switch.
- While attempting to load the LDTR using an LLDT instruction. Detection of a not-present LDT while loading the LDTR during a task switch operation causes an invalid-TSS exception (#TS) to be generated.
- When executing the LTR instruction and the TSS is marked not present.
- While attempting to use a gate descriptor or TSS that is marked segment-not-present, but is otherwise valid.

An operating system typically uses the segment-not-present exception to implement virtual memory at the segment level. If the exception handler loads the segment and returns, the interrupted program or task resumes execution.

A not-present indication in a gate descriptor, however, does not indicate that a segment is not present (because gates do not correspond to segments). The operating system may use the present flag for gate descriptors to trigger exceptions of special significance to the operating system.

A contributory exception or page fault that subsequently referenced a not-present segment would cause a double fault (#DF) to be generated instead of #NP.

### Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception resulted from either:

- an external event (NMI or INTR) that caused an interrupt, which subsequently referenced a not-present segment
- a benign exception that subsequently referenced a not-present segment

The IDT flag is set if the error code refers to an IDT entry. This occurs when the IDT entry for an interrupt being serviced references a not-present gate. Such an event could be generated by an INT instruction or a hardware interrupt.

### Saved Instruction Pointer

The saved contents of CS and EIP registers normally point to the instruction that generated the exception. If the exception occurred while loading segment descriptors for the segment selectors in a new TSS, the CS and EIP registers point to the first instruction in the new task. If the exception occurred while accessing a gate descriptor, the CS and EIP registers point to the instruction that invoked the access (for example a CALL instruction that references a call gate).

### Program State Change

If the segment-not-present exception occurs as the result of loading a register (CS, DS, SS, ES, FS, GS, or LDTR), a program-state change does accompany the exception because the register is not loaded. Recovery from this exception is possible by simply loading the missing segment into memory and setting the present flag in the segment descriptor.

If the segment-not-present exception occurs while accessing a gate descriptor, a program-state change does not accompany the exception. Recovery from this exception is possible merely by setting the present flag in the gate descriptor.

If a segment-not-present exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 7.3, "Task Switching"). If it occurs before the commit point, no program state change

occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The segment-not-present exception handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

## Interrupt 12—Stack Fault Exception (#SS)

**Exception Class**     **Fault.**

### Description

Indicates that one of the following stack related conditions was detected:

- A limit violation is detected during an operation that refers to the SS register. Operations that can cause a limit violation include stack-oriented instructions such as POP, PUSH, CALL, RET, IRET, ENTER, and LEAVE, as well as other memory references which implicitly or explicitly use the SS register (for example, MOV AX, [BP+6] or MOV AX, SS:[EAX+6]). The ENTER instruction generates this exception when there is not enough stack space for allocating local variables.
- A not-present stack segment is detected when attempting to load the SS register. This violation can occur during the execution of a task switch, a CALL instruction to a different privilege level, a return to a different privilege level, an LSS instruction, or a MOV or POP instruction to the SS register.
- A canonical violation is detected in 64-bit mode during an operation that reference memory using the stack pointer register containing a non-canonical memory address.

Recovery from this fault is possible by either extending the limit of the stack segment (in the case of a limit violation) or loading the missing stack segment into memory (in the case of a not-present violation).

In the case of a canonical violation that was caused intentionally by software, recovery is possible by loading the correct canonical value into RSP. Otherwise, a canonical violation of the address in RSP likely reflects some register corruption in the software.

### Exception Error Code

If the exception is caused by a not-present stack segment or by overflow of the new stack during an inter-privilege-level call, the error code contains a segment selector for the segment that caused the exception. Here, the exception handler can test the present flag in the segment descriptor pointed to by the segment selector to determine the cause of the exception. For a normal limit violation (on a stack segment already in use) the error code is set to 0.

### Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. However, when the exception results from attempting to load a not-present stack segment during a task switch, the CS and EIP registers point to the first instruction of the new task.

### Program State Change

A program-state change does not generally accompany a stack-fault exception, because the instruction that generated the fault is not executed. Here, the instruction can be restarted after the exception handler has corrected the stack fault condition.

If a stack fault occurs during a task switch, it occurs after the commit-to-new-task point (see Section 7.3, "Task Switching"). Here, the processor loads all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The stack fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that are more difficult to diagnose. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

## Interrupt 13—General Protection Exception (#GP)

**Exception Class**     **Fault.**

### Description

Indicates that the processor detected one of a class of protection violations called “general-protection violations.” The conditions that cause this exception to be generated comprise all the protection violations that do not cause other exceptions to be generated (such as, invalid-TSS, segment-not-present, stack-fault, or page-fault exceptions). The following conditions cause general-protection exceptions to be generated:

- Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments.
- Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
- Transferring execution to a segment that is not executable.
- Writing to a code segment or a read-only data segment.
- Reading from an execute-only code segment.
- Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs).
- Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment.
- Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment.
- Loading the SS register with the segment selector of an executable segment or a null segment selector.
- Loading the CS register with a segment selector for a data segment or a null segment selector.
- Accessing memory using the DS, ES, FS, or GS register when it contains a null segment selector.
- Switching to a busy task during a call or jump to a TSS.
- Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
- Violating any of the privilege rules described in Chapter 5, “Protection.”
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).
- Loading the CR0 register with a set PG flag (paging enabled) and a clear PE flag (protection disabled).
- Loading the CR0 register with a set NW flag and a clear CD flag.
- Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
- Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handler’s code segment DPL is greater than 0.
- Attempting to write a 1 into a reserved bit of CR4.
- Attempting to execute a privileged instruction when the CPL is not equal to 0 (see Section 5.9, “Privileged Instructions,” for a list of privileged instructions).
- Attempting to execute SGDT, SIDT, SLDT, SMSW, or STR when CR4.UMIP = 1 and the CPL is not equal to 0.
- Writing to a reserved bit in an MSR.
- Accessing a gate that contains a null segment selector.
- Executing the INT *n* instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.
- The segment selector in a call, interrupt, or trap gate does not point to a code segment.
- The segment selector operand in the LLDT instruction is a local type (TI flag is set) or does not point to a segment descriptor of the LDT type.
- The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
- The target code-segment selector for a call, jump, or return is null.



- If the PAE and/or PSE flag in control register CR4 is set and the processor detects any reserved bits in a page-directory-pointer-table entry set to 1. These bits are checked during a write to control registers CR0, CR3, or CR4 that causes a reloading of the page-directory-pointer-table entry.
- Attempting to write a non-zero value into the reserved bits of the MXCSR register.
- Executing an SSE/SSE2/SSE3 instruction that attempts to access a 128-bit memory location that is not aligned on a 16-byte boundary when the instruction requires 16-byte alignment. This condition also applies to the stack segment.

A program or task can be restarted following any general-protection exception. If the exception occurs while attempting to call an interrupt handler, the interrupted program can be restartable, but the interrupt may be lost.

### Exception Error Code

The processor pushes an error code onto the exception handler's stack. If the fault condition was detected while loading a segment descriptor, the error code contains a segment selector to or IDT vector number for the descriptor; otherwise, the error code is 0. The source of the selector in an error code may be any of the following:

- An operand of the instruction.
- A selector from a gate which is the operand of the instruction.
- A selector from a TSS involved in a task switch.
- IDT vector number.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

In general, a program-state change does not accompany a general-protection exception, because the invalid instruction or operation is not executed. An exception handler can be designed to correct all of the conditions that cause general-protection exceptions and restart the program or task without any loss of program continuity.

If a general-protection exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 7.3, "Task Switching"). If it occurs before the commit point, no program state change occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The general-protection exception handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

### General Protection Exception in 64-bit Mode

The following conditions cause general-protection exceptions in 64-bit mode:

- If the memory address is in a non-canonical form.
- If a segment descriptor memory address is in non-canonical form.
- If the target offset in a destination operand of a call or jmp is in a non-canonical form.
- If a code segment or 64-bit call gate overlaps non-canonical space.
- If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
- If the EFLAGS.NT bit is set in IRET.
- If the stack segment selector of IRET is null when going back to compatibility mode.
- If the stack segment selector of IRET is null going back to CPL3 and 64-bit mode.
- If a null stack segment selector RPL of IRET is not equal to CPL going back to non-CPL3 and 64-bit mode.
- If the proposed new code segment descriptor of IRET has both the D-bit and the L-bit set.

- If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and it has both the D-bit and the L-bit set.
- If the segment descriptor from a 64-bit call gate is in non-canonical space.
- If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
- If the type field of the upper 64 bits of a 64-bit call gate is not 0.
- If an attempt is made to load a null selector in the SS register in compatibility mode.
- If an attempt is made to load null selector in the SS register in CPL3 and 64-bit mode.
- If an attempt is made to load a null selector in the SS register in non-CPL3 and 64-bit mode where RPL is not equal to CPL.
- If an attempt is made to clear CR0.PG while IA-32e mode is enabled.
- If an attempt is made to set a reserved bit in CR3, CR4 or CR8.

## Interrupt 14—Page-Fault Exception (#PF)

**Exception Class**     **Fault.**

### Description

Indicates that, with paging enabled (the PG flag in the CR0 register is set), the processor detected one of the following conditions while using the page-translation mechanism to translate a linear address to a physical address:

- The P (present) flag in a page-directory or page-table entry needed for the address translation is clear, indicating that a page table or the page containing the operand is not present in physical memory.
- The procedure does not have sufficient privilege to access the indicated page (that is, a procedure running in user mode attempts to access a supervisor-mode page). If the SMAP flag is set in CR4, a page fault may also be triggered by code running in supervisor mode that tries to access data at a user-mode address. If either the PKE flag or the PKS flag is set in CR4, the protection-key rights registers may cause page faults on data accesses to linear addresses with certain protection keys.
- Code running in user mode attempts to write to a read-only page. If the WP flag is set in CR0, the page fault will also be triggered by code running in supervisor mode that tries to write to a read-only page.
- An instruction fetch to a linear address that translates to a physical address in a memory page with the execute-disable bit set (for information about the execute-disable bit, see Chapter 4, “Paging”). If the SMEP flag is set in CR4, a page fault will also be triggered by code running in supervisor mode that tries to fetch an instruction from a user-mode address.
- One or more reserved bits in paging-structure entry are set to 1. See description below of RSVD error code flag.
- A shadow-stack access is made to a page that is not a shadow-stack page. See Section 18.2, “Shadow Stacks” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* and Chapter 4.6, “Access Rights.”
- An enclave access violates one of the specified access-control requirements. See Section 33.3, “Access-control Requirements” and Section 33.20, “Enclave Page Cache Map (EPCM)” in Chapter 33, “Enclave Access Control and Data Structures.” In this case, the exception is called an **SGX-induced page fault**. The processor uses the error code (below) to distinguish SGX-induced page faults from ordinary page faults.

The exception handler can recover from page-not-present conditions and restart the program or task without any loss of program continuity. It can also restart the program or task after a privilege violation, but the problem that caused the privilege violation may be uncorrectable.

See also: Section 4.7, “Page-Fault Exceptions.”

### Exception Error Code

Yes (special format). The processor provides the page-fault handler with two items of information to aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 6-11). The processor establishes the bits in the error code as follows:
  - P flag (bit 0).  
This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
  - W/R (bit 1).  
If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
  - U/S (bit 2).  
If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

- RSVD flag (bit 3).  
This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address.
- I/D flag (bit 4).  
This flag is 1 if the access causing the page-fault exception was an instruction fetch. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- PK flag (bit 5).  
This flag is 1 if the access causing the page-fault exception was a data access to a linear address with a protection key for which the protection-key rights registers disallow access.
- SS (bit 1).  
If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- SGX flag (bit 15).  
This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

See Section 4.6, “Access Rights” and Section 4.7, “Page-Fault Exceptions” for more information about page-fault exceptions and the error codes that they produce.

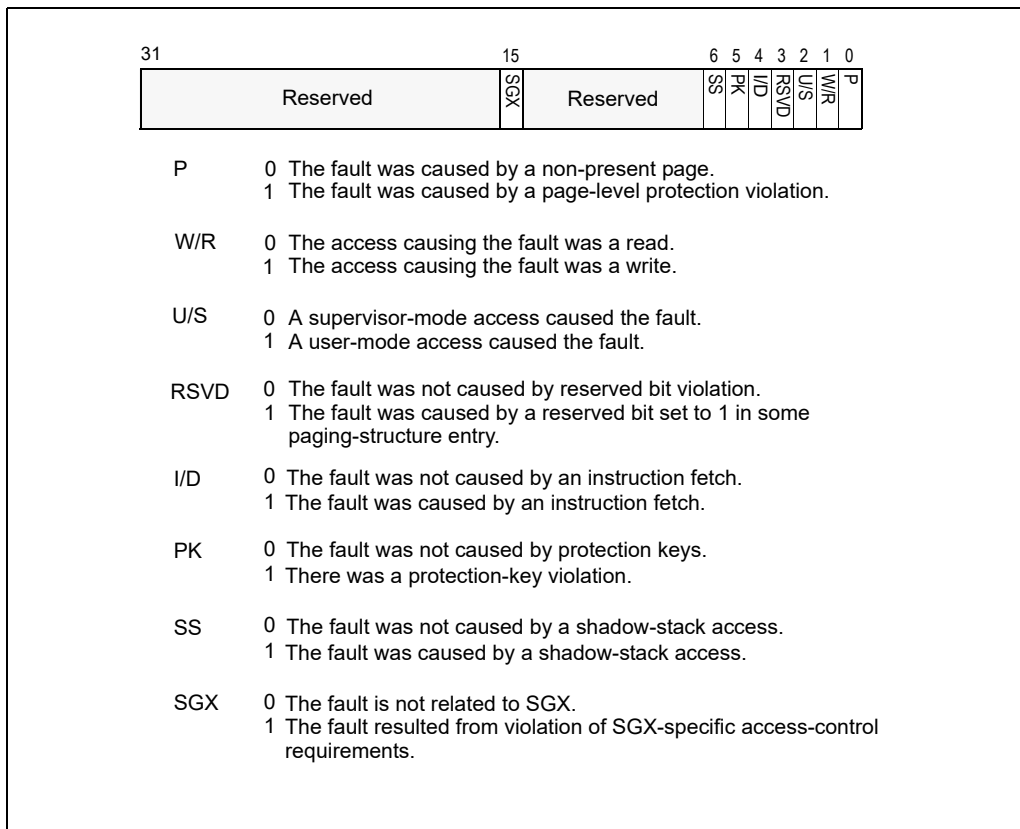


Figure 6-11. Page-Fault Error Code

- The contents of the CR2 register. The processor loads the CR2 register with the 32-bit linear address that generated the exception. The page-fault handler can use this address to locate the corresponding page directory and page-table entries. Another page fault can potentially occur during execution of the page-fault handler; the handler should save the contents of the CR2 register before a second page fault can occur.<sup>6</sup> If a page fault is caused by a page-level protection violation, the access flag in the page-directory entry is set when

the fault occurs. The behavior of IA-32 processors regarding the access flag in the corresponding page-table entry is model specific and not architecturally defined.

### Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. If the page-fault exception occurred during a task switch, the CS and EIP registers may point to the first instruction of the new task (as described in the following “Program State Change” section).

### Program State Change

A program-state change does not normally accompany a page-fault exception, because the instruction that causes the exception to be generated is not executed. After the page-fault exception handler has corrected the violation (for example, loaded the missing page into memory), execution of the program or task can be resumed.

When a page-fault exception is generated during a task switch, the program-state may change, as follows. During a task switch, a page-fault exception can occur during any of following operations:

- While writing the state of the original task into the TSS of that task.
- While reading the GDT to locate the TSS descriptor of the new task.
- While reading the TSS of the new task.
- While reading segment descriptors associated with segment selectors from the new task.
- While reading the LDT of the new task to verify the segment registers stored in the new TSS.

In the last two cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be called through a task gate.

If a page fault occurs during a task switch, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The page-fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

### Additional Exception-Handling Information

Special care should be taken to ensure that an exception that occurs during an explicit stack switch does not cause the processor to use an invalid stack pointer (SS:ESP). Software written for 16-bit IA-32 processors often use a pair of instructions to change to a new stack, for example:

```
MOV SS, AX
MOV SP, StackTop
```

When executing this code on one of the 32-bit IA-32 processors, it is possible to get a page fault, general-protection fault (#GP), or alignment check fault (#AC) after the segment selector has been loaded into the SS register but before the ESP register has been loaded. At this point, the two parts of the stack pointer (SS and ESP) are inconsistent. The new stack segment is being used with the old stack pointer.

The processor does not use the inconsistent stack pointer if the exception handler switches to a well defined stack (that is, the handler is a task or a more privileged procedure). However, if the exception handler is called at the same privilege level and from the same task, the processor will attempt to use the inconsistent stack pointer.

In systems that handle page-fault, general-protection, or alignment check exceptions within the faulting task (with trap or interrupt gates), software executing at the same privilege level as the exception handler should initialize a new stack by using the LSS instruction rather than a pair of MOV instructions, as described earlier in this note. When the exception handler is running at privilege level 0 (the normal case), the problem is limited to procedures or tasks that run at privilege level 0, typically the kernel of the operating system.

- 
6. Processors update CR2 whenever a page fault is detected. If a second page fault occurs while an earlier page fault is being delivered, the faulting linear address of the second fault will overwrite the contents of CR2 (replacing the previous address). These updates to CR2 occur even if the page fault results in a double fault or occurs during the delivery of a double fault.

## Interrupt 16—x87 FPU Floating-Point Error (#MF)

**Exception Class**     **Fault.**

### Description

Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, “Control Registers,” for a detailed description of the NE flag.)

### NOTE

SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

- Invalid operation (#I)
  - Stack overflow or underflow (#IS)
  - Invalid arithmetic operation (#IA)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

Each of these error conditions represents an x87 FPU exception type, and for each of exception type, the x87 FPU provides a flag in the x87 FPU status register and a mask bit in the x87 FPU control register. If the x87 FPU detects a floating-point error and the mask bit for the exception type is set, the x87 FPU handles the exception automatically by generating a predefined (default) response and continuing program execution. The default responses have been designed to provide a reasonable result for most floating-point applications.

If the mask for the exception is clear and the NE flag in register CR0 is set, the x87 FPU does the following:

1. Sets the necessary flag in the FPU status register.
2. Waits until the next “waiting” x87 FPU instruction or WAIT/FWAIT instruction is encountered in the program’s instruction stream.
3. Generates an internal error signal that cause the processor to generate a floating-point exception (#MF).

Prior to executing a waiting x87 FPU instruction or the WAIT/FWAIT instruction, the x87 FPU checks for pending x87 FPU floating-point exceptions (as described in step 2 above). Pending x87 FPU floating-point exceptions are ignored for “non-waiting” x87 FPU instructions, which include the FNINIT, FNCLEX, FNSTSW, FNSTSW AX, FNSTCW, FNSTENV, and FNSAVE instructions. Pending x87 FPU exceptions are also ignored when executing the state management instructions FXSAVE and FXRSTOR.

All of the x87 FPU floating-point error conditions can be recovered from. The x87 FPU floating-point-error exception handler can determine the error condition that caused the exception from the settings of the flags in the x87 FPU status word. See “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on handling x87 FPU floating-point exceptions.

### Exception Error Code

None. The x87 FPU provides its own error information.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the floating-point or WAIT/FWAIT instruction that was about to be executed when the floating-point-error exception was generated. This is not the faulting instruction in which the error condition was detected. The address of the faulting instruction is contained in the x87 FPU instruction pointer

register. See Section 8.1.8, “x87 FPU Instruction and Data (Operand) Pointers” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about information the FPU saves for use in handling floating-point-error exceptions.

### Program State Change

A program-state change generally accompanies an x87 FPU floating-point exception because the handling of the exception is delayed until the next waiting x87 FPU floating-point or WAIT/FWAIT instruction following the faulting instruction. The x87 FPU, however, saves sufficient information about the error condition to allow recovery from the error and re-execution of the faulting instruction if needed.

In situations where non- x87 FPU floating-point instructions depend on the results of an x87 FPU floating-point instruction, a WAIT or FWAIT instruction can be inserted in front of a dependent instruction to force a pending x87 FPU floating-point exception to be handled before the dependent instruction is executed. See “x87 FPU Exception Synchronization” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about synchronization of x87 floating-point-error exceptions.

## Interrupt 17—Alignment Check Exception (#AC)

**Exception Class**     **Fault.**

### Description

Indicates that the processor detected an unaligned memory operand when alignment checking was enabled. Alignment checks are only carried out in data (or stack) accesses (not in code fetches or system segment accesses). An example of an alignment-check violation is a word stored at an odd byte address, or a doubleword stored at an address that is not an integer multiple of 4. Table 6-7 lists the alignment requirements various data types recognized by the processor.

**Table 6-7. Alignment Requirements by Data Type**

Data Type	Address Must Be Divisible By
Word	2
Doubleword	4
Single-precision floating-point (32-bits)	4
Double-precision floating-point (64-bits)	8
Double extended-precision floating-point (80-bits)	8
Quadword	8
Double quadword	16
Segment Selector	2
32-bit Far Pointer	2
48-bit Far Pointer	4
32-bit Pointer	4
GDTR, IDTR, LDTR, or Task Register Contents	4
FSTENV/FLDENV Save Area	4 or 2, depending on operand size
FSAVE/FRSTOR Save Area	4 or 2, depending on operand size
Bit String	2 or 4 depending on the operand-size attribute.

Note that the alignment check exception (#AC) is generated only for data types that must be aligned on word, doubleword, and quadword boundaries. A general-protection exception (#GP) is generated 128-bit data types that are not aligned on a 16-byte boundary.

To enable alignment checking, the following conditions must be true:

- AM flag in CR0 register is set.
- AC flag in the EFLAGS register is set.
- The CPL is 3 (including virtual-8086 mode).

Alignment-check exceptions (#AC) are generated only when operating at privilege level 3 (user mode). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate alignment-check exceptions, even when caused by a memory reference made from privilege level 3.

Storing the contents of the GDTR, IDTR, LDTR, or task register in memory while at privilege level 3 can generate an alignment-check exception. Although application programs do not normally store these registers, the fault can be avoided by aligning the information stored on an even word-address.

The FXSAVE/XSAVE and FXRSTOR/XRSTOR instructions save and restore a 512-byte data structure, the first byte of which must be aligned on a 16-byte boundary. If the alignment-check exception (#AC) is enabled when executing these instructions (and CPL is 3), a misaligned memory operand can cause either an alignment-check exception or a general-protection exception (#GP) depending on the processor implementation (see “FXSAVE-Save x87 FPU, MMX, SSE, and SSE2 State” and “FXRSTOR-Restore x87 FPU, MMX, SSE, and SSE2 State” in



Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*; see "XSAVE—Save Processor Extended States" and "XRSTOR—Restore Processor Extended States" in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*).

The MOVDQU, MOVUPS, and MOVUPD instructions perform 128-bit unaligned loads or stores. The LDDQU instruction loads 128-bit unaligned data. They do not generate general-protection exceptions (#GP) when operands are not aligned on a 16-byte boundary. If alignment checking is enabled, alignment-check exceptions (#AC) may or may not be generated depending on processor implementation when data addresses are not aligned on an 8-byte boundary.

FSAVE and FRSTOR instructions can generate unaligned references, which can cause alignment-check faults. These instructions are rarely needed by application programs.

### Exception Error Code

Yes. The error code is null; all bits are clear except possibly bit 0 — EXT; see Section 6.13. EXT is set if the #AC is recognized during delivery of an event other than a software interrupt (see "INT n/INTO/INT3/INT1—Call to Interrupt Procedure" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state change does not accompany an alignment-check fault, because the instruction is not executed.

## Interrupt 18—Machine-Check Exception (#MC)

**Exception Class**     **Abort.**

### Description

Indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The machine-check exception is model-specific, available on the Pentium and later generations of processors. The implementation of the machine-check exception is different between different processor families, and these implementations may not be compatible with future Intel 64 or IA-32 processors. (Use the CPUID instruction to determine whether this feature is present.)

Bus errors detected by external agents are signaled to the processor on dedicated pins: the BINIT# and MCERR# pins on the Pentium 4, Intel Xeon, and P6 family processors and the BUSCHK# pin on the Pentium processor. When one of these pins is enabled, asserting the pin causes error information to be loaded into machine-check registers and a machine-check exception is generated.

The machine-check exception and machine-check architecture are discussed in detail in Chapter 15, “Machine-Check Architecture.” Also, see the data books for the individual processors for processor-specific hardware information.

### Exception Error Code

None. Error information is provided by machine-check MSRs.

### Saved Instruction Pointer

For the Pentium 4 and Intel Xeon processors, the saved contents of extended machine-check state registers are directly associated with the error that caused the machine-check exception to be generated (see Section 15.3.1.2, “IA32\_MCG\_STATUS MSR,” and Section 15.3.2.6, “IA32\_MCG Extended Machine Check State MSRs”).

For the P6 family processors, if the EIPV flag in the MCG\_STATUS MSR is set, the saved contents of CS and EIP registers are directly associated with the error that caused the machine-check exception to be generated; if the flag is clear, the saved instruction pointer may not be associated with the error (see Section 15.3.1.2, “IA32\_MCG\_STATUS MSR”).

For the Pentium processor, contents of the CS and EIP registers may not be associated with the error.

### Program State Change

The machine-check mechanism is enabled by setting the MCE flag in control register CR4.

For the Pentium 4, Intel Xeon, P6 family, and Pentium processors, a program-state change always accompanies a machine-check exception, and an abort class exception is generated. For abort exceptions, information about the exception can be collected from the machine-check MSRs, but the program cannot generally be restarted.

If the machine-check mechanism is not enabled (the MCE flag in control register CR4 is clear), a machine-check exception causes the processor to enter the shutdown state.

## Interrupt 19—SIMD Floating-Point Exception (#XM)

**Exception Class**     **Fault.**

### Description

Indicates the processor has detected an SSE/SSE2/SSE3 SIMD floating-point exception. The appropriate status flag in the MXCSR register must be set and the particular exception unmasked for this interrupt to be generated.

There are six classes of numeric exception conditions that can occur while executing an SSE/ SSE2/SSE3 SIMD floating-point instruction:

- Invalid operation (#I)
- Divide-by-zero (#Z)
- Denormal operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (Precision) (#P)

The invalid operation, divide-by-zero, and denormal-operand exceptions are pre-computation exceptions; that is, they are detected before any arithmetic operation occurs. The numeric underflow, numeric overflow, and inexact result exceptions are post-computational exceptions.

See “SIMD Floating-Point Exceptions” in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information about the SIMD floating-point exception classes.

When a SIMD floating-point exception occurs, the processor does either of the following things:

- It handles the exception automatically by producing the most reasonable result and allowing program execution to continue undisturbed. This is the response to masked exceptions.
- It generates a SIMD floating-point exception, which in turn invokes a software exception handler. This is the response to unmasked exceptions.

Each of the six SIMD floating-point exception conditions has a corresponding flag bit and mask bit in the MXCSR register. If an exception is masked (the corresponding mask bit in the MXCSR register is set), the processor takes an appropriate automatic default action and continues with the computation. If the exception is unmasked (the corresponding mask bit is clear) and the operating system supports SIMD floating-point exceptions (the OSXM-MEXCPT flag in control register CR4 is set), a software exception handler is invoked through a SIMD floating-point exception. If the exception is unmasked and the OSXMMEXCPT bit is clear (indicating that the operating system does not support unmasked SIMD floating-point exceptions), an invalid opcode exception (#UD) is signaled instead of a SIMD floating-point exception.

Note that because SIMD floating-point exceptions are precise and occur immediately, the situation does not arise where an x87 FPU instruction, a WAIT/FWAIT instruction, or another SSE/SSE2/SSE3 instruction will catch a pending unmasked SIMD floating-point exception.

In situations where a SIMD floating-point exception occurred while the SIMD floating-point exceptions were masked (causing the corresponding exception flag to be set) and the SIMD floating-point exception was subsequently unmasked, then no exception is generated when the exception is unmasked.

When SSE/SSE2/SSE3 SIMD floating-point instructions operate on packed operands (made up of two or four sub-operands), multiple SIMD floating-point exception conditions may be detected. If no more than one exception condition is detected for one or more sets of sub-operands, the exception flags are set for each exception condition detected. For example, an invalid exception detected for one sub-operand will not prevent the reporting of a divide-by-zero exception for another sub-operand. However, when two or more exceptions conditions are generated for one sub-operand, only one exception condition is reported, according to the precedences shown in Table 6-8. This exception precedence sometimes results in the higher priority exception condition being reported and the lower priority exception conditions being ignored.

**Table 6-8. SIMD Floating-Point Exceptions Priority**

Priority	Description
1 (Highest)	Invalid operation exception due to SNaN operand (or any NaN operand for maximum, minimum, or certain compare and convert operations).
2	QNaN operand <sup>1</sup> .
3	Any other invalid operation exception not mentioned above or a divide-by-zero exception <sup>2</sup> .
4	Denormal operand exception <sup>2</sup> .
5	Numeric overflow and underflow exceptions possibly in conjunction with the inexact result exception <sup>2</sup> .
6 (Lowest)	Inexact result exception.

**NOTES:**

1. Though a QNaN this is not an exception, the handling of a QNaN operand has precedence over lower priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a divide-by-zero- exception.
2. If masked, then instruction execution continues, and a lower priority exception can occur as well.

**Exception Error Code**

None.

**Saved Instruction Pointer**

The saved contents of CS and EIP registers point to the SSE/SSE2/SSE3 instruction that was executed when the SIMD floating-point exception was generated. This is the faulting instruction in which the error condition was detected.

**Program State Change**

A program-state change does not accompany a SIMD floating-point exception because the handling of the exception is immediate unless the particular exception is masked. The available state information is often sufficient to allow recovery from the error and re-execution of the faulting instruction if needed.

## Interrupt 20—Virtualization Exception (#VE)

**Exception Class**     **Fault.**

### Description

Indicates that the processor detected an EPT violation in VMX non-root operation. Not all EPT violations cause virtualization exceptions. See Section 24.5.7.2 for details.

The exception handler can recover from EPT violations and restart the program or task without any loss of program continuity. In some cases, however, the problem that caused the EPT violation may be uncorrectable.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception.

### Program State Change

A program-state change does not normally accompany a virtualization exception, because the instruction that causes the exception to be generated is not executed. After the virtualization exception handler has corrected the violation (for example, by executing the EPTP-switching VM function), execution of the program or task can be resumed.

### Additional Exception-Handling Information

The processor saves information about virtualization exceptions in the virtualization-exception information area. See Section 24.5.7.2 for details.

## Interrupt 21—Control Protection Exception (#CP)

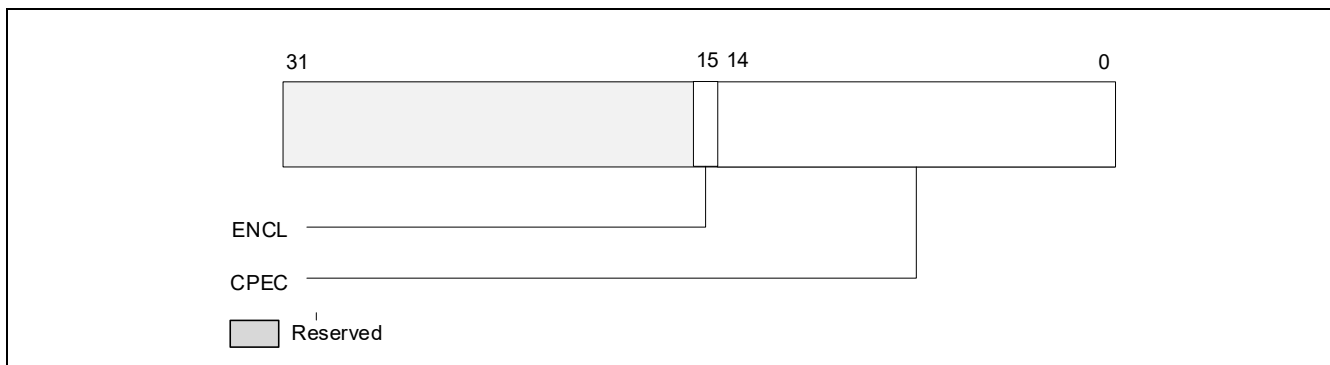
**Exception Class**     **Fault.**

### Description

Indicates a control flow transfer attempt violated the control flow enforcement technology constraints.

### Exception Error Code

Yes (special format). The processor provides the control protection exception handler with following information through the error code on the stack.



**Figure 6-12. Exception Error Code Information**

- Bit 14:0 - CPEC
  - 1 - NEAR-RET: Indicates the #CP was caused by a near RET instruction.
  - 2 - FAR-RET/IRET: Indicates the #CP was caused by a FAR RET or IRET instruction.
  - 3 - ENDBRANCH: indicates the #CP was due to missing ENDBRANCH at target of an indirect call or jump instruction.
  - 4 - RSTORSSP: Indicates the #CP was caused by a shadow-stack-restore token check failure in the RSTORSSP instruction.
  - 5- SETSSBSY: Indicates #CP was caused by a supervisor shadow stack token check failure in the SETSSBSY instruction.
- Bit 15 (ENCL) of the error code, if set to 1, indicates the #CP occurred during enclave execution.

### Saved Instruction Pointer

The saved contents of the CS and EIP registers generally point to the instruction that generated the exception.

### Program State Change

A program-state change does not normally accompany a control protection exception, because the instruction that causes the exception to be generated is not executed.

When a control protection exception is generated during a task switch, the program-state may change as follows. During a task switch, a control protection exception can occur during any of following operations:

- If task switch is initiated by IRET, CS and LIP stored on old task shadow stack do not match CS and LIP of new task (where LIP is the linear address of the return address).
- If task switch is initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is < 3), OR the SSP from IA32\_PL3\_SSP (if new task CPL = 3) is not aligned to 4 bytes or is a value beyond 4GB.

## INTERRUPT AND EXCEPTION HANDLING

In these cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits control protection faults to occur during task-switches, the control protection fault handler should be called through a task gate.

## Interrupts 32 to 255—User Defined Interrupts

**Exception Class**     **Not applicable.**

### Description

Indicates that the processor did one of the following things:

- Executed an `INT n` instruction where the instruction operand is one of the vector numbers from 32 through 255.
- Responded to an interrupt request at the `INTR` pin or from the local APIC when the interrupt vector number associated with the request is from 32 through 255.

### Exception Error Code

Not applicable.

### Saved Instruction Pointer

The saved contents of `CS` and `EIP` registers point to the instruction that follows the `INT n` instruction or instruction following the instruction on which the `INTR` signal occurred.

### Program State Change

A program-state change does not accompany interrupts generated by the `INT n` instruction or the `INTR` signal. The `INT n` instruction generates the interrupt within the instruction stream. When the processor receives an `INTR` signal, it commits all state changes for all previous instructions before it responds to the interrupt; so, program execution can resume upon returning from the interrupt handler.





## 17. Updates to Chapter 9, Volume 3A

Change bars and green text show changes to Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Changes to this chapter: Minor update to table 9-1, "IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT".

This chapter describes the facilities provided for managing processor wide functions and for initializing the processor. The subjects covered include: processor initialization, x87 FPU initialization, processor configuration, feature determination, mode switching, the MSRs (in the Pentium, P6 family, Pentium 4, and Intel Xeon processors), and the MTRRs (in the P6 family, Pentium 4, and Intel Xeon processors).

### 9.1 INITIALIZATION OVERVIEW

Following power-up or an assertion of the RESET# pin, each processor on the system bus performs a hardware initialization of the processor (known as a hardware reset) and an optional built-in self-test (BIST). A hardware reset sets each processor's registers to a known state and places the processor in real-address mode. It also invalidates the internal caches, translation lookaside buffers (TLBs) and the branch target buffer (BTB). At this point, the action taken depends on the processor family:

- **Pentium 4 processors (CPUID DisplayFamily 0FH)** — All the processors on the system bus (including a single processor in a uniprocessor system) execute the multiple processor (MP) initialization protocol. The processor that is selected through this protocol as the bootstrap processor (BSP) then immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The application (non-BSP) processors (APs) go into a Wait For Startup IPI (SIPI) state while the BSP is executing initialization code. See Section 8.4, "Multiple-Processor (MP) Initialization," for more details. Note that in a uniprocessor system, the single Pentium 4 or Intel Xeon processor automatically becomes the BSP.
- **IA-32 and Intel 64 processors (CPUID DisplayFamily 06H)** — The action taken is the same as for the Pentium 4 processors (as described in the previous paragraph).
- **Pentium processors** — In either a single- or dual- processor system, a single Pentium processor is always pre-designated as the primary processor. Following a reset, the primary processor behaves as follows in both single- and dual-processor systems. Using the dual-processor (DP) ready initialization protocol, the primary processor immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The secondary processor (if there is one) goes into a halt state.
- **Intel486 processor** — The primary processor (or single processor in a uniprocessor system) immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. (The Intel486 does not automatically execute a DP or MP initialization protocol to determine which processor is the primary processor.)

The software-initialization code performs all system-specific initialization of the BSP or primary processor and the system logic.

At this point, for MP (or DP) systems, the BSP (or primary) processor wakes up each AP (or secondary) processor to enable those processors to execute self-configuration code.

When all processors are initialized, configured, and synchronized, the BSP or primary processor begins executing an initial operating-system or executive task.

The x87 FPU is also initialized to a known state during hardware reset. x87 FPU software initialization code can then be executed to perform operations such as setting the precision of the x87 FPU and the exception masks. No special initialization of the x87 FPU is required to switch operating modes.

Asserting the INIT# pin on the processor invokes a similar response to a hardware reset. The major difference is that during an INIT, the internal caches, MSRs, MTRRs, and x87 FPU state are left unchanged (although, the TLBs and BTB are invalidated as with a hardware reset). An INIT provides a method for switching from protected to real-address mode while maintaining the contents of the internal caches.

### 9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor in real-address mode with paging disabled.

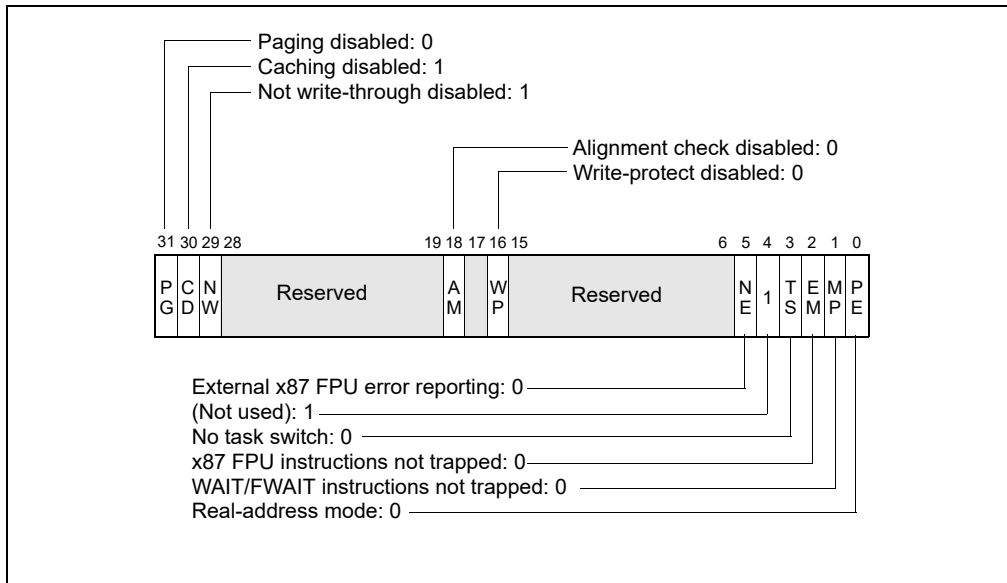


Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 21.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

Register	Power up	Reset	INIT
EFLAGS <sup>1</sup>	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H <sup>2</sup>	60000010H <sup>2</sup>	60000010H <sup>2</sup>
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH <sup>3</sup>	000n06xxH <sup>3</sup>	000n06xxH <sup>3</sup>
EAX	0 <sup>4</sup>	0 <sup>4</sup>	0 <sup>4</sup>
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7 <sup>5</sup>	+0.0	+0.0	FINIT/FNINIT: Unchanged

**Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)**

Register	Power up	Reset	INIT
x87 FPU Control Word <sup>5</sup>	0040H	0040H	FINIT/FNINIT: 037FH
x87 FPU Status Word <sup>5</sup>	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Tag Word <sup>5</sup>	5555H	5555H	FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors <sup>5</sup>	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers <sup>5</sup>	00000000H	00000000H	FINIT/FNINIT: 00000000H
MM0 through MM7 <sup>5</sup>	0000000000000000H	0000000000000000H	INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	0H	0H	Unchanged
MXCSR	1F80H	1F80H	Unchanged
GDTR, IDTR	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H
DR7	00000400H	00000400H	00000400H
R8-R15	0000000000000000H	0000000000000000H	0000000000000000H
XMM8-XMM15	0H	0H	Unchanged
XCRO	1H	1H	Unchanged
IA32_XSS	0H	0H	Unchanged
YMM_H[255:128]	0H	0H	Unchanged
BNDCFGU	0H	0H	0H
BND0-BND3	0H	0H	0H
IA32_BNDCFGS	0H	0H	0H
OPMASK	0H	0H	Unchanged
ZMM_H[511:256]	0H	0H	Unchanged
ZMMHi16[511:0]	0H	0H	Unchanged
PKRU	0H	0H	Unchanged
Intel Processor Trace MSRs	0H	0H <sup>w</sup>	Unchanged
Time-Stamp Counter	0H	0H <sup>w</sup>	Unchanged
IA32_TSC_AUX	0H	0H	Unchanged
IA32_TSC_ADJUST	0H	0H	Unchanged
IA32_TSC_DEADLINE	0H	0H	Unchanged
IA32_SYSENTER_CS/ESP/EIP	0H	0H	Unchanged
IA32_EFER	0000000000000000H	0000000000000000H	0000000000000000H
IA32_STAR/LSTAR	0H	0H	Unchanged
IA32_FS_BASE/GS_BASE	0H	0H	0H

**Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)**

Register	Power up	Reset	INIT
IA32_PMCx, IA32_PERFEVTSELx	0H	0H	Unchanged
IA32_FIXED_CTRx, IA32_FIXED_CTR_CTRL, Global Perf Counter Controls	0H	0H	Unchanged
Data and Code Cache, TLBs	Invalid <sup>6</sup>	Invalid <sup>6</sup>	Unchanged
Fixed MTRRs	Disabled	Disabled	Unchanged
Variable MTRRs	Disabled	Disabled	Unchanged
Machine-Check Banks	Undefined	Undefined <sup>w</sup>	Unchanged
Last Branch Record Stack	0	0 <sup>w</sup>	Unchanged
APIC	Enabled	Enabled	Unchanged
X2APIC	Disabled	Disabled	Unchanged
IA32_DEBUG_INTERFACE	0	0 <sup>w</sup>	Unchanged

**NOTES:**

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
  2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
  3. Where “n” is the Extended Model Value for the respective processor, and “xx” = don’t care.
  4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
  5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
  6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
- W: Warm RESET behavior differs from power-on RESET with details listed in Table 9-2.

**Table 9-2. Variance of RESET Values in Selected Intel Architecture Processors**

State	XREF	Value	Feature Flag or DisplayFamily_DisplayModel Signatures
Time-Stamp Counter	Warm RESET	Unmodified across warm Reset	06_2DH, 06_3EH
Machine-Check Banks	Warm RESET	IA32_MCi_Status banks are unmodified across warm Reset	06_2DH, 06_3EH, 06_3FH, 06_4FH, 06_56H
Last Branch Record Stack	Warm RESET	LBR stack MSRs are unmodified across warm Reset	06_1AH, 06_1CH, DisplayFamiy= 06 and DisplayModel > 1DH
Intel Processor Trace MSRs	Warm RESET	Clears IA32_RTIT_CTL.TraceEn, the rest of MSRs are unmodified	If CPUID.(EAX=14H, ECX=0H):EBX[bit 2] = 1
IA32_DEBUG_INTERFACE	Warm RESET	Unmodified across warm Reset	If CPUID.01H:ECX.[1 1] = 1

## 9.1.2 Processor Built-In Self-Test (BIST)

Hardware may request that the BIST be performed at power-up. The EAX register is cleared (0H) if the processor passes the BIST. A nonzero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

The overhead for performing a BIST varies between processor families. For example, the BIST takes approximately 30 million processor clock periods to execute on the Pentium 4 processor. This clock count is model-specific; Intel reserves the right to change the number of periods for any Intel 64 or IA-32 processor, without notification.

## 9.1.3 Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 9-2). For example, the model, family, and processor type returned for the first processor in the Intel Pentium 4 family is as follows: model (0000B), family (1111B), and processor type (00B).

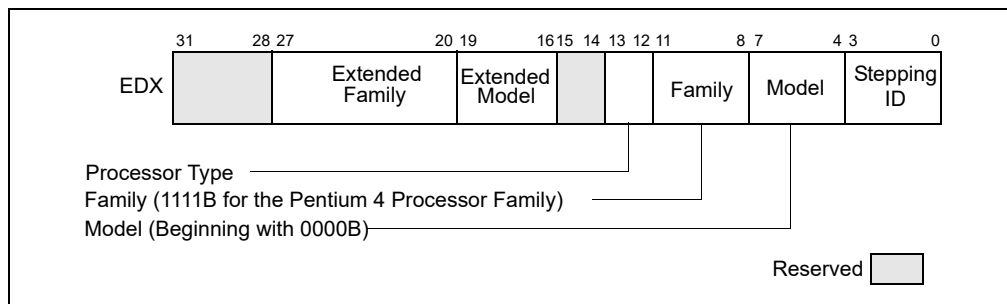


Figure 9-2. Version Information in the EDX Register after Reset

The stepping ID field contains a unique identifier for the processor's stepping ID or revision level. The extended family and extended model fields were added to the IA-32 architecture in the Pentium 4 processors.

## 9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor's uppermost physical address. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector \* 16]). To ensure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

## 9.2 X87 FPU INITIALIZATION

Software-initialization code can determine whether the processor contains an x87 FPU by using the CPUID instruction. The code must then initialize the x87 FPU and set flags in control register CR0 to reflect the state of the x87 FPU environment.

A hardware reset places the x87 FPU in the state shown in Table 9-1. This state is different from the state the x87 FPU is placed in following the execution of an FINIT or FNINIT instruction (also shown in Table 9-1). If the x87 FPU is to be used, the software-initialization code should execute an FINIT/FNINIT instruction following a hardware reset. These instructions, tag all data registers as empty, clear all the exception masks, set the TOP-of-stack value to 0, and select the default rounding and precision controls setting (round to nearest and 64-bit precision).

If the processor is reset by asserting the INIT# pin, the x87 FPU state is not changed.

### 9.2.1 Configuring the x87 FPU Environment

Initialization code must load the appropriate values into the MP, EM, and NE flags of control register CR0. These bits are cleared on hardware reset of the processor. Figure 9-3 shows the suggested settings for these flags, depending on the IA-32 processor being initialized. Initialization code can test for the type of processor present before setting or clearing these flags.

**Table 9-3. Recommended Settings of EM and MP Flags on IA-32 Processors**

EM	MP	NE	IA-32 processor
1	0	1	Intel486™ SX, Intel386™ DX, and Intel386™ SX processors only, without the presence of a math coprocessor.
0	1	1 or 0*	Pentium 4, Intel Xeon, P6 family, Pentium, Intel486™ DX, and Intel 487 SX processors, and Intel386 DX and Intel386 SX processors when a companion math coprocessor is present.
0	1	1 or 0*	More recent Intel 64 or IA-32 processors

**NOTE:**

\* The setting of the NE flag depends on the operating system being used.

The EM flag determines whether floating-point instructions are executed by the x87 FPU (EM is cleared) or a device-not-available exception (#NM) is generated for all floating-point instructions so that an exception handler can emulate the floating-point operation (EM = 1). Ordinarily, the EM flag is cleared when an x87 FPU or math coprocessor is present and set if they are not present. If the EM flag is set and no x87 FPU, math coprocessor, or floating-point emulator is present, the processor will hang when a floating-point instruction is executed.

The MP flag determines whether WAIT/FWAIT instructions react to the setting of the TS flag. If the MP flag is clear, WAIT/FWAIT instructions ignore the setting of the TS flag; if the MP flag is set, they will generate a device-not-available exception (#NM) if the TS flag is set. Generally, the MP flag should be set for processors with an integrated x87 FPU and clear for processors without an integrated x87 FPU and without a math coprocessor present. However, an operating system can choose to save the floating-point context at every context switch, in which case there would be no need to set the MP bit.

Table 2-2 shows the actions taken for floating-point and WAIT/FWAIT instructions based on the settings of the EM, MP, and TS flags.

The NE flag determines whether unmasked floating-point exceptions are handled by generating a floating-point error exception internally (NE is set, native mode) or through an external interrupt (NE is cleared). In systems where an external interrupt controller is used to invoke numeric exception handlers (such as MS-DOS-based systems), the NE bit should be cleared.

### 9.2.2 Setting the Processor for x87 FPU Software Emulation

Setting the EM flag causes the processor to generate a device-not-available exception (#NM) and trap to a software exception handler whenever it encounters a floating-point instruction. (Table 9-3 shows when it is appropriate to use this flag.) Setting this flag has two functions:

- It allows x87 FPU code to run on an IA-32 processor that has neither an integrated x87 FPU nor is connected to an external math coprocessor, by using a floating-point emulator.
- It allows floating-point code to be executed using a special or nonstandard floating-point emulator, selected for a particular application, regardless of whether an x87 FPU or math coprocessor is present.



To emulate floating-point instructions, the EM, MP, and NE flag in control register CR0 should be set as shown in Table 9-4.

**Table 9-4. Software Emulation Settings of EM, MP, and NE Flags**

CRO Bit	Value
EM	1
MP	0
NE	1

Regardless of the value of the EM bit, the Intel486 SX processor generates a device-not-available exception (#NM) upon encountering any floating-point instruction.

## 9.3 CACHE ENABLING

IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors contain internal instruction and data caches. These caches are enabled by clearing the CD and NW flags in control register CR0. (They are set during a hardware reset.) Because all internal cache lines are invalid following reset initialization, it is not necessary to invalidate the cache before enabling caching. Any external caches may require initialization and invalidation using a system-specific initialization and invalidation code sequence.

Depending on the hardware and operating system or executive requirements, additional configuration of the processor's caching facilities will probably be required. Beginning with the Intel486 processor, page-level caching can be controlled with the PCD and PWT flags in page-directory and page-table entries. Beginning with the P6 family processors, the memory type range registers (MTRRs) control the caching characteristics of the regions of physical memory. (For the Intel486 and Pentium processors, external hardware can be used to control the caching characteristics of regions of physical memory.) See Chapter 11, "Memory Cache Control," for detailed information on configuration of the caching facilities in the Pentium 4, Intel Xeon, and P6 family processors and system memory.

## 9.4 MODEL-SPECIFIC REGISTERS (MSRS)

Most IA-32 processors (starting from Pentium processors) and Intel 64 processors contain a model-specific registers (MSRs). A given MSR may not be supported across all families and models for Intel 64 and IA-32 processors. Some MSRs are designated as architectural to simplify software programming; a feature introduced by an architectural MSR is expected to be supported in future processors. Non-architectural MSRs are not guaranteed to be supported or to have the same functions on future processors.

MSRs that provide control for a number of hardware and software-related features, include:

- Performance-monitoring counters (see Chapter 18, "Performance Monitoring").
- Debug extensions (see Chapter 17, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features").
- Machine-check exception capability and its accompanying machine-check architecture (see Chapter 15, "Machine-Check Architecture").
- MTRRs (see Section 11.11, "Memory Type Range Registers (MTRRs)").
- Thermal and power management.
- Instruction-specific support (for example: SYSENTER, SYSEXIT, SWAPGS, etc.).
- Processor feature/mode support (for example: IA32\_EFER, IA32\_FEATURE\_CONTROL).

The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

When performing software initialization of an IA-32 or Intel 64 processor, many of the MSRs will need to be initialized to set up things like performance-monitoring events, run-time machine checks, and memory types for physical memory.

Lists of available performance-monitoring events can be found at: <https://perfmon-events.intel.com/>, and lists of available MSRs are given in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*. The references earlier in this section show where the functions of the various groups of MSRs are described in this manual.

## 9.5 MEMORY TYPE RANGE REGISTERS (MTRRS)

Memory type range registers (MTRRs) were introduced into the IA-32 architecture with the Pentium Pro processor. They allow the type of caching (or no caching) to be specified in system memory for selected physical address ranges. They allow memory accesses to be optimized for various types of memory such as RAM, ROM, frame buffer memory, and memory-mapped I/O devices.

In general, initializing the MTRRs is normally handled by the software initialization code or BIOS and is not an operating system or executive function. At the very least, all the MTRRs must be cleared to 0, which selects the uncached (UC) memory type. See Section 11.11, “Memory Type Range Registers (MTRRs),” for detailed information on the MTRRs.

## 9.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS

For processors that contain SSE/SSE2/SSE3/SSSE3 extensions, steps must be taken when initializing the processor to allow execution of these instructions.

1. Check the CPUID feature flags for the presence of the SSE/SSE2/SSE3/SSSE3 extensions (respectively: EDX bits 25 and 26, ECX bit 0 and 9) and support for the FXSAVE and FXRSTOR instructions (EDX bit 24). Also check for support for the CLFLUSH instruction (EDX bit 19). The CPUID feature flags are loaded in the EDX and ECX registers when the CPUID instruction is executed with a 1 in the EAX register.
2. Set the OSFXSR flag (bit 9 in control register CR4) to indicate that the operating system supports saving and restoring the SSE/SSE2/SSE3/SSSE3 execution environment (XMM and MXCSR registers) with the FXSAVE and FXRSTOR instructions, respectively. See Section 2.5, “Control Registers,” for a description of the OSFXSR flag.
3. Set the OSXMMEXCPT flag (bit 10 in control register CR4) to indicate that the operating system supports the handling of SSE/SSE2/SSE3 SIMD floating-point exceptions (#XM). See Section 2.5, “Control Registers,” for a description of the OSXMMEXCPT flag.
4. Set the mask bits and flags in the MXCSR register according to the mode of operation desired for SSE/SSE2/SSE3 SIMD floating-point instructions. See “MXCSR Control and Status Register” in Chapter 10, “Programming with Streaming SIMD Extensions (SSE),” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a detailed description of the bits and flags in the MXCSR register.

## 9.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION

Following a hardware reset (either through a power-up or the assertion of the RESET# pin) the processor is placed in real-address mode and begins executing software initialization code from physical address FFFFFFF0H. Software initialization code must first set up the necessary data structures for handling basic system functions, such as a real-mode IDT for handling interrupts and exceptions. If the processor is to remain in real-address mode, software must then load additional operating-system or executive code modules and data structures to allow reliable execution of application programs in real-address mode.

If the processor is going to operate in protected mode, software must load the necessary data structures to operate in protected mode and then switch to protected mode. The protected-mode data structures that must be loaded are described in Section 9.8, “Software Initialization for Protected-Mode Operation.”

### 9.7.1 Real-Address Mode IDT

In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the “interrupt vector table”). By default, the address of the base of the IDT is physical address 0H. This address can be

changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled.

The actual interrupt- and exception-handler code can be contained either in EPROM or RAM; however, the code must be located within the 1-MByte addressable range of the processor in real-address mode. If the handler code is to be stored in RAM, it must be loaded along with the IDT.

## 9.7.2 NMI Interrupt Handling

The NMI interrupt is always enabled (except when multiple NMIs are nested). If the IDT and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following hardware reset when an NMI interrupt cannot be handled. During this time, hardware must provide a mechanism to prevent an NMI interrupt from halting code execution until the IDT and the necessary NMI handler software is loaded. Here are two examples of how NMIs can be handled during the initial states of processor initialization:

- A simple IDT and NMI interrupt handler can be provided in EPROM. This allows an NMI interrupt to be handled immediately after reset initialization.
- The system hardware can provide a mechanism to enable and disable NMIs by passing the NMI# signal through an AND gate controlled by a flag in an I/O port. Hardware can clear the flag when the processor is reset, and software can set the flag when it is ready to handle NMI interrupts.

## 9.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION

The processor is placed in real-address mode following a hardware reset. At this point in the initialization process, some basic data structures and code modules must be loaded into physical memory to support further initialization of the processor, as described in Section 9.7, "Software Initialization for Real-Address Mode Operation." Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:

- A IDT.
- A GDT.
- A TSS.
- (Optional) An LDT.
- If paging is to be used, at least one page directory and one page table.
- A code segment that contains the code to be executed when the processor switches to protected mode.
- One or more code modules that contain the necessary interrupt and exception handlers.

Software initialization code must also initialize the following system registers before the processor can be switched to protected mode:

- The GDTR.
- (Optional.) The IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
- Control registers CR1 through CR4.
- (Pentium 4, Intel Xeon, and P6 family processors only.) The memory type range registers (MTRRs).

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by loading control register CR0 with a value that sets the PE flag (bit 0).

### 9.8.1 Protected-Mode System Data Structures

The contents of the protected-mode system data structures loaded into memory during software initialization, depend largely on the type of memory management the protected-mode operating-system or executive is going to support: flat, flat with paging, segmented, or segmented with paging.

To implement a flat memory model without paging, software initialization code must at a minimum load a GDT with one code and one data-segment descriptor. A null descriptor in the first GDT entry is also required. The stack can be placed in a normal read/write data segment, so no dedicated descriptor for the stack is required. A flat memory model with paging also requires a page directory and at least one page table (unless all pages are 4 MBytes in which case only a page directory is required). See Section 9.8.3, "Initializing Paging."

Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

A multi-segmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Some operating systems allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment. However, many operating systems use a single LDT for all tasks, allocating GDT entries in advance. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a real-time system.

### 9.8.2 Initializing Protected-Mode Exceptions and Interrupts

Software initialization code must at a minimum load a protected-mode IDT with gate descriptor for each exception vector that the processor can generate. If interrupt or trap gates are used, the gate descriptors can all point to the same code segment, which contains the necessary exception handlers. If task gates are used, one TSS and accompanying code, data, and task segments are required for each exception handler called with a task gate.

If hardware allows interrupts to be generated, gate descriptors must be provided in the IDT for one or more interrupt handlers.

Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction. This operation is typically carried out immediately after switching to protected mode.

### 9.8.3 Initializing Paging

Paging is controlled by the PG flag in control register CR0. When this flag is clear (its state following a hardware reset), the paging mechanism is turned off; when it is set, paging is enabled. Before setting the PG flag, the following data structures and registers must be initialized:

- Software must load at least one page directory and one page table into physical memory. The page table can be eliminated if the page directory contains a directory entry pointing to itself (here, the page directory and page table reside in the same page), or if only 4-MByte pages are used.
- Control register CR3 (also called the PDBR register) is loaded with the physical base address of the page directory.
- (Optional) Software may provide one set of code and data descriptors in the GDT or in an LDT for supervisor mode and another set for user mode.

With this paging initialization complete, paging is enabled and the processor is switched to protected mode at the same time by loading control register CR0 with an image in which the PG and PE flags are set. (Paging cannot be enabled before the processor is switched to protected mode.)

### 9.8.4 Initializing Multitasking

If the multitasking mechanism is not going to be used and changes between privilege levels are not allowed, it is not necessary to load a TSS into memory or to initialize the task register.

If the multitasking mechanism is going to be used and/or changes between privilege levels are allowed, software initialization code must load at least one TSS and an accompanying TSS descriptor. (A TSS is required to change privilege levels because pointers to the privileged-level 0, 1, and 2 stack segments and the stack pointers for these stacks are obtained from the TSS.) TSS descriptors must not be marked as busy when they are created; they should be marked busy by the processor only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT.

After the processor has switched to protected mode, the LTR instruction can be used to load a segment selector for a TSS descriptor into the task register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The processor can, however, use the TSS to locate pointers to privilege-level 0, 1, and 2 stacks. The segment selector for the TSS must be loaded before software performs its first task switch in protected mode, because a task switch copies the current task state into the TSS.

After the LTR instruction has been executed, further operations on the task register are performed by task switching. As with other segments and LDTs, TSSs and TSS descriptors can be either pre-allocated or allocated as needed.

## 9.8.5 Initializing IA-32e Mode

On Intel 64 processors, the IA32\_EFER MSR is cleared on system reset. The operating system must be in protected mode with paging enabled before attempting to initialize IA-32e mode. IA-32e mode operation also requires physical-address extensions with four or five levels of enhanced paging structures (see Section 4.5, “4-Level Paging and 5-Level Paging”).

Operating systems should follow this sequence to initialize IA-32e mode:

1. Starting from protected mode, disable paging by setting CR0.PG = 0. Use the MOV CR0 instruction to disable paging (the instruction must be located in an identity-mapped page).
2. Enable physical-address extensions (PAE) by setting CR4.PAE = 1. Failure to enable PAE will result in a #GP fault when an attempt is made to initialize IA-32e mode.
3. Load CR3 with the physical base address of the Level 4 page map table (PML4) or Level 5 page map table (PML5).
4. Enable IA-32e mode by setting IA32\_EFER.LME = 1.
5. Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32\_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page (until such time that a branch to non-identity mapped pages can be effected).

64-bit mode paging structures must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode (setting CR0.PG = 1 to enable paging). Because MOV CR3 is executed in protected mode, only the lower 32 bits of the register are written, limiting the table location to the low 4 GBytes of memory. Software can relocate the page tables anywhere in physical memory after IA-32e mode is activated.

The processor performs 64-bit mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating IA-32e mode (IA32\_EFER.LME, CR0.PG, and CR4.PAE). It will generate a general protection fault (#GP) if consistency checks fail. 64-bit mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

64-bit mode consistency checks fail in the following circumstances:

- An attempt is made to enable or disable IA-32e mode while paging is enabled.
- IA-32e mode is enabled and an attempt is made to enable paging prior to enabling physical-address extensions (PAE).
- IA-32e mode is active and an attempt is made to disable physical-address extensions (PAE).
- If the current CS has the L-bit set on an attempt to activate IA-32e mode.
- If the TR contains a 16-bit TSS on an attempt to activate IA-32e mode.

### 9.8.5.1 IA-32e Mode System Data Structures

After activating IA-32e mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy protected-mode descriptor tables. Tables referenced by the descriptors all reside in the lower 4 GBytes of linear-address space. After activating IA-32e mode, 64-bit operating-systems should use the LGDT, LLDT, LIDT, and LTR instructions to load the system-descriptor-table registers with references to 64-bit descriptor tables.

### 9.8.5.2 IA-32e Mode Interrupts and Exceptions

Software must not allow exceptions or interrupts to occur between the time IA-32e mode is activated and the update of the interrupt-descriptor-table register (IDTR) that establishes references to a 64-bit interrupt-descriptor table (IDT). This is because the IDT remains in legacy form immediately after IA-32e mode is activated.

If an interrupt or exception occurs prior to updating the IDTR, a legacy 32-bit interrupt gate will be referenced and interpreted as a 64-bit interrupt gate with unpredictable results. External interrupts can be disabled by using the CLI instruction.

Non-maskable interrupts (NMI) must be disabled using external hardware.

### 9.8.5.3 64-bit Mode and Compatibility Mode Operation

IA-32e mode uses two code segment-descriptor bits (CS.L and CS.D, see Figure 3-8) to control the operating modes after IA-32e mode is initialized. If CS.L = 1 and CS.D = 0, the processor is running in 64-bit mode. With this encoding, the default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, operand size can be changed to 64 bits or 16 bits; address size can be changed to 32 bits.

When IA-32e mode is active and CS.L = 0, the processor operates in compatibility mode. In this mode, CS.D controls default operand and address sizes exactly as it does in the IA-32 architecture. Setting CS.D = 1 specifies default operand and address size as 32 bits. Clearing CS.D to 0 specifies default operand and address size as 16 bits (the CS.L = 1, CS.D = 1 bit combination is reserved).

Compatibility mode execution is selected on a code-segment basis. This mode allows legacy applications to coexist with 64-bit applications running in 64-bit mode. An operating system running in IA-32e mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor's CS.L bit to 0.

In compatibility mode, the following system-level mechanisms continue to operate using the IA-32e-mode architectural semantics:

- Linear-to-physical address translation uses the 64-bit mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the 64-bit mode mechanisms.
- System calls (calls through call gates and SYSENTER/SYSEXIT) are handled using the IA-32e mode mechanisms.

### 9.8.5.4 Switching Out of IA-32e Mode Operation

To return from IA-32e mode to paged-protected mode operation operating systems must use the following sequence:

1. Switch to compatibility mode.
2. Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32\_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and subsequent instructions must be located in an identity-mapped page.
3. Load CR3 with the physical base address of the legacy page-table-directory base address.
4. Disable IA-32e mode by setting IA32\_EFER.LME = 0.
5. Enable legacy paged-protected mode by setting CR0.PG = 1
6. A branch instruction must follow the MOV CR0 that enables paging. Both the MOV CR0 and the branch instruction must be located in an identity-mapped page.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.



## 9.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

### 9.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 9.8, “Software Initialization for Protected-Mode Operation.” Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To ensure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)
5. The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.
6. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
7. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
8. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
9. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
  - Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
  - Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
10. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
11. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.

## 9.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
  - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
  - Ensure that the GDT and IDT are in identity mapped pages.
  - Clear the PG bit in the CR0 register.
  - Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
  - Limit = 64 KBytes (0FFFFH)
  - Byte granular (G = 0)
  - Expand up (E = 0)
  - Writable (W = 1)
  - Present (P = 1)
  - Base = any value

The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.

5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

### NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

## 9.10 INITIALIZATION AND MODE SWITCHING EXAMPLE

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.



- Load the necessary protected-mode system data structures into RAM.
- Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
- Switch the processor to protected mode.

Figure 9-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor’s physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 9-5. The source listing for the example (with the filename STARTUP.ASM) is given in Example 9-1. The line numbers given in Table 9-5 refer to the source listing.

The following are some additional notes concerning this example:

- When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
- Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
- The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
- This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.

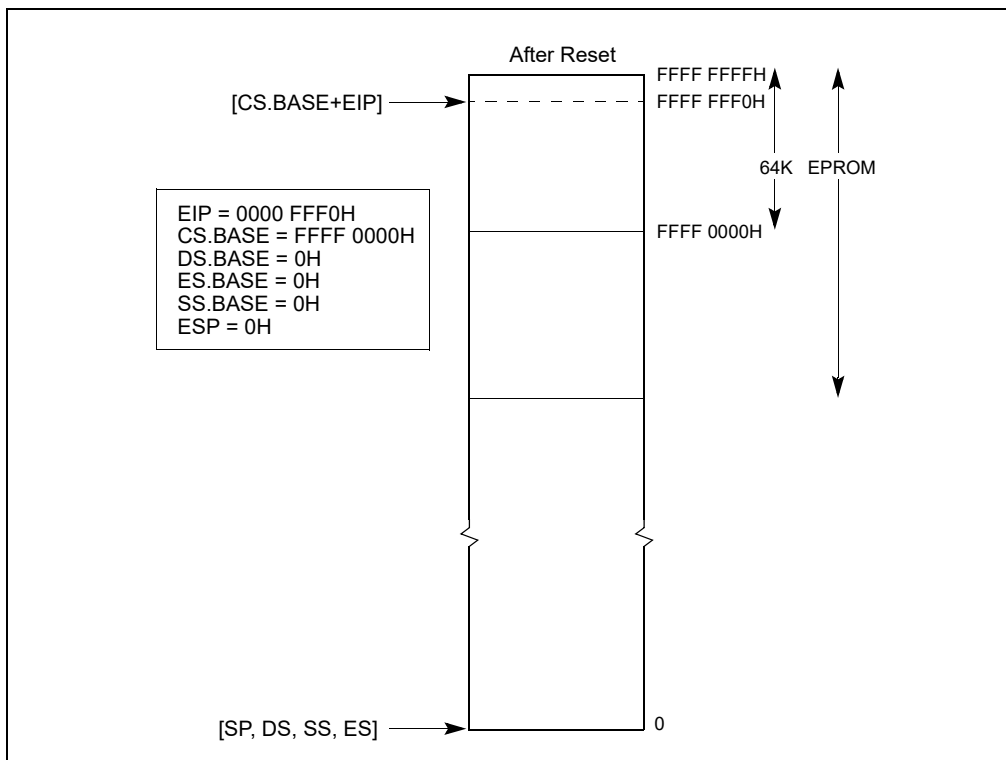


Figure 9-3. Processor State After Reset

**Table 9-5. Main Initialization Steps in STARTUP.ASM Source Listing**

STARTUP.ASM Line Numbers		Description
From	To	
157	157	Jump (short) to the entry code in the EPROM
162	169	Construct a temporary GDT in RAM with one entry: 0 - null 1 - R/W data segment, base = 0, limit = 4 GBytes
171	172	Load the GDTR to point to the temporary GDT
174	177	Load CRO with PE flag set to switch to protected mode
179	181	Jump near to clear real mode instruction queue
184	186	Load DS, ES registers with GDT[1] descriptor, so both point to the entire physical memory space
188	195	Perform specific board initialization that is imposed by the new protected mode
196	218	Copy the application's GDT from ROM into RAM
220	238	Copy the application's IDT from ROM into RAM
241	243	Load application's GDTR
244	245	Load application's IDTR
247	261	Copy the application's TSS from ROM into RAM
263	267	Update TSS descriptor and other aliases in GDT (GDT alias or IDT alias)
277	277	Load the task register (without task switch) using LTR instruction
282	286	Load SS, ESP with the value found in the application's TSS
287	287	Push EFLAGS value found in the application's TSS
288	288	Push CS value found in the application's TSS
289	289	Push EIP value found in the application's TSS
290	293	Load DS, ES with the value found in the application's TSS
296	296	Perform IRET; pop the above values and enter the application code

### 9.10.1 Assembler Usage

In this example, the Intel assembler ASM386 and build tools BLD386 are used to assemble and build the initialization code module. The following assumptions are used when using the Intel ASM386 and BLD386 tools.

- The ASM386 will generate the right operand size opcodes according to the code-segment attribute. The attribute is assigned either by the ASM386 invocation controls or in the code-segment definition.
- If a code segment that is going to run in real-address mode is defined, it must be set to a USE 16 attribute. If a 32-bit operand is used in an instruction in this code segment (for example, MOV EAX, EBX), the assembler automatically generates an operand prefix for the instruction that forces the processor to execute a 32-bit operation, even though its default code-segment attribute is 16-bit.
- Intel's ASM386 assembler allows specific use of the 16- or 32-bit instructions, for example, LGDTW, LGDTD, IRETD. If the generic instruction LGDT is used, the default- segment attribute will be used to generate the right opcode.

### 9.10.2 STARTUP.ASM Listing

Example 9-1 provides high-level sample code designed to move the processor into protected mode. This listing does not include any opcode and offset information.

**Example 9-1. STARTUP.ASM**

MS-DOS\* 5.0(045-N) 386(TM) MACRO ASSEMBLER STARTUP 09:44:51 08/19/92 PAGE 1

MS-DOS 5.0(045-N) 386(TM) MACRO ASSEMBLER V4.0, ASSEMBLY OF MODULE STARTUP  
 OBJECT MODULE PLACED IN startup.obj  
 ASSEMBLER INVOKED BY: f:\386tools\ASM386.EXE startup.a58 pw (132 )

```

LINE      SOURCE

1         NAME      STARTUP
2
3         ;;;;;;;;;;;;;;
4         ;
5         ;   ASSUMPTIONS:
6         ;
7         ;       1.  The bottom 64K of memory is ram, and can be used for
8         ;           scratch space by this module.
9         ;
10        ;       2.  The system has sufficient free usable ram to copy the
11        ;           initial GDT, IDT, and TSS
12        ;
13        ;;;;;;;;;;;;;;
14
15        ; configuration data - must match with build definition
16
17        CS_BASE      EQU      0FFFF0000H
18
19        ; CS_BASE is the linear address of the segment STARTUP_CODE
20        ; - this is specified in the build language file
21
22        RAM_START    EQU      400H
23
24        ; RAM_START is the start of free, usable ram in the linear
25        ; memory space.  The GDT, IDT, and initial TSS will be
26        ; copied above this space, and a small data segment will be
27        ; discarded at this linear address.  The 32-bit word at
28        ; RAM_START will contain the linear address of the first
29        ; free byte above the copied tables - this may be useful if
30        ; a memory manager is used.
31
32        TSS_INDEX    EQU      10
33
34        ; TSS_INDEX is the index of the TSS of the first task to
35        ; run after startup
36
37
38        ;;;;;;;;;;;;;;
39
40        ; ----- STRUCTURES and EQU -----
41        ; structures for system data
42
43        ; TSS structure
44        TASK_STATE   STRUC
45        link         DW ?

```

## PROCESSOR MANAGEMENT AND INITIALIZATION

```
46     link_h     DW ?
47     ESP0      DD ?
48     SS0       DW ?
49     SS0_h     DW ?
50     ESP1      DD ?
51     SS1       DW ?
52     SS1_h     DW ?
53     ESP2      DD ?
54     SS2       DW ?
55     SS2_h     DW ?
56     CR3_reg   DD ?
57     EIP_reg   DD ?
58     EFLAGS_reg DD ?
59     EAX_reg   DD ?
60     ECX_reg   DD ?
61     EDX_reg   DD ?
62     EBX_reg   DD ?
63     ESP_reg   DD ?
64     EBP_reg   DD ?
65     ESI_reg   DD ?
66     EDI_reg   DD ?
67     ES_reg    DW ?
68     ES_h     DW ?
69     CS_reg    DW ?
70     CS_h     DW ?
71     SS_reg    DW ?
72     SS_h     DW ?
73     DS_reg    DW ?
74     DS_h     DW ?
75     FS_reg    DW ?
76     FS_h     DW ?
77     GS_reg    DW ?
78     GS_h     DW ?
79     LDT_reg   DW ?
80     LDT_h     DW ?
81     TRAP_reg  DW ?
82     IO_map_base DW ?
83     TASK_STATE ENDS
84
85     ; basic structure of a descriptor
86     DESC      STRUC
87         lim_0_15 DW ?
88         bas_0_15 DW ?
89         bas_16_23 DB ?
90         access   DB ?
91         gran     DB ?
92         bas_24_31 DB ?
93     DESC      ENDS
94
95     ; structure for use with LGDT and LIDT instructions
96     TABLE_REG STRUC
97         table_lim DW ?
98         table_linear DD ?
99     TABLE_REG ENDS
```

```

100
101 ; offset of GDT and IDT descriptors in builder generated GDT
102 GDT_DESC_OFF EQU 1*SIZE(DESC)
103 IDT_DESC_OFF EQU 2*SIZE(DESC)
104
105 ; equates for building temporary GDT in RAM
106 LINEAR_SEL EQU 1*SIZE(DESC)
107 LINEAR_PROTO_LO EQU 00000FFFFH ; LINEAR_ALIAS
108 LINEAR_PROTO_HI EQU 000CF9200H
109
110 ; Protection Enable Bit in CR0
111 PE_BIT EQU 1B
112
113 ; -----
114
115 ; ----- DATA SEGMENT-----
116
117 ; Initially, this data segment starts at linear 0, according
118 ; to the processor's power-up state.
119
120 STARTUP_DATA SEGMENT RW
121
122 free_mem_linear_base LABEL DWORD
123 TEMP_GDT LABEL BYTE ; must be first in segment
124 TEMP_GDT_NULL_DESC DESC <>
125 TEMP_GDT_LINEAR_DESC DESC <>
126
127 ; scratch areas for LGDT and LIDT instructions
128 TEMP_GDT_SCRATCH TABLE_REG <>
129 APP_GDT_RAM TABLE_REG <>
130 APP_IDT_RAM TABLE_REG <>
131 ; align end_data
132 fill DW ?
133
134 ; last thing in this segment - should be on a dword boundary
135 end_data LABEL BYTE
136
137 STARTUP_DATA ENDS
138 ; -----
139
140
141 ; ----- CODE SEGMENT-----
142 STARTUP_CODE SEGMENT ER PUBLIC USE16
143
144 ; filled in by builder
145 PUBLIC GDT_EPROM
146 GDT_EPROM TABLE_REG <>
147
148 ; filled in by builder
149 PUBLIC IDT_EPROM
150 IDT_EPROM TABLE_REG <>
151
152 ; entry point into startup code - the bootstrap will vector
153 ; here with a near JMP generated by the builder. This

```

## PROCESSOR MANAGEMENT AND INITIALIZATION

```
154 ; label must be in the top 64K of linear memory.
155
156     PUBLIC  STARTUP
157 STARTUP:
158
159 ; DS,ES address the bottom 64K of flat linear memory
160     ASSUME  DS:STARTUP_DATA, ES:STARTUP_DATA
161 ; See Figure 9-4
162 ; load GDTR with temporary GDT
163     LEA    EBX,TEMP_GDT ; build the TEMP_GDT in low ram,
164     MOV    DWORD PTR [EBX],0 ; where we can address
165     MOV    DWORD PTR [EBX]+4,0
166     MOV    DWORD PTR [EBX]+8, LINEAR_PROTO_LO
167     MOV    DWORD PTR [EBX]+12, LINEAR_PROTO_HI
168     MOV    TEMP_GDT_scratch.table_linear,EBX
169     MOV    TEMP_GDT_scratch.table_lim,15
170
171     DB 66H; execute a 32 bit LGDT
172     LGDT  TEMP_GDT_scratch
173
174 ; enter protected mode
175     MOV    EBX,CR0
176     OR    EBX,PE_BIT
177     MOV    CR0,EBX
178
179 ; clear prefetch queue
180     JMP    CLEAR_LABEL
181 CLEAR_LABEL:
182
183 ; make DS and ES address 4G of linear memory
184     MOV    CX,LINEAR_SEL
185     MOV    DS,CX
186     MOV    ES,CX
187
188 ; do board specific initialization
189 ;
190 ;
191 ; .....
192 ;
193
194
195 ; See Figure 9-5
196 ; copy EPROM GDT to ram at:
197 ;          RAM_START + size (STARTUP_DATA)
198     MOV    EAX,RAM_START
199     ADD    EAX,OFFSET (end_data)
200     MOV    EBX,RAM_START
201     MOV    ECX, CS_BASE
202     ADD    ECX, OFFSET (GDT_EPROM)
203     MOV    ESI, [ECX].table_linear
204     MOV    EDI,EAX
205     MOVZX  ECX, [ECX].table_lim
206     MOV    APP_GDT_ram[EBX].table_lim,CX
```

```

207     INC     ECX
208     MOV     EDX,EAX
209     MOV     APP_GDT_ram[EBX].table_linear,EAX
210     ADD     EAX,ECX
211     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
212
213     ; fixup GDT base in descriptor
214     MOV     ECX,EDX
215     MOV     [EDX].bas_0_15+GDT_DESC_OFF,CX
216     ROR     ECX,16
217     MOV     [EDX].bas_16_23+GDT_DESC_OFF,CL
218     MOV     [EDX].bas_24_31+GDT_DESC_OFF,CH
219
220     ; copy EPROM IDT to ram at:
221     ; RAM_START+size(STARTUP_DATA)+SIZE (EPROM GDT)
222     MOV     ECX, CS_BASE
223     ADD     ECX, OFFSET (IDT_EPROM)
224     MOV     ESI, [ECX].table_linear
225     MOV     EDI,EAX
226     MOVZX  ECX, [ECX].table_lim
227     MOV     APP_IDT_ram[EBX].table_lim,CX
228     INC     ECX
229     MOV     APP_IDT_ram[EBX].table_linear,EAX
230     MOV     EBX,EAX
231     ADD     EAX,ECX
232     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
233
234     ; fixup IDT pointer in GDT
235     MOV     [EDX].bas_0_15+IDT_DESC_OFF,BX
236     ROR     EBX,16
237     MOV     [EDX].bas_16_23+IDT_DESC_OFF,BL
238     MOV     [EDX].bas_24_31+IDT_DESC_OFF,BH
239
240     ; load GDTR and IDTR
241     MOV     EBX, RAM_START
242     DB     66H           ; execute a 32 bit LGDT
243     LGDT   APP_GDT_ram[EBX]
244     DB     66H           ; execute a 32 bit LIDT
245     LIDT   APP_IDT_ram[EBX]
246
247     ; move the TSS
248     MOV     EDI,EAX
249     MOV     EBX,TSS_INDEX*SIZE(DESC)
250     MOV     ECX,GDT_DESC_OFF ;build linear address for TSS
251     MOV     GS,CX
252     MOV     DH,GS:[EBX].bas_24_31
253     MOV     DL,GS:[EBX].bas_16_23
254     ROL     EDX,16
255     MOV     DX,GS:[EBX].bas_0_15
256     MOV     ESI,EDX
257     LSL     ECX,EBX
258     INC     ECX
259     MOV     EDX,EAX
260     ADD     EAX,ECX

```

## PROCESSOR MANAGEMENT AND INITIALIZATION

```
261     REP MOVSB   BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
262
263         ; fixup TSS pointer
264     MOV     GS:[EBX].bas_0_15,DX
265     ROL     EDX,16
266     MOV     GS:[EBX].bas_24_31,DH
267     MOV     GS:[EBX].bas_16_23,DL
268     ROL     EDX,16
269     ;save start of free ram at linear location RAMSTART
270     MOV     free_mem_linear_base+RAM_START,EAX
271
272     ;assume no LDT used in the initial task - if necessary,
273     ;code to move the LDT could be added, and should resemble
274     ;that used to move the TSS
275
276     ; load task register
277     LTR     BX ; No task switch, only descriptor loading
278     ; See Figure 9-6
279     ; load minimal set of registers necessary to simulate task
280     ; switch
281
282
283     MOV     AX,[EDX].SS_reg ; start loading registers
284     MOV     EDI,[EDX].ESP_reg
285     MOV     SS,AX
286     MOV     ESP,EDI ; stack now valid
287     PUSH   DWORD PTR [EDX].EFLAGS_reg
288     PUSH   DWORD PTR [EDX].CS_reg
289     PUSH   DWORD PTR [EDX].EIP_reg
290     MOV     AX,[EDX].DS_reg
291     MOV     BX,[EDX].ES_reg
292     MOV     DS,AX ; DS and ES no longer linear memory
293     MOV     ES,BX
294
295     ; simulate far jump to initial task
296     IRETD
297
298     STARTUP_CODE ENDS
*** WARNING #377 IN 298, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
299
300     END STARTUP, DS:STARTUP_DATA, SS:STARTUP_DATA
301
302
```

ASSEMBLY COMPLETE, 1 WARNING, NO ERRORS.



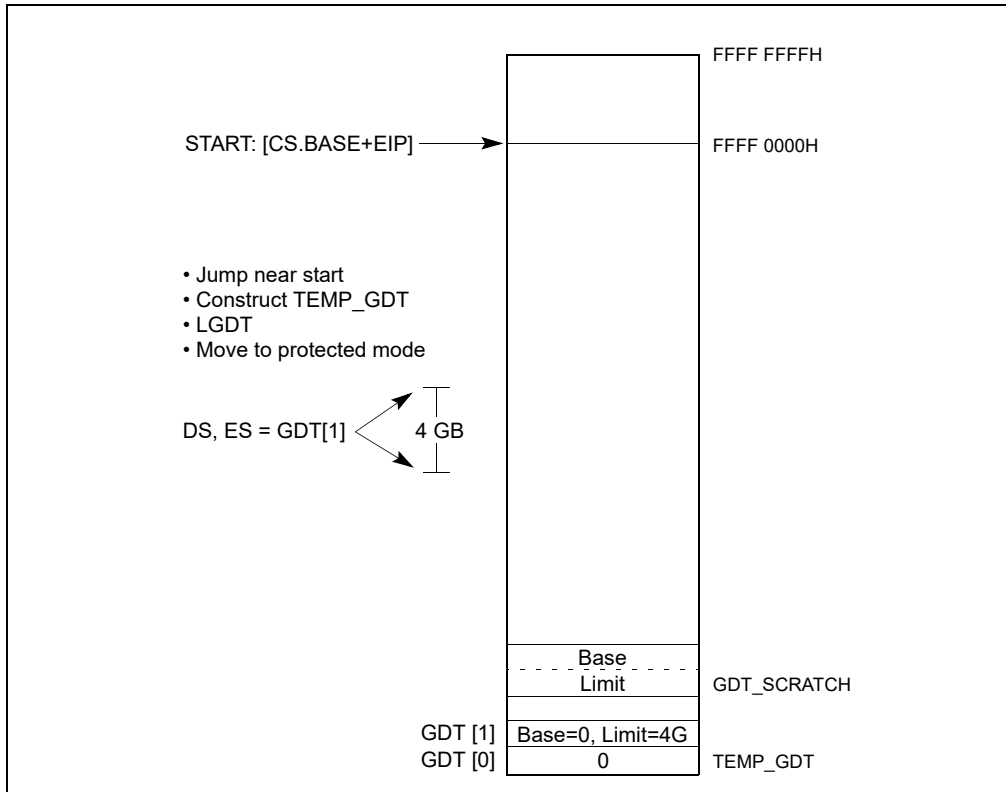


Figure 9-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)

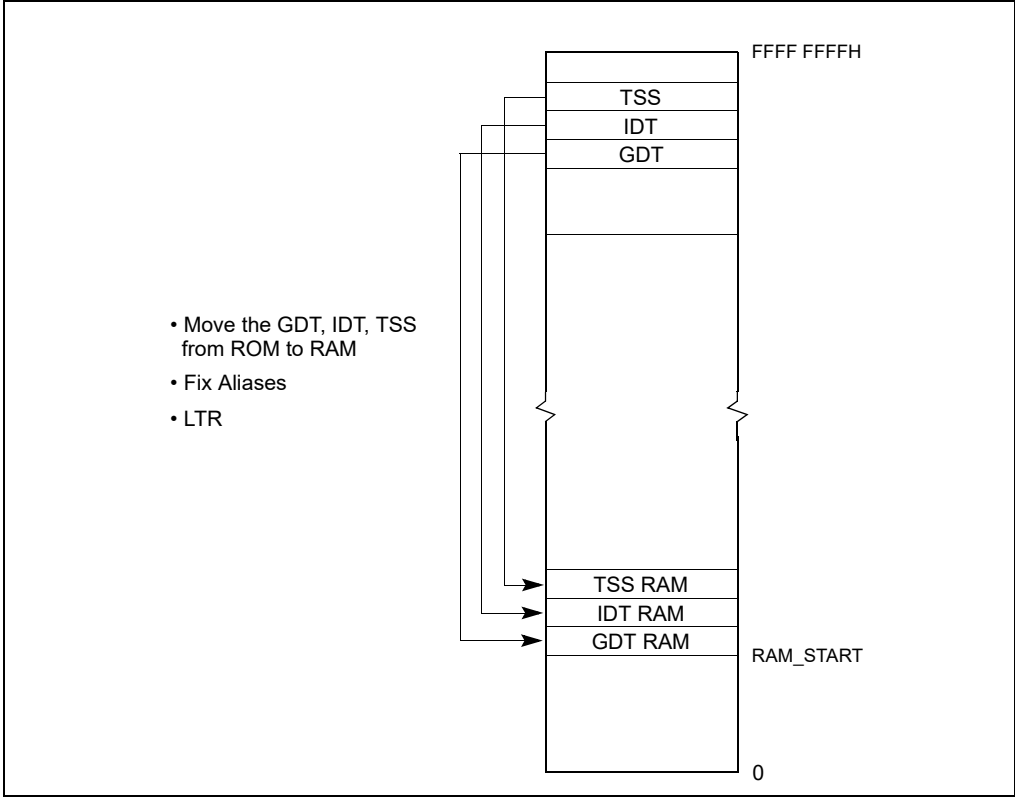


Figure 9-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

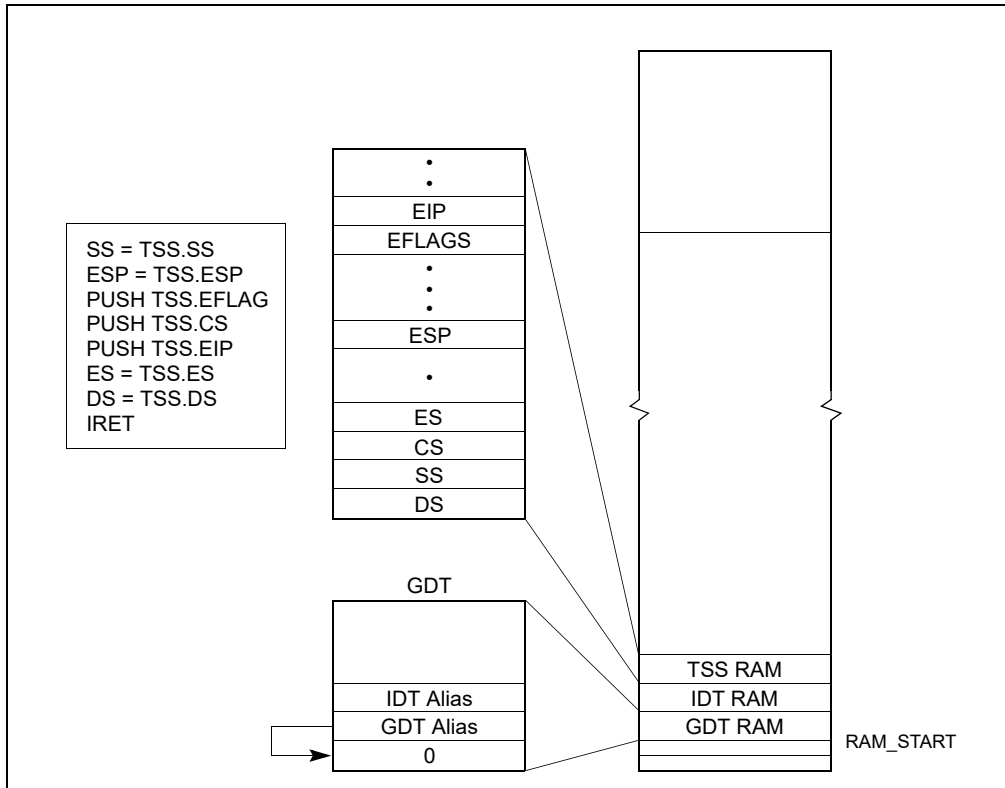


Figure 9-6. Task Switching (Lines 282-296 of List File)

### 9.10.3 MAIN.ASM Source Code

The file MAIN.ASM shown in Example 9-2 defines the data and stack segments for this application and can be substituted with the main module task written in a high-level language that is invoked by the IRET instruction executed by STARTUP.ASM.

#### Example 9-2. MAIN.ASM

```

NAME    main_module
data    SEGMENT RW
        dw 1000 dup(?)
DATA    ENDS
stack   stackseg 800
CODE SEGMENT ER use32 PUBLIC
main_start:
        nop
        nop
        nop
CODE    ENDS
END main_start, ds:data, ss:stack

```

### 9.10.4 Supporting Files

The batch file shown in Example 9-3 can be used to assemble the source code files STARTUP.ASM and MAIN.ASM and build the final application.

**Example 9-3. Batch File to Assemble and Build the Application**

```
ASM386 STARTUP.ASM
ASM386 MAIN.ASM
BLD386 STARTUP.OBJ, MAIN.OBJ buildfile(EPROM.BLD) bootstrap(STARTUP) Bootload
```

BLD386 performs several operations in this example:  
 It allocates physical memory location to segments and tables.  
 It generates tables using the build file and the input files.  
 It links object files and resolves references.  
 It generates a boot-loadable file to be programmed into the EPROM.

Example 9-4 shows the build file used as an input to BLD386 to perform the above functions.

**Example 9-4. Build File**

```
INIT_BLD_EXAMPLE;

SEGMENT
    *SEGMENTS(DPL = 0)
    , startup.startup_code(BASE = 0FFFF0000H)
    ;

TASK
    BOOT_TASK(OBJECT = startup, INITIAL,DPL = 0,
              NOT INTENABLED)
    , PROTECTED_MODE_TASK(OBJECT = main_module,DPL = 0,
                          NOT INTENABLED)
    ;

TABLE
    GDT (
        LOCATION = GDT_EPROM
        , ENTRY = (
            10: PROTECTED_MODE_TASK
            , startup.startup_code
            , startup.startup_data
            , main_module.data
            , main_module.code
            , main_module.stack
            )
        ),
    IDT (
        LOCATION = IDT_EPROM
        );

MEMORY
    (
        RESERVE = (0..3FFFH
                  -- Area for the GDT, IDT, TSS copied from ROM
                  , 60000H..0FFFEFFFFH)
        , RANGE = (ROM_AREA = ROM (0FFFF0000H..0FFFFFFFHH)
                  -- Eprom size 64K
                  , RANGE = (RAM_AREA = RAM (4000H..05FFFFH))
```

);

END

Table 9-6 shows the relationship of each build item with an ASM source file.

**Table 9-6. Relationship Between BLD Item and ASM Source File**

Item	ASM386 and Startup.A58	BLD386 Controls and BLD file	Effect
Bootstrap	public startup startup:	bootstrap start(startup)	Near jump at OFFFFFFFF0H to start.
GDT location	public GDT_EPROM GDT_EPROM TABLE_REG <>	TABLE GDT(location = GDT_EPROM)	The location of the GDT will be programmed into the GDT_EPROM location.
IDT location	public IDT_EPROM IDT_EPROM TABLE_REG <>	TABLE IDT(location = IDT_EPROM)	The location of the IDT will be programmed into the IDT_EPROM location.
RAM start	RAM_START equ 400H	memory (reserve = (0..3FFFH))	RAM_START is used as the ram destination for moving the tables. It must be excluded from the application's segment area.
Location of the application TSS in the GDT	TSS_INDEX EQU 10	TABLE GDT( ENTRY = (10: PROTECTED_MODE_ TASK))	Put the descriptor of the application TSS in GDT entry 10.
EPROM size and location	size and location of the initialization code	SEGMENT startup.code (base = OFFF0000H) ...memory (RANGE(ROM_AREA = ROM(x..y))	Initialization code size must be less than 64K and resides at upper most 64K of the 4-GByte memory space.

## 9.11 MICROCODE UPDATE FACILITIES

The P6 family and later processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 9-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

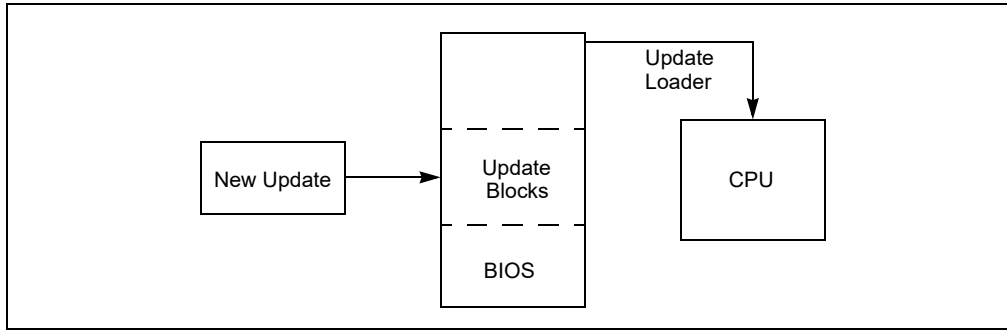


Figure 9-7. Applying Microcode Updates

### 9.11.1 Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor’s signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0FH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 9.11.2 for details).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 9-7 provides a definition of the fields; Table 9-8 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the microcode update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The total size field of the microcode update header specifies the encrypted data size plus the header size; its value must be in multiples of 1024 bytes (1 KBytes). The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

#### NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

Table 9-7. Microcode Update Field Definitions

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g. 07/18/98 is 07181998H).

Table 9-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature	12	4	<i>Extended family, extended model, type, family, model, and stepping</i> of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, <i>type, family, model, and stepping</i> of the processor.  Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all the DWORDs (including the extended Processor Signature Table) that comprise the microcode update result in 00000000H.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table. This value is always a multiple of 1024.
Reserved	36	12	Reserved fields for future expansion.
Update Data	48	Data Size or 2000	Update data.
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields.

**Table 9-7. Microcode Update Field Definitions (Contd.)**

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature[n]	Data Size + 68 + (n * 12)	4	<p><i>Extended family, extended model, type, family, model, and stepping</i> of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, <i>type, family, model</i>, and <i>stepping</i> of the processor.</p> <p>Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.</p>
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	<p>Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table.</p> <p>To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.</p>

**Table 9-8. Microcode Update Format**

31	24	16	8	0	Bytes
<b>Header Version</b>					0
<b>Update Revision</b>					4
Month: 8		Day: 8		Year: 16	8
<b>Processor Signature (CPUID)</b>					12
Res: 4		Extended Mode: 4		Stepping: 4	
Extended Family: 8		Reserved: 2		Type: 2	
Family: 4		Model: 4		Family: 4	
<b>Checksum</b>					16
<b>Loader Revision</b>					20
<b>Processor Flags</b>					24
Reserved (24 bits)					P7 P6 P5 P4 P3 P2 P1 P0
<b>Data Size</b>					28
<b>Total Size</b>					32
<b>Reserved (12 Bytes)</b>					36



**Table 9-8. Microcode Update Format (Contd.)**

31	24	16	8	0	Bytes
Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)					48
Extended Signature Count 'n'					Data Size + 48
Extended Processor Signature Table Checksum					Data Size + 52
Reserved (12 Bytes)					Data Size + 56
Processor Signature[n]					Data Size + 68 + (n * 12)
Processor Flags[n]					Data Size + 72 + (n * 12)
Checksum[n]					Data Size + 76 + (n * 12)

### 9.11.2 Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 9-9). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 9-10).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

**Table 9-9. Extended Processor Signature Table Header Structure**

Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

**Table 9-10. Processor Signature Structure**

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

### 9.11.3 Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the BIOS to reject the update.

Example 9-5 shows how to check for a valid processor signature match between the processor and microcode update.

#### Example 9-5. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion = 00000001h)
{
    // first check the ProcessorSignature field
    If (ProcessorSignature = Update.ProcessorSignature)
        Success

    // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[0].
        //

        For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
            (ProcessorSignature ≠ Update.ProcessorSignature[N])); N++);

            // if the loops ended when the iteration count is
            // less than the number of processor signatures in
            // the table, we have a match
            If (N < Update.ExtendedSignatureCount)
                Success
            Else
                Fail
    }
    Else
        Fail
Else
    Fail
```

### 9.11.4 Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32\_PLATFORM\_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name: IA32\_PLATFORM\_ID  
MSR Address: 017H

Access: Read Only

IA32\_PLATFORM\_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

**Table 9-11. Processor Flags**

Bit	Descriptions																																				
63:53	Reserved																																				
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 9-8.  <table style="margin-left: 40px; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;">52</td> <td style="padding-right: 10px;">51</td> <td style="padding-right: 10px;">50</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>Processor Flag 0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>Processor Flag 1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>Processor Flag 2</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>Processor Flag 3</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>Processor Flag 4</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>Processor Flag 5</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>Processor Flag 6</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>Processor Flag 7</td> </tr> </table>	52	51	50		0	0	0	Processor Flag 0	0	0	1	Processor Flag 1	0	1	0	Processor Flag 2	0	1	1	Processor Flag 3	1	0	0	Processor Flag 4	1	0	1	Processor Flag 5	1	1	0	Processor Flag 6	1	1	1	Processor Flag 7
52	51	50																																			
0	0	0	Processor Flag 0																																		
0	0	1	Processor Flag 1																																		
0	1	0	Processor Flag 2																																		
0	1	1	Processor Flag 3																																		
1	0	0	Processor Flag 4																																		
1	0	1	Processor Flag 5																																		
1	1	0	Processor Flag 6																																		
1	1	1	Processor Flag 7																																		
49:0	Reserved																																				

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 9-6.

#### Example 9-6. Pseudo Code Example of Processor Flags Test

```

Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion = 00000001h)
{
  If (Update.ProcessorFlags & Flag)
  {
    Load Update
  }
  Else
  {

    //
    // Assume the Data Size has been used to calculate the
    // location of Update.ProcessorSignature[N] and a match
    // on Update.ProcessorSignature[N] has already succeeded
    //

    If (Update.ProcessorFlags[n] & Flag)
    {
      Load Update
    }
  }
}

```

### 9.11.5 Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum

procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 9-7 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs =  $(Total\ Size / 4)$ , where the total size is a multiple of 1024 bytes (1 KBytes).

#### Example 9-7. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize ≠ 00000000H)
    N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
    ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum = 00000000H)
    Success
Else
    Fail
```

### 9.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a P6 family or later processors. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

#### Example 9-8. Assembly Code Example of Simple Microcode Update Loader

```
mov  ecx,79h           ; MSR to write in ECX
xor  eax,eax          ; clear EAX
xor  ebx,ebx          ; clear EBX
mov  ax,cs            ; Segment of microcode update
shl  eax,4
mov  bx,offset Update ; Offset of microcode update
add  eax,ebx          ; Linear Address of Update in EAX
add  eax,48d          ; Offset of the Update Data within the Update
xor  edx,edx          ; Zero in EDX
WRMSR                 ; microcode update trigger
```

The loader shown in Example 9-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32\_BIOS\_UPDT\_TRIG).

Other requirements are:

- The addresses for the microcode update data must be in canonical form.
- If paging is enabled, the microcode update data must map that data as present.
- The microcode update data must start at a 16-byte aligned linear address.

### 9.11.6.1 Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

### 9.11.6.2 Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

### 9.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology

Intel Hyper-Threading Technology has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor supporting Intel Hyper-Threading Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g. a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

### 9.11.6.4 Update in a System Supporting Dual-Core Technology

Dual-core technology has implications on the loading of the microcode update. The microcode update facility is not shared between processor cores in the same physical package. The update must be loaded for each core in a physical processor.

If processor core supports Intel Hyper-Threading Technology, the guideline described in Section 9.11.6.3 also applies.

### 9.11.6.5 Update Loader Enhancements

The update loader presented in Section 9.11.6, "Microcode Update Loader," is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.

- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 9.11.7, “Update Signature and Verification.”
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 9.11.5, “Microcode Update Checksum.”
- A loader can provide power-on messages indicating successful loading of an update.

### 9.11.7 Update Signature and Verification

The P6 family and later processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32\_BIOS\_SIGN\_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different stepings.

#### 9.11.7.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 9-9.

#### Example 9-9. Assembly Code to Retrieve the Update Revision

```

MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
XOR    EAX, EAX           ;clear EAX
XOR    EDX, EDX           ;clear EDX
WRMSR                ;Load 0 to MSR at 8BH
MOV    EAX, 1
cpuid
MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
rdmsr                ;Read Model Specific Register
    
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	Microcode Update Signature Register
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32\_BIOS\_SIGN\_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 9-12).

Table 9-12. Microcode Update Signature

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

### 9.11.7.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 9-10.

#### Example 9-10. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
  Load Update that is to be authenticated;
  Y ← Obtain New Signature from MSR 8BH;

  If (Z = Y)
    Success
  Else
    Fail
}
Else
  Fail
```

Example 9-10 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature ( $X$ ) < New Update Revision ( $Z$ ). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

### 9.11.8 Optional Processor Microcode Update Specifications

This section an interface that an OEM-BIOS may provide to its client system software to manage processor microcode updates. System software may choose to build its own facility to manage microcode updates (e.g. similar to the facility described in Section 9.11.6) or rely on a facility provided by the BIOS to perform microcode updates.

Sections 9.11.8.1-9.11.8.9 describes an extension (Function 0D042H) to the real mode INT 15H service. INT 15H 0D042H function is one of several alternatives that a BIOS may choose to implement microcode update facility and offer to its client application (e.g. an OS). Other alternative microcode update facility that BIOS can choose are dependent on platform-specific capabilities, including the Capsule Update mechanism from the UEFI specification ([www.uefi.org](http://www.uefi.org)). In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

### 9.11.8.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

#### NOTES

For IA-32 processors starting with family 0FH and model 03H and Intel 64 processors, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16-KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 9-11) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

#### Example 9-11. Pseudo Code, Checks Required Prior to Loading an Update

```

For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version = 00000001H)
        {

```



```

If ((Update.ProcessorSignature = Processor Signature) &&
    (Update.ProcessorFlags & Platform Bits))
{
    Load Update.UpdateData into the Processor;
    Verify update was correctly loaded into the processor
    Go on to next processor
    Break;
}
Else If (Update.TotalSize > (Update.DataSize + 48))
{
    N ← 0
    While (N < Update.ExtendedSignatureCount)
    {
        If ((Update.ProcessorSignature[N] =
            Processor Signature) &&
            (Update.ProcessorFlags[N] & Platform Bits))
        {
            Load Update.UpdateData into the Processor;
            Verify update correctly loaded into the processor
            Go on to next processor
            Break;
        }
        N ← N + 1
    }
    I ← I + (Update.TotalSize / 2048)
    If ((Update.TotalSize MOD 2048) = 0)
        I ← I + 1
    }
}
}
}
}

```

## NOTES

The platform Id bits in IA32\_PLATFORM\_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

### 9.11.8.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of a calling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an

older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.

- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 9-12 represents a calling program.

**Example 9-12. INT 15 D042 Calling Program Pseudo-code**

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e., Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform
    Id Bits

    If current processor cannot be updated
        exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
        (we have a unique update in the database for this processor))
    {
        Checksum the update from the database;
        If Checksum fails
            exit
        NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}
//
// Do we have enough update slots for all CPUs?
//
```

```

If there are more blocks required to support the unique processor steppings than update blocks
provided by the BIOS exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks = 0)
    exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
    //
    // Load each Update
    //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned
    {
        Display Error -- BIOS is not managing NVRAM appropriately
        exit
    }

    If (INVALID_REVISION) returned
    {
        Display Message: More recent update already loaded in NVRAM for
        this stepping
        continue
    }

    If any other error returned
    {
        Display Diagnostic
        exit
    }

    //
    // Verify the update was loaded correctly
    //
    Issue the ReadUpdate function

    If an error occurred
    {
        Display Diagnostic
        exit
    }
    //
    // Compare the Update read to that written
    //
    If (Update read ≠ Update written)
    {
        Display Diagnostic
        exit
    }

    I ← I + (size of microcode update / 2048)
}
//
// Enable Update Loading, and inform user

```

//  
 Issue the Update Control function with Task = Enable.

### 9.11.8.3 Microcode Update Functions

Table 9-13 defines the processor microcode update functions that implementations of INT 15H 0D042H must support.

**Table 9-13. Microcode Update Functions**

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

### 9.11.8.4 INT 15H-based Interface

If an OEM-BIOS is implementing INT 15H 0D042H interface and offer to its client, the BIOS should allow additional microcode updates to be added to system flash.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 9.11.8.9, "Return Codes."

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT\_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

### 9.11.8.5 Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 9-14 lists the parameters and return codes for the function.

**Table 9-14. Parameters for the Presence Test**

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP' - Part two of the signature
EDX	Loader Version	Version number of the microcode update loader

**Table 9-14. Parameters for the Presence Test (Contd.)**

Input		
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

### 9.11.8.6 Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 9-15 lists the parameters and return codes for the function.

**Table 9-15. Parameters for the Write Update Data Function**

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or...  Real Mode pointer to the Intel Update structure. This buffer is 64 KBytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 KBytes of RAM block
SS:SP	Stack pointer	32 KBytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

**Table 9-15. Parameters for the Write Update Data Function (Contd.)**

Input	
STORAGE_FULL	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	The processor stepping does not currently exist in the system.
INVALID_HEADER	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	The update does not checksum correctly.
SECURITY_FAILURE	The processor rejected the update.
INVALID_REVISION	The same or more recent revision of the update exists in the storage device.

**Description**

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 9.11.6, "Microcode Update Loader." This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 9-8 and Figure 9-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

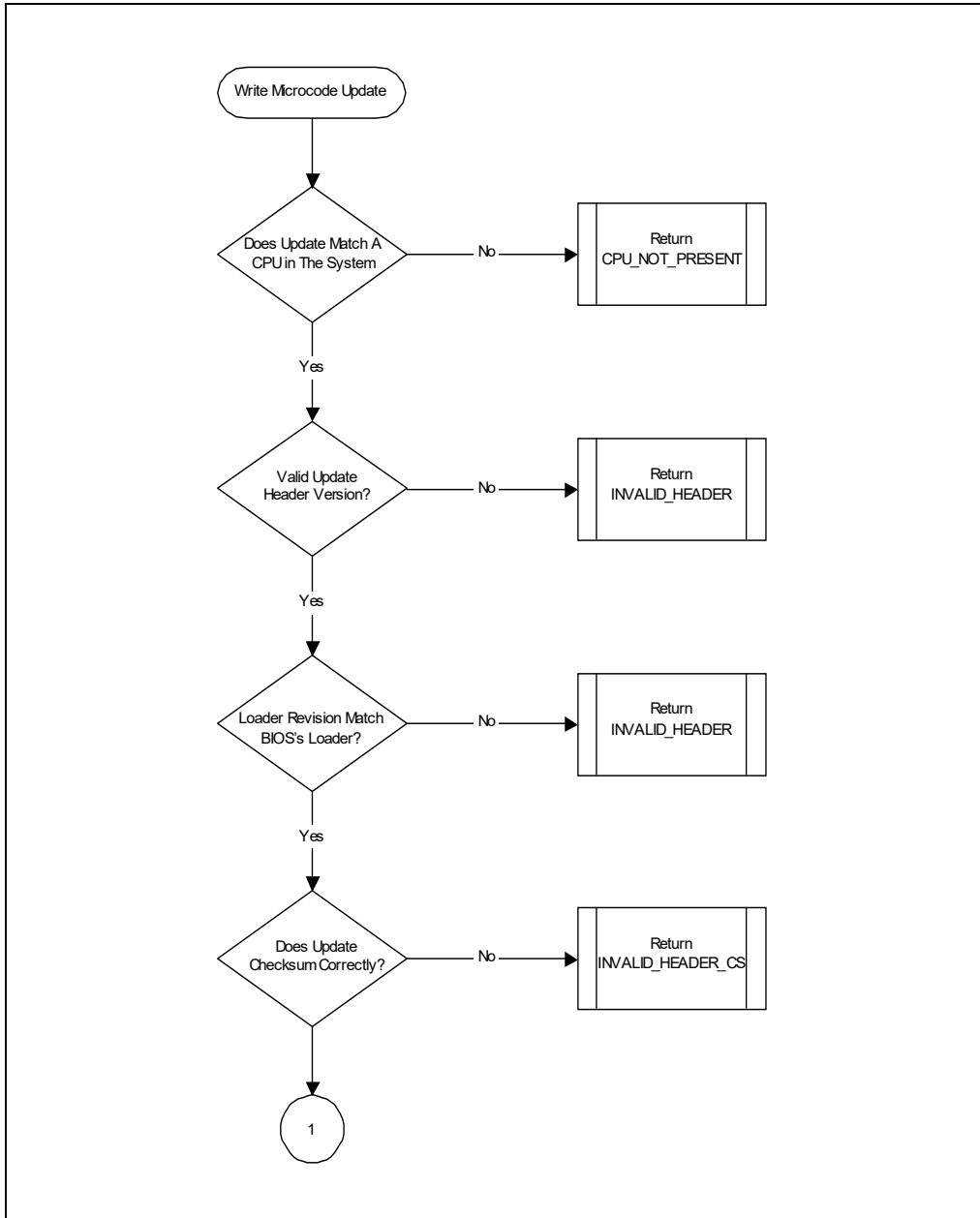


Figure 9-8. Microcode Update Write Operation Flow [1]

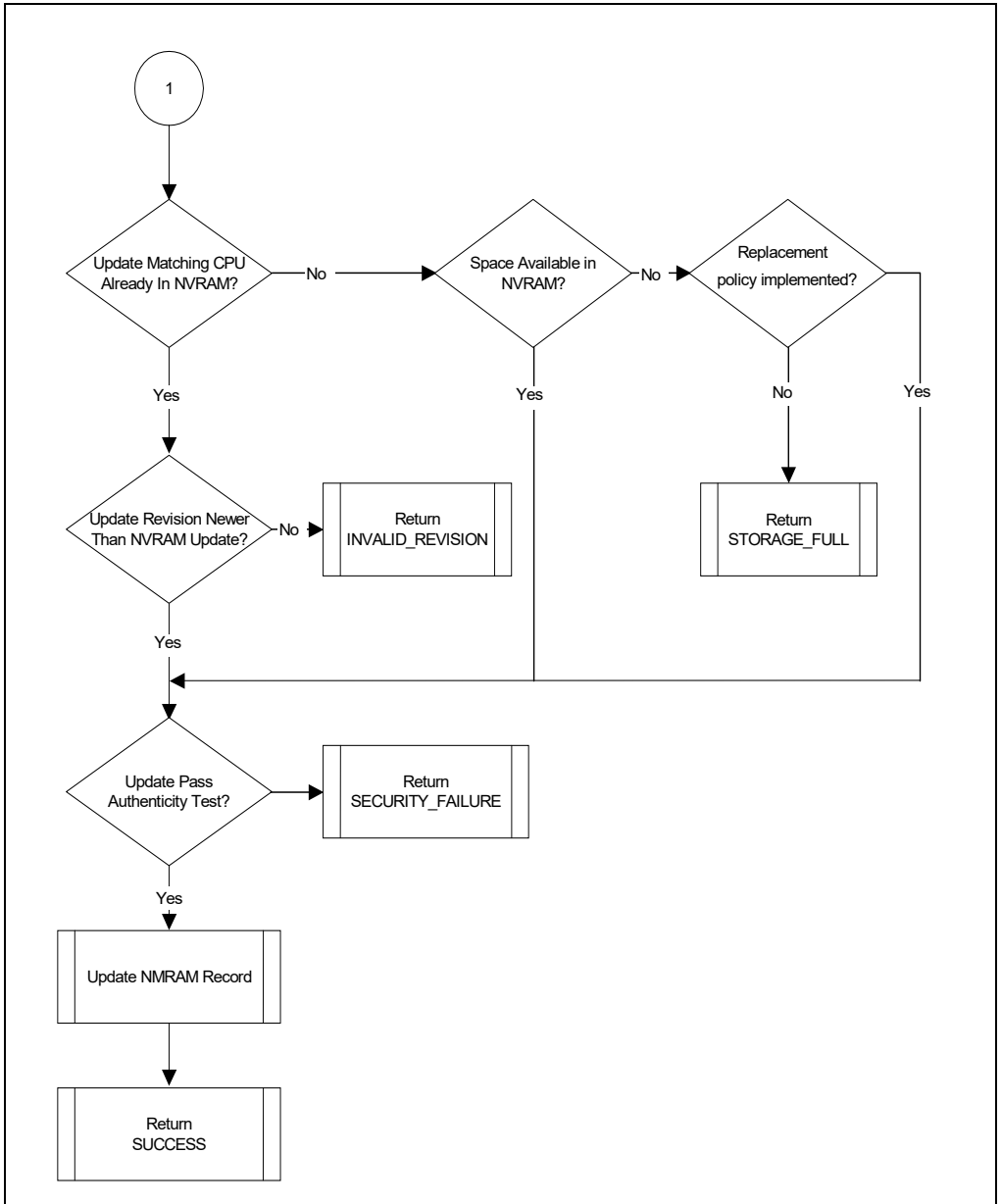


Figure 9-9. Microcode Update Write Operation Flow [2]

### 9.11.8.7 Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 9-16 lists the parameters and return codes for the function.



**Table 9-16. Parameters for the Control Update Sub-function**

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 KBytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 9-17 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

**Table 9-17. Mnemonic Values**

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ\_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

### 9.11.8.8 Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 9-18 lists the parameters and return codes.

**Table 9-18. Parameters for the Read Microcode Update Data Function**

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data

**Table 9-18. Parameters for the Read Microcode Update Data Function (Contd.)**

ECX	Scratch Pad1	Real Mode Segment address of 64 KBytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 KBytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 KBytes of RAM Block
SS:SP	Stack pointer	32 KBytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
<b>Output</b>		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
<b>Return Codes (see Table 9-19 for code definitions)</b>		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT\_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFH after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

### 9.11.8.9 Return Codes

After the call has been made, the return codes listed in Table 9-19 are available in the AH register.

Table 9-19. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented.
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks.  The specified block is not a header block and is not empty.



## 18. Updates to Chapter 15, Volume 3B

Change bars and green text show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes to this chapter:

Changed "master" to "main" and "slave" to "secondary".

Intel is committed to ensuring technology is fully inclusive. In the spirit of this effort, we are actively working to replace instances of non-inclusive terms (e.g., Master / Slave) with more accurate and descriptive inclusive terms

This chapter describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. See Chapter 6, “Interrupt 18—Machine-Check Exception (#MC),” for more information on machine-check exceptions. A brief description of the Pentium processor’s machine check capability is also given.

Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

### 15.1 MACHINE-CHECK ARCHITECTURE

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors implement a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected.

The processor signals the detection of an uncorrected machine-check error by generating a machine-check exception (#MC), which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

Starting with 45 nm Intel 64 processor on which CPUID reports DisplayFamily\_DisplayModel as 06H\_1AH (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*), the processor can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors, referred to as corrected machine-check error interrupt (CMCI). See Section 15.5 for detail.

Intel 64 processors supporting machine-check architecture and CMCI may also support an additional enhancement, namely, support for software recovery from certain uncorrected recoverable machine check errors. See Section 15.6 for detail.

### 15.2 COMPATIBILITY WITH PENTIUM PROCESSOR

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support and extend the machine-check exception mechanism introduced in the Pentium processor. The Pentium processor reports the following machine-check errors:

- data parity errors during read cycles
- unsuccessful completion of a bus cycle

The above errors are reported using the P5\_MC\_TYPE and P5\_MC\_ADDR MSRs (implementation specific for the Pentium processor). Use the RDMSR instruction to read these MSRs. See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for the addresses.

The machine-check error reporting mechanism that Pentium processors use is similar to that used in Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. When an error is detected, it is recorded in P5\_MC\_TYPE and P5\_MC\_ADDR; the processor then generates a machine-check exception (#MC).

See Section 15.3.3, “Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture,” and Section 15.10.2, “Pentium Processor Machine-Check Exception Handling,” for information on compatibility between machine-check code written to run on the Pentium processors and code written to run on P6 family processors.

## 15.3 MACHINE-CHECK MSRS

Machine check MSRs in the Pentium 4, Intel Atom, Intel Xeon, and P6 family processors consist of a set of global control and status registers and several error-reporting register banks. See Figure 15-1.

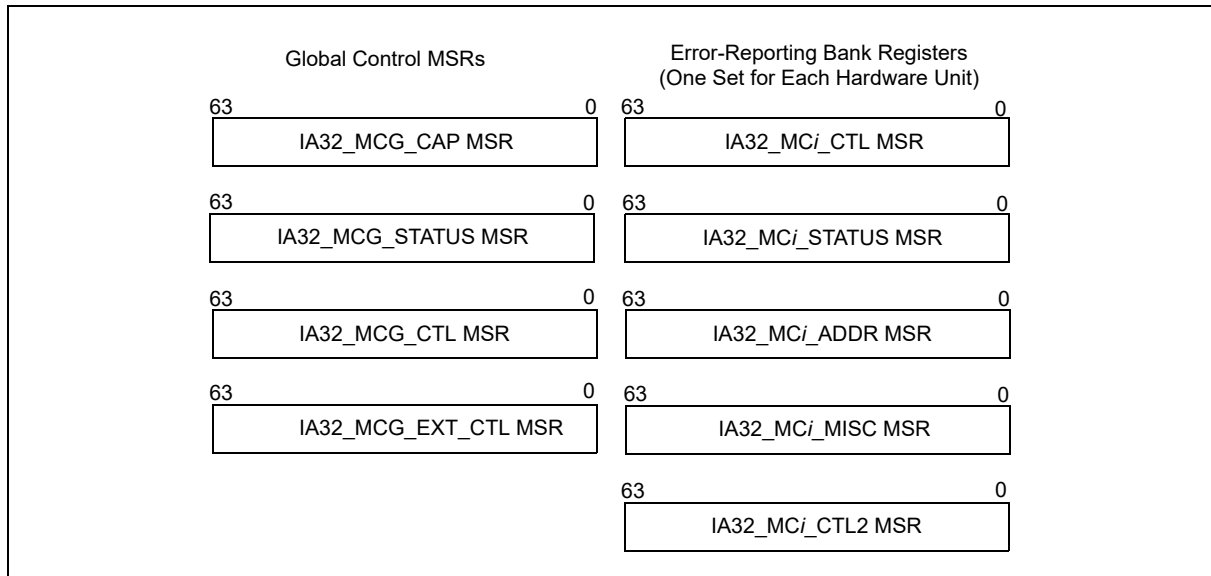


Figure 15-1. Machine-Check MSRs

Each error-reporting bank is associated with a specific hardware unit (or group of hardware units) in the processor. Use RDMSR and WRMSR to read and to write these registers.

### 15.3.1 Machine-Check Global Control MSRs

The machine-check global control MSRs include the IA32\_MCG\_CAP, IA32\_MCG\_STATUS, and optionally IA32\_MCG\_CTL and IA32\_MCG\_EXT\_CTL. See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for the addresses of these registers.

#### 15.3.1.1 IA32\_MCG\_CAP MSR

The IA32\_MCG\_CAP MSR is a read-only register that provides information about the machine-check architecture of the processor. Figure 15-2 shows the layout of the register.

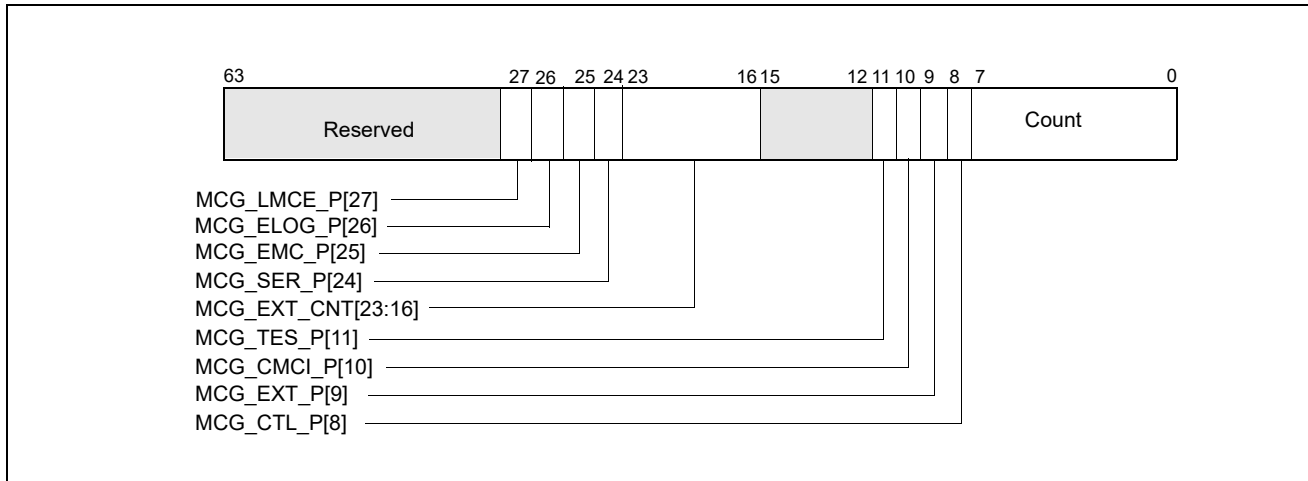


Figure 15-2. IA32\_MCG\_CAP Register

Where:

- **Count field, bits 7:0** — Indicates the number of hardware unit error-reporting banks available in a particular processor implementation.
- **MCG\_CTL\_P (control MSR present) flag, bit 8** — Indicates that the processor implements the IA32\_MCG\_CTL MSR when set; this register is absent when clear.
- **MCG\_EXT\_P (extended MSRs present) flag, bit 9** — Indicates that the processor implements the extended machine-check state registers found starting at MSR address 180H; these registers are absent when clear.
- **MCG\_CMCI\_P (Corrected MC error counting/signaling extension present) flag, bit 10** — Indicates (when set) that extended state and associated MSRs necessary to support the reporting of an interrupt on a corrected MC error event and/or count threshold of corrected MC errors, is present. When this bit is set, it does not imply this feature is supported across all banks. Software should check the availability of the necessary logic on a bank by bank basis when using this signaling capability (i.e. bit 30 settable in individual IA32\_MCi\_CTL2 register).
- **MCG\_TES\_P (threshold-based error status present) flag, bit 11** — Indicates (when set) that bits 56:53 of the IA32\_MCi\_STATUS MSR are part of the architectural space. Bits 56:55 are reserved, and bits 54:53 are used to report threshold-based error status. Note that when MCG\_TES\_P is not set, bits 56:53 of the IA32\_MCi\_STATUS MSR are model-specific.
- **MCG\_EXT\_CNT, bits 23:16** — Indicates the number of extended machine-check state registers present. This field is meaningful only when the MCG\_EXT\_P flag is set.
- **MCG\_SER\_P (software error recovery support present) flag, bit 24** — Indicates (when set) that the processor supports software error recovery (see Section 15.6), and IA32\_MCi\_STATUS MSR bits 56:55 are used to report the signaling of uncorrected recoverable errors and whether software must take recovery actions for uncorrected errors. Note that when MCG\_TES\_P is not set, bits 56:53 of the IA32\_MCi\_STATUS MSR are model-specific. If MCG\_TES\_P is set but MCG\_SER\_P is not set, bits 56:55 are reserved.
- **MCG EMC\_P (Enhanced Machine Check Capability) flag, bit 25** — Indicates (when set) that the processor supports enhanced machine check capabilities for firmware first signaling.
- **MCG\_ELOG\_P (extended error logging) flag, bit 26** — Indicates (when set) that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format “Generic Error Data Entry” that augments the data included in machine check bank registers.

For additional information about extended error logging interface, see

<https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/enhanced-mca-logging-xeon-paper.pdf>.



- **MCG\_LMCE\_P (local machine check exception) flag, bit 27** — Indicates (when set) that the following interfaces are present:
  - an extended state LMCE\_S (located in bit 3 of IA32\_MCG\_STATUS), and
  - the IA32\_MCG\_EXT\_CTL MSR, necessary to support Local Machine Check Exception (LMCE).

A non-zero MCG\_LMCE\_P indicates that, when LMCE is enabled as described in Section 15.3.1.5, some machine check errors may be delivered to only a single logical processor.

The effect of writing to the IA32\_MCG\_CAP MSR is undefined.

### 15.3.1.2 IA32\_MCG\_STATUS MSR

The IA32\_MCG\_STATUS MSR describes the current state of the processor after a machine-check exception has occurred (see Figure 15-3).

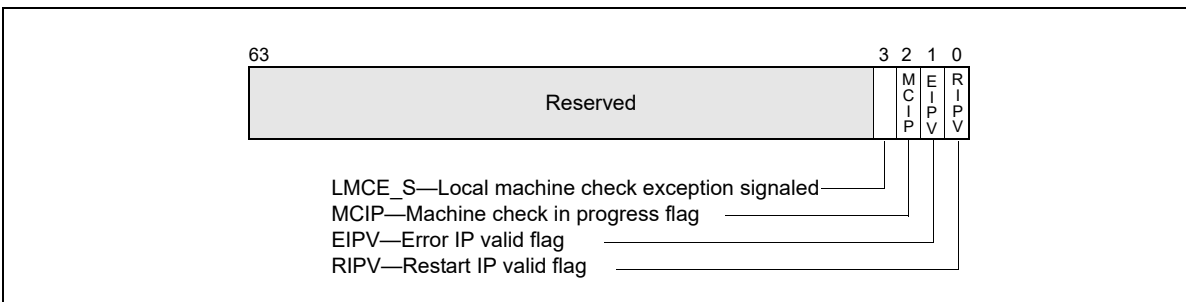


Figure 15-3. IA32\_MCG\_STATUS Register

Where:

- **RIPV (restart IP valid) flag, bit 0** — Indicates (when set) that program execution can be restarted reliably at the instruction pointed to by the instruction pointer pushed on the stack when the machine-check exception is generated. When clear, the program cannot be reliably restarted at the pushed instruction pointer.
- **EIPV (error IP valid) flag, bit 1** — Indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- **MCIP (machine check in progress) flag, bit 2** — Indicates (when set) that a machine-check exception was generated. Software can set or clear this flag. The occurrence of a second Machine-Check Event while MCIP is set will cause the processor to enter a shutdown state. For information on processor behavior in the shutdown state, please refer to the description in Chapter 6, "Interrupt and Exception Handling": "Interrupt 8—Double Fault Exception (#DF)".
- **LMCE\_S (local machine check exception signaled), bit 3** — Indicates (when set) that a local machine-check exception was generated. This indicates that the current machine-check event was delivered to only this logical processor.

Bits 63:04 in IA32\_MCG\_STATUS are reserved. An attempt to write to IA32\_MCG\_STATUS with any value other than 0 would result in #GP.

### 15.3.1.3 IA32\_MCG\_CTL MSR

The IA32\_MCG\_CTL MSR is present if the capability flag MCG\_CTL\_P is set in the IA32\_MCG\_CAP MSR.

IA32\_MCG\_CTL controls the reporting of machine-check exceptions. If present, writing 1s to this register enables machine-check features and writing all 0s disables machine-check features. All other values are undefined and/or implementation specific.

### 15.3.1.4 IA32\_MCG\_EXT\_CTL MSR

The IA32\_MCG\_EXT\_CTL MSR is present if the capability flag MCG\_LMCE\_P is set in the IA32\_MCG\_CAP MSR. IA32\_MCG\_EXT\_CTL.LMCE\_EN (bit 0) allows the processor to signal some MCEs to only a single logical processor in the system.

If MCG\_LMCE\_P is not set in IA32\_MCG\_CAP, or platform software has not enabled LMCE by setting IA32\_FEATURE\_CONTROL.LMCE\_ENABLED (bit 20), any attempt to write or read IA32\_MCG\_EXT\_CTL will result in #GP.

The IA32\_MCG\_EXT\_CTL MSR is cleared on RESET.

Figure 15-4 shows the layout of the IA32\_MCG\_EXT\_CTL register

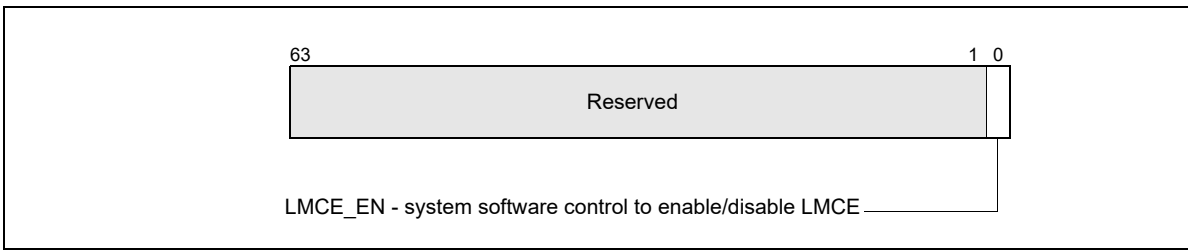


Figure 15-4. IA32\_MCG\_EXT\_CTL Register

where

- **LMCE\_EN (local machine check exception enable) flag, bit 0** - System software sets this to allow hardware to signal some MCEs to only a single logical processor. System software can set LMCE\_EN only if the platform software has configured IA32\_FEATURE\_CONTROL as described in Section 15.3.1.5.

### 15.3.1.5 Enabling Local Machine Check

The intended usage of LMCE requires proper configuration by both platform software and system software. Platform software can turn LMCE on by setting bit 20 (LMCE\_ENABLED) in IA32\_FEATURE\_CONTROL MSR (MSR address 3AH).

System software must ensure that both IA32\_FEATURE\_CONTROL.Lock (bit 0) and IA32\_FEATURE\_CONTROL.LMCE\_ENABLED (bit 20) are set before attempting to set IA32\_MCG\_EXT\_CTL.LMCE\_EN (bit 0). When system software has enabled LMCE, then hardware will determine if a particular error can be delivered only to a single logical processor. Software should make no assumptions about the type of error that hardware can choose to deliver as LMCE. The severity and override rules stay the same as described in Table 15-8 to determine the recovery actions.

## 15.3.2 Error-Reporting Register Banks

Each error-reporting register bank can contain the IA32\_MCi\_CTL, IA32\_MCi\_STATUS, IA32\_MCi\_ADDR, and IA32\_MCi\_MISC MSRs. The number of reporting banks is indicated by bits [7:0] of IA32\_MCG\_CAP MSR (address 0179H). The first error-reporting register (IA32\_MC0\_CTL) always starts at address 400H.

See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for addresses of the error-reporting registers in the Pentium 4, Intel Atom, and Intel Xeon processors; and for addresses of the error-reporting registers P6 family processors.

### 15.3.2.1 IA32\_MCi\_CTL MSRs

The IA32\_MCi\_CTL MSR controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EE<sub>j</sub>) represents a potential error. Setting an EE<sub>j</sub> flag enables signaling #MC of the associated error and clearing it disables signaling of the error. Error logging happens regardless of the setting

of these bits. The processor drops writes to bits that are not implemented. Figure 15-5 shows the bit fields of IA32\_MCi\_CTL.

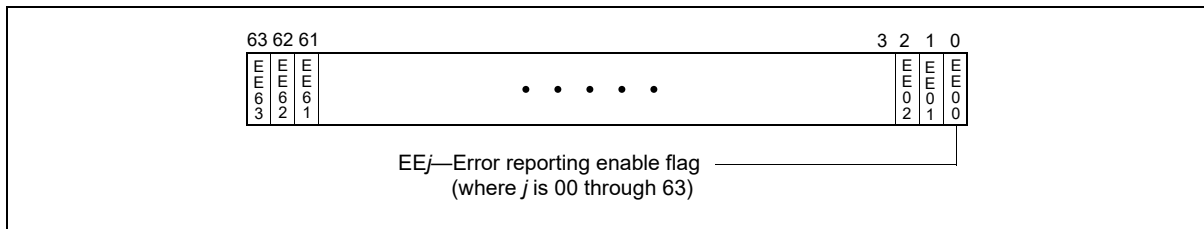


Figure 15-5. IA32\_MCi\_CTL Register

**NOTE**

For P6 family processors, processors based on Intel Core microarchitecture (excluding those on which on which CPUID reports DisplayFamily\_DisplayModel as 06H\_1AH and onward): the operating system or executive software must not modify the contents of the IA32\_MCO\_CTL MSR. This MSR is internally aliased to the EBL\_CR\_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the IA32\_MCO\_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the IA32\_MCi\_CTL MSR.

**15.3.2.2 IA32\_MCi\_STATUS MSRS**

Each IA32\_MCi\_STATUS MSR contains information related to a machine-check error if its VAL (valid) flag is set (see Figure 15-6). Software is responsible for clearing IA32\_MCi\_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.

**NOTE**

Figure 15-6 depicts the IA32\_MCi\_STATUS MSR when IA32\_MCG\_CAP[24] = 1, IA32\_MCG\_CAP[11] = 1 and IA32\_MCG\_CAP[10] = 1. When IA32\_MCG\_CAP[24] = 0 and IA32\_MCG\_CAP[11] = 1, bits 56:55 is reserved and bits 54:53 for threshold-based error reporting. When IA32\_MCG\_CAP[11] = 0, bits 56:53 are part of the “Other Information” field. The use of bits 54:53 for threshold-based error reporting began with Intel Core Duo processors, and is currently used for cache memory. See Section 15.4, “Enhanced Cache Error reporting,” for more information. When IA32\_MCG\_CAP[10] = 0, bits 52:38 are part of the “Other Information” field. The use of bits 52:38 for corrected MC error count is introduced with Intel 64 processor on which CPUID reports DisplayFamily\_DisplayModel as 06H\_1AH.

Where:

- **MCA (machine-check architecture) error code field, bits 15:0** — Specifies the machine-check architecture-defined error code for the machine-check error condition detected. The machine-check architecture-defined error codes are guaranteed to be the same for all IA-32 processors that implement the machine-check architecture. See Section 15.9, “Interpreting the MCA Error Codes,” and Chapter 16, “Interpreting Machine-Check Error Codes”, for information on machine-check error codes.
- **Model-specific error code field, bits 31:16** — Specifies the model-specific error code that uniquely identifies the machine-check error condition detected. The model-specific error codes may differ among IA-32 processors for the same machine-check error condition. See Chapter 16, “Interpreting Machine-Check Error Codes” for information on model-specific error codes.
- **Reserved, Error Status, and Other Information fields, bits 56:32** —
  - If IA32\_MCG\_CAP.MCG EMC\_P[bit 25] is 0, bits 37:32 contain “Other Information” that is implementation-specific and is not part of the machine-check architecture.
  - If IA32\_MCG\_CAP.MCG EMC\_P is 1, “Other Information” is in bits 36:32. If bit 37 is 0, system firmware has not changed the contents of IA32\_MCi\_STATUS. If bit 37 is 1, system firmware may have edited the contents of IA32\_MCi\_STATUS.

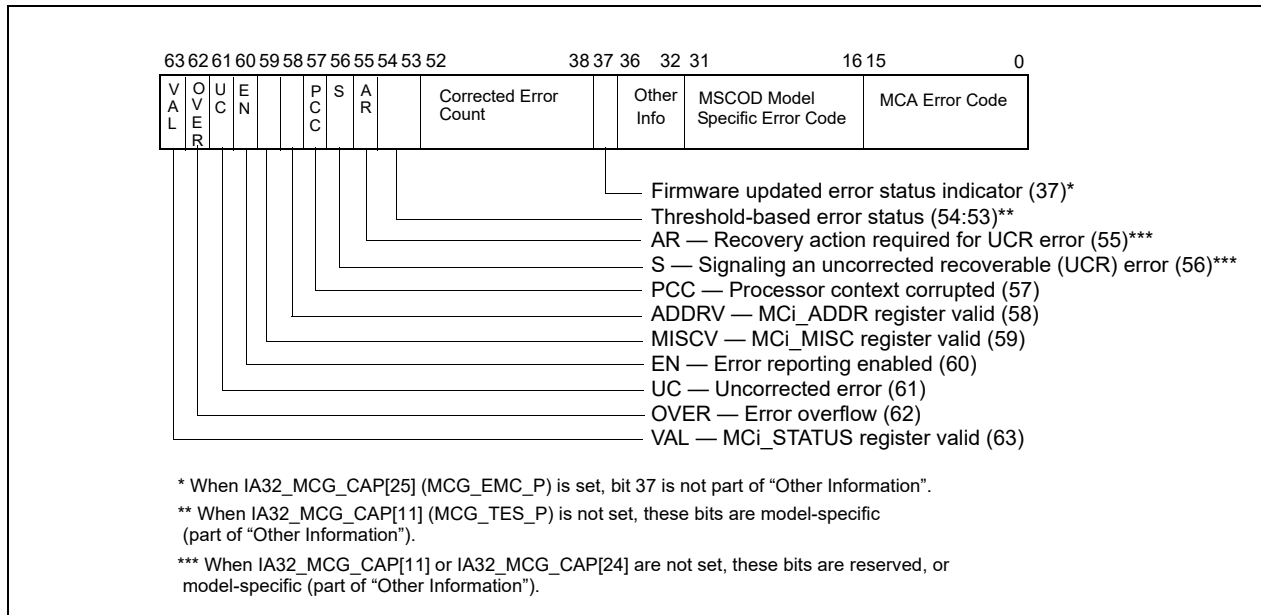


Figure 15-6. IA32\_MCI\_STATUS Register

- If IA32\_MCG\_CAP.MCG\_CMCI\_P[bit 10] is 0, bits 52:38 also contain "Other Information" (in the same sense as bits 37:32).
- If IA32\_MCG\_CAP[10] is 1, bits 52:38 are architectural (not model-specific). In this case, bits 52:38 reports the value of a 15 bit counter that increments each time a corrected error is observed by the MCA recording bank. This count value will continue to increment until cleared by software. The most significant bit, 52, is a sticky count overflow bit.
- If IA32\_MCG\_CAP[11] is 0, bits 56:53 also contain "Other Information" (in the same sense).
- If IA32\_MCG\_CAP[11] is 1, bits 56:53 are architectural (not model-specific). In this case, bits 56:53 have the following functionality:
  - If IA32\_MCG\_CAP[24] is 0, bits 56:55 are reserved.
  - If IA32\_MCG\_CAP[24] is 1, bits 56:55 are defined as follows:
    - S (Signaling) flag, bit 56 - Signals the reporting of UCR errors in this MC bank. See Section 15.6.2 for additional detail.
    - AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. See Section 15.6.2 for additional detail.
  - If the UC bit (Figure 15-6) is 1, bits 54:53 are undefined.
  - If the UC bit (Figure 15-6) is 0, bits 54:53 indicate the status of the hardware structure that reported the threshold-based error. See Table 15-1.

Table 15-1. Bits 54:53 in IA32\_MCI\_STATUS MSRs when IA32\_MCG\_CAP[11] = 1 and UC = 0

Bits 54:53	Meaning
00	<b>No tracking</b> - No hardware status tracking is provided for the structure reporting this event.
01	<b>Green</b> - Status tracking is provided for the structure posting the event; the current status is green (below threshold). For more information, see Section 15.4, "Enhanced Cache Error reporting".
10	<b>Yellow</b> - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). For more information, see Section 15.4, "Enhanced Cache Error reporting".
11	Reserved

- **PCC (processor context corrupt) flag, bit 57** — Indicates (when set) that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor’s state, and software may be able to restart. When system software supports recovery, consult Section 15.10.4, “Machine-Check Software Handler Guidelines for Error Recovery” for additional rules that apply.
- **ADDRV (IA32\_MCi\_ADDR register valid) flag, bit 58** — Indicates (when set) that the IA32\_MCi\_ADDR register contains the address where the error occurred (see Section 15.3.2.3, “IA32\_MCi\_ADDR MSRs”). When clear, this flag indicates that the IA32\_MCi\_ADDR register is either not implemented or does not contain the address where the error occurred. Do not read these registers if they are not implemented in the processor.
- **MISCV (IA32\_MCi\_MISC register valid) flag, bit 59** — Indicates (when set) that the IA32\_MCi\_MISC register contains additional information regarding the error. When clear, this flag indicates that the IA32\_MCi\_MISC register is either not implemented or does not contain additional information regarding the error. Do not read these registers if they are not implemented in the processor.
- **EN (error enabled) flag, bit 60** — Indicates (when set) that the error was enabled by the associated EEj bit of the IA32\_MCi\_CTL register.
- **UC (error uncorrected) flag, bit 61** — Indicates (when set) that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.
- **OVER (machine check overflow) flag, bit 62** — Indicates (when set) that a machine-check error occurred while the results of a previous error were still in the error-reporting register bank (that is, the VAL bit was already set in the IA32\_MCi\_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. In general, enabled errors are written over disabled errors, and uncorrected errors are written over corrected errors. Uncorrected errors are not written over previous valid uncorrected errors. When MCG\_CMCI\_P is set, corrected errors may not set the OVER flag. Software can rely on corrected error count in IA32\_MCi\_Status[52:38] to determine if any additional corrected errors may have occurred. For more information, see Section 15.3.2.2.1, “Overwrite Rules for Machine Check Overflow”.
- **VAL (IA32\_MCi\_STATUS register valid) flag, bit 63** — Indicates (when set) that the information within the IA32\_MCi\_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the IA32\_MCi\_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

15.3.2.2.1 Overwrite Rules for Machine Check Overflow

Table 15-2 shows the overwrite rules for how to treat a second event if the cache has already posted an event to the MC bank – that is, what to do if the valid bit for an MC bank already is set to 1. When more than one structure posts events in a given bank, these rules specify whether a new event will overwrite a previous posting or not. These rules define a priority for uncorrected (highest priority), yellow, and green/unmonitored (lowest priority) status.

In Table 15-2, the values in the two left-most columns are IA32\_MCi\_STATUS[54:53].

Table 15-2. Overwrite Rules for Enabled Errors

First Event	Second Event	UC bit	Color	MCA Info
00/green	00/green	0	00/green	either
00/green	yellow	0	yellow	second error
yellow	00/green	0	yellow	first error
yellow	yellow	0	yellow	either
00/green/yellow	UC	1	undefined	second
UC	00/green/yellow	1	undefined	first

If a second event overwrites a previously posted event, the information (as guarded by individual valid bits) in the MCI bank is entirely from the second event. Similarly, if a first event is retained, all of the information previously posted for that event is retained. In general, when the logged error or the recent error is a corrected error, the OVER bit (MCI\_Status[62]) may be set to indicate an overflow. When MCG\_CMCI\_P is set in IA32\_MCG\_CAP, system software should consult IA32\_MCi\_STATUS[52:38] to determine if additional corrected errors may have

occurred. Software may re-read IA32\_MCi\_STATUS, IA32\_MCi\_ADDR and IA32\_MCi\_MISC appropriately to ensure data collected represent the last error logged.

After software polls a posting and clears the register, the valid bit is no longer set and therefore the meaning of the rest of the bits, including the yellow/green/00 status field in bits 54:53, is undefined. The yellow/green indication will only be posted for events associated with monitored structures – otherwise the unmonitored (00) code will be posted in IA32\_MCi\_STATUS[54:53].

### 15.3.2.3 IA32\_MCi\_ADDR MSRs

The IA32\_MCi\_ADDR MSR contains the address of the code or data memory location that produced the machine-check error if the ADDR\_V flag in the IA32\_MCi\_STATUS register is set (see Section 15-7, “IA32\_MCi\_ADDR MSR”). The IA32\_MCi\_ADDR register is either not implemented or contains no address if the ADDR\_V flag in the IA32\_MCi\_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception.

The address returned is an offset into a segment, linear address, or physical address. This depends on the error encountered. When these registers are implemented, these registers can be cleared by explicitly writing 0s to these registers. Writing 1s to these registers will cause a general-protection exception. See Figure 15-7.

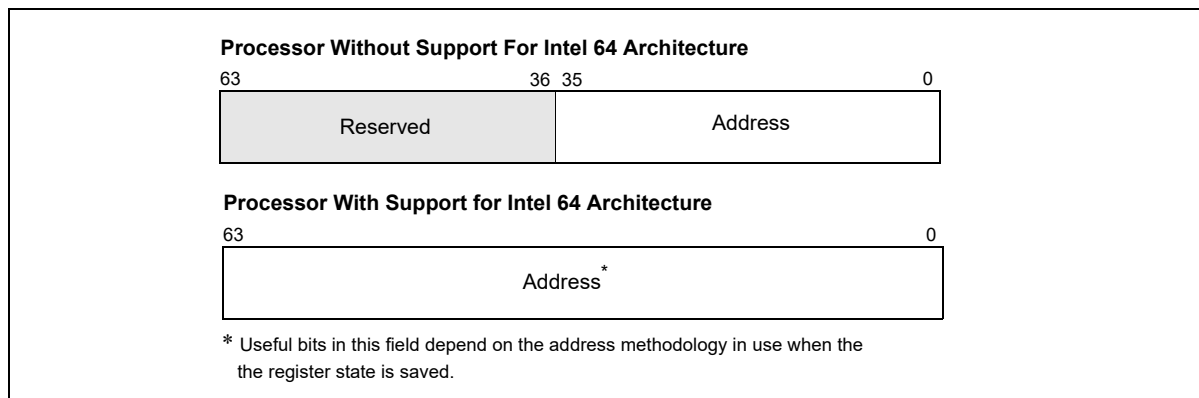


Figure 15-7. IA32\_MCi\_ADDR MSR

### 15.3.2.4 IA32\_MCi\_MISC MSRs

The IA32\_MCi\_MISC MSR contains additional information describing the machine-check error if the MISC\_V flag in the IA32\_MCi\_STATUS register is set. The IA32\_MCi\_MISC\_MSR is either not implemented or does not contain additional information if the MISC\_V flag in the IA32\_MCi\_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception. When implemented in a processor, these registers can be cleared by explicitly writing all 0s to them; writing 1s to them causes a general-protection exception to be generated. This register is not implemented in any of the error-reporting register banks for the P6 or Intel Atom family processors.

If both MISC\_V and IA32\_MCG\_CAP[24] are set, the IA32\_MCi\_MISC\_MSR is defined according to Figure 15-8 to support software recovery of uncorrected errors (see Section 15.6).

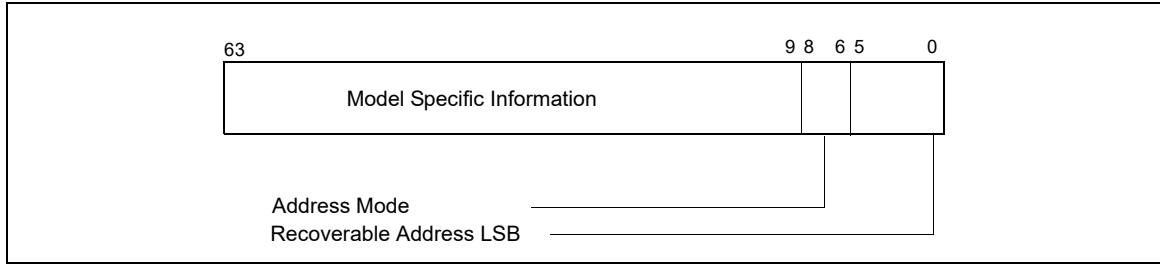


Figure 15-8. UCR Support in IA32\_MCI\_MISC Register

- Recoverable Address LSB (bits 5:0): The lowest valid recoverable address bit. Indicates the position of the least significant bit (LSB) of the recoverable error address. For example, if the processor logs bits [43:9] of the address, the LSB sub-field in IA32\_MCI\_MISC is 01001b (9 decimal). For this example, bits [8:0] of the recoverable error address in IA32\_MCI\_ADDR should be ignored.
- Address Mode (bits 8:6): Address mode for the address logged in IA32\_MCI\_ADDR. The supported address modes are given in Table 15-3.

Table 15-3. Address Mode in IA32\_MCI\_MISC[8:6]

IA32_MCI_MISC[8:6] Encoding	Definition
000	Segment Offset
001	Linear Address
010	Physical Address
011	Memory Address
100 to 110	Reserved
111	Generic

- Model Specific Information (bits 63:9): Not architecturally defined.

### 15.3.2.4.2 IOMCA

Logging and Signaling of errors from PCI Express domain is governed by PCI Express Advanced Error Reporting (AER) architecture. PCI Express architecture divides errors in two categories: Uncorrectable errors and Correctable errors. Uncorrectable errors can further be classified as Fatal or Non-Fatal. Uncorrected IO errors are signaled to the system software either as AER Message Signaled Interrupt (MSI) or via platform specific mechanisms such as NMI. Generally, the signaling mechanism is controlled by BIOS and/or platform firmware. Certain processors support an error handling mode, called IOMCA mode, where Uncorrected PCI Express errors are signaled in the form of machine check exception and logged in machine check banks.

When a processor is in this mode, Uncorrected PCI Express errors are logged in the MCACOD field of the IA32\_MCI\_STATUS register as Generic I/O error. The corresponding MCA error code is defined in Table 15-8. IA32\_MCI\_Status [15:0] Simple Error Code Encoding. Machine check logging complements and does not replace AER logging that occurs inside the PCI Express hierarchy. The PCI Express Root Complex and Endpoints continue to log the error in accordance with PCI Express AER mechanism. In IOMCA mode, MCI\_MISC register in the bank that logged IOMCA can optionally contain information that link the Machine Check logs with the AER logs or proprietary logs. In such a scenario, the machine check handler can utilize the contents of MCI\_MISC to locate the next level of error logs corresponding to the same error. Specifically, if MCI\_Status.MISCV is 1 and MCACOD is 0x0E0B, MCI\_MISC contains the PCI Express address of the Root Complex device containing the AER Logs. Software can consult the header type and class code registers in the Root Complex device's PCIe Configuration space to determine what type of device it is. This Root Complex device can either be a PCI Express Root Port, PCI Express Root Complex Event Collector or a proprietary device.



Errors that originate from PCI Express or Legacy Endpoints are logged in the corresponding Root Port in addition to the generating device. If `MISCV=1` and `MCi_MISC` contains the address of the Root Port or a Root Complex Event collector, software can parse the AER logs to learn more about the error.

If `MISCV=1` and `MCi_MISC` points to a device that is neither a Root Complex Event Collector nor a Root Port, software must consult the Vendor ID/Device ID and use device specific knowledge to locate and interpret the error log registers. In some cases, the Root Complex device configuration space may not be accessible to the software and both the Vendor and Device ID read as `0xFFFF`.

- The format of `MCi_MISC` for IOMCA errors is shown in Table 15-4.

**Table 15-4. Address Mode in IA32\_MCi\_MISC[8:6]**

63:40	39:32	31:16	15:9	8:6	5:0
RSVD	PCI Express Segment number	PCI Express Requestor ID	RSVD	ADDR MODE <sup>1</sup>	RECOV ADDR LSB <sup>1</sup>

**NOTES:**

1. Not Applicable if `ADDRV=0`.

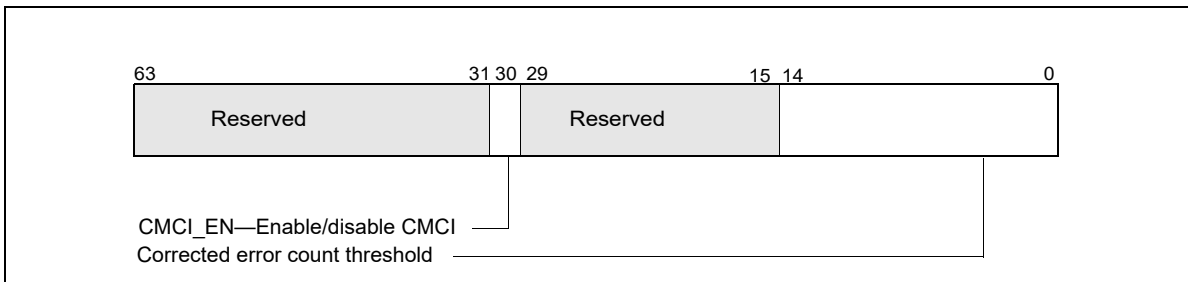
Refer to PCI Express Specification 3.0 for definition of PCI Express Requestor ID and AER architecture. Refer to PCI Firmware Specification 3.0 for an explanation of PCI Express Segment number and how software can access configuration space of a PCI Express device given the segment number and Requestor ID.

### 15.3.2.5 IA32\_MCi\_CTL2 MSRs

The `IA32_MCi_CTL2` MSR provides the programming interface to use corrected MC error signaling capability that is indicated by `IA32_MCG_CAP[10] = 1`. Software must check for the presence of `IA32_MCi_CTL2` on a per-bank basis.

When `IA32_MCG_CAP[10] = 1`, the `IA32_MCi_CTL2` MSR for each bank exists, i.e. reads and writes to these MSR are supported. However, signaling interface for corrected MC errors may not be supported in all banks.

The layout of `IA32_MCi_CTL2` is shown in Figure 15-9:



**Figure 15-9. IA32\_MCi\_CTL2 Register**

- **Corrected error count threshold, bits 14:0** — Software must initialize this field. The value is compared with the corrected error count field in `IA32_MCi_STATUS`, bits 38 through 52. An overflow event is signaled to the CMCI LVT entry (see Table 10-1) in the APIC when the count value equals the threshold value. The new LVT entry in the APIC is at `02F0H` offset from the `APIC_BASE`. If CMCI interface is not supported for a particular bank (but `IA32_MCG_CAP[10] = 1`), this field will always read 0.
- **CMCI\_EN (Corrected error interrupt enable/disable/indicator), bits 30** — Software sets this bit to enable the generation of corrected machine-check error interrupt (CMCI). If CMCI interface is not supported for a particular bank (but `IA32_MCG_CAP[10] = 1`), this bit is writeable but will always return 0 for that bank. This bit also indicates CMCI is supported or not supported in the corresponding bank. See Section 15.5 for details of software detection of CMCI facility.



Some microarchitectural sub-systems that are the source of corrected MC errors may be shared by more than one logical processors. Consequently, the facilities for reporting MC errors and controlling mechanisms may be shared by more than one logical processors. For example, the IA32\_MCi\_CTL2 MSR is shared between logical processors sharing a processor core. Software is responsible to program IA32\_MCi\_CTL2 MSR in a consistent manner with CMCi delivery and usage.

After processor reset, IA32\_MCi\_CTL2 MSRs are zero'ed.

### 15.3.2.6 IA32\_MCG Extended Machine Check State MSRs

The Pentium 4 and Intel Xeon processors implement a variable number of extended machine-check state MSRs. The MCG\_EXT\_P flag in the IA32\_MCG\_CAP MSR indicates the presence of these extended registers, and the MCG\_EXT\_CNT field indicates the number of these registers actually implemented. See Section 15.3.1.1, "IA32\_MCG\_CAP MSR." Also see Table 15-5.

**Table 15-5. Extended Machine Check State MSRs in Processors Without Support for Intel 64 Architecture**

MSR	Address	Description
IA32_MCG_EAX	180H	Contains state of the EAX register at the time of the machine-check error.
IA32_MCG_EBX	181H	Contains state of the EBX register at the time of the machine-check error.
IA32_MCG_ECX	182H	Contains state of the ECX register at the time of the machine-check error.
IA32_MCG_EDX	183H	Contains state of the EDX register at the time of the machine-check error.
IA32_MCG_ESI	184H	Contains state of the ESI register at the time of the machine-check error.
IA32_MCG_EDI	185H	Contains state of the EDI register at the time of the machine-check error.
IA32_MCG_EBP	186H	Contains state of the EBP register at the time of the machine-check error.
IA32_MCG_ESP	187H	Contains state of the ESP register at the time of the machine-check error.
IA32_MCG_EFLAGS	188H	Contains state of the EFLAGS register at the time of the machine-check error.
IA32_MCG_EIP	189H	Contains state of the EIP register at the time of the machine-check error.
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.

In processors with support for Intel 64 architecture, 64-bit machine check state MSRs are aliased to the legacy MSRs. In addition, there may be registers beyond IA32\_MCG\_MISC. These may include up to five reserved MSRs (IA32\_MCG\_RESERVED[1:5]) and save-state MSRs for registers introduced in 64-bit mode. See Table 15-6.

**Table 15-6. Extended Machine Check State MSRs In Processors With Support For Intel 64 Architecture**

MSR	Address	Description
IA32_MCG_RAX	180H	Contains state of the RAX register at the time of the machine-check error.
IA32_MCG_RBX	181H	Contains state of the RBX register at the time of the machine-check error.
IA32_MCG_RCX	182H	Contains state of the RCX register at the time of the machine-check error.
IA32_MCG_RDX	183H	Contains state of the RDX register at the time of the machine-check error.
IA32_MCG_RSI	184H	Contains state of the RSI register at the time of the machine-check error.
IA32_MCG_RDI	185H	Contains state of the RDI register at the time of the machine-check error.
IA32_MCG_RBP	186H	Contains state of the RBP register at the time of the machine-check error.
IA32_MCG_RSP	187H	Contains state of the RSP register at the time of the machine-check error.
IA32_MCG_RFLAGS	188H	Contains state of the RFLAGS register at the time of the machine-check error.
IA32_MCG_RIP	189H	Contains state of the RIP register at the time of the machine-check error.

**Table 15-6. Extended Machine Check State MSRs  
In Processors With Support For Intel 64 Architecture (Contd.)**

MSR	Address	Description
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.
IA32_MCG_RSERVED[1:5]	18BH-18FH	These registers, if present, are reserved.
IA32_MCG_R8	190H	Contains state of the R8 register at the time of the machine-check error.
IA32_MCG_R9	191H	Contains state of the R9 register at the time of the machine-check error.
IA32_MCG_R10	192H	Contains state of the R10 register at the time of the machine-check error.
IA32_MCG_R11	193H	Contains state of the R11 register at the time of the machine-check error.
IA32_MCG_R12	194H	Contains state of the R12 register at the time of the machine-check error.
IA32_MCG_R13	195H	Contains state of the R13 register at the time of the machine-check error.
IA32_MCG_R14	196H	Contains state of the R14 register at the time of the machine-check error.
IA32_MCG_R15	197H	Contains state of the R15 register at the time of the machine-check error.

When a machine-check error is detected on a Pentium 4 or Intel Xeon processor, the processor saves the state of the general-purpose registers, the R/EFLAGS register, and the R/EIP in these extended machine-check state MSRs. This information can be used by a debugger to analyze the error.

These registers are read/write to zero registers. This means software can read them; but if software writes to them, only all zeros is allowed. If software attempts to write a non-zero value into one of these registers, a general-protection (#GP) exception is generated. These registers are cleared on a hardware reset (power-up or RESET), but maintain their contents following a soft reset (INIT reset).

### 15.3.3 Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture

The Pentium processor reports machine-check errors using two registers: P5\_MC\_TYPE and P5\_MC\_ADDR. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors map these registers to the IA32\_MC<sub>i</sub>\_STATUS and IA32\_MC<sub>i</sub>\_ADDR in the error-reporting register bank. This bank reports on the same type of external bus errors reported in P5\_MC\_TYPE and P5\_MC\_ADDR.

The information in these registers can then be accessed in two ways:

- By reading the IA32\_MC<sub>i</sub>\_STATUS and IA32\_MC<sub>i</sub>\_ADDR registers as part of a general machine-check exception handler written for Pentium 4, Intel Atom and P6 family processors.
- By reading the P5\_MC\_TYPE and P5\_MC\_ADDR registers using the RDMSR instruction.

The second capability permits a machine-check exception handler written to run on a Pentium processor to be run on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor. There is a limitation in that information returned by the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors is encoded differently than information returned by the Pentium processor. To run a Pentium processor machine-check exception handler on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor; the handler must be written to interpret P5\_MC\_TYPE encodings correctly.

## 15.4 ENHANCED CACHE ERROR REPORTING

Starting with Intel Core Duo processors, cache error reporting was enhanced. In earlier Intel processors, cache status was based on the number of correction events that occurred in a cache. In the new paradigm, called "threshold-based error status", cache status is based on the number of lines (ECC blocks) in a cache that incur repeated corrections. The threshold is chosen by Intel, based on various factors. If a processor supports threshold-based error status, it sets IA32\_MCG\_CAP[11] (MCG\_TES\_P) to 1; if not, to 0.

A processor that supports enhanced cache error reporting contains hardware that tracks the operating status of certain caches and provides an indicator of their “health”. The hardware reports a “green” status when the number of lines that incur repeated corrections is at or below a pre-defined threshold, and a “yellow” status when the number of affected lines exceeds the threshold. Yellow status means that the cache reporting the event is operating correctly, but you should schedule the system for servicing within a few weeks.

Intel recommends that you rely on this mechanism for structures supported by threshold-base error reporting.

The CPU/system/platform response to a yellow event should be less severe than its response to an uncorrected error. An uncorrected error means that a serious error has actually occurred, whereas the yellow condition is a warning that the number of affected lines has exceeded the threshold but is not, in itself, a serious event: the error was corrected and system state was not compromised.

The green/yellow status indicator is not a foolproof early warning for an uncorrected error resulting from the failure of two bits in the same ECC block. Such a failure can occur and cause an uncorrected error before the yellow threshold is reached. However, the chance of an uncorrected error increases as the number of affected lines increases.

## 15.5 CORRECTED MACHINE CHECK ERROR INTERRUPT

Corrected machine-check error interrupt (CMCI) is an architectural enhancement to the machine-check architecture. It provides capabilities beyond those of threshold-based error reporting (Section 15.4). With threshold-based error reporting, software is limited to use periodic polling to query the status of hardware corrected MC errors. CMCI provides a signaling mechanism to deliver a local interrupt based on threshold values that software can program using the IA32\_MCi\_CTL2 MSR.

CMCI is disabled by default. System software is required to enable CMCI for each IA32\_MCi bank that support the reporting of hardware corrected errors if IA32\_MCG\_CAP[10] = 1.

System software use IA32\_MCi\_CTL2 MSR to enable/disable the CMCI capability for each bank and program threshold values into IA32\_MCi\_CTL2 MSR. CMCI is not affected by the CR4.MCE bit, and it is not affected by the IA32\_MCi\_CTL MSR.

To detect the existence of thresholding for a given bank, software writes only bits 14:0 with the threshold value. If the bits persist, then thresholding is available (and CMCI is available). If the bits are all 0's, then no thresholding exists. To detect that CMCI signaling exists, software writes a 1 to bit 30 of the MCI\_CTL2 register. Upon subsequent read, if bit 30 = 0, no CMCI is available for this bank and no corrected or UCNA errors will be reported on this bank. If bit 30 = 1, then CMCI is available and enabled.

### 15.5.1 CMCI Local APIC Interface

The operation of CMCI is depicted in Figure 15-10.

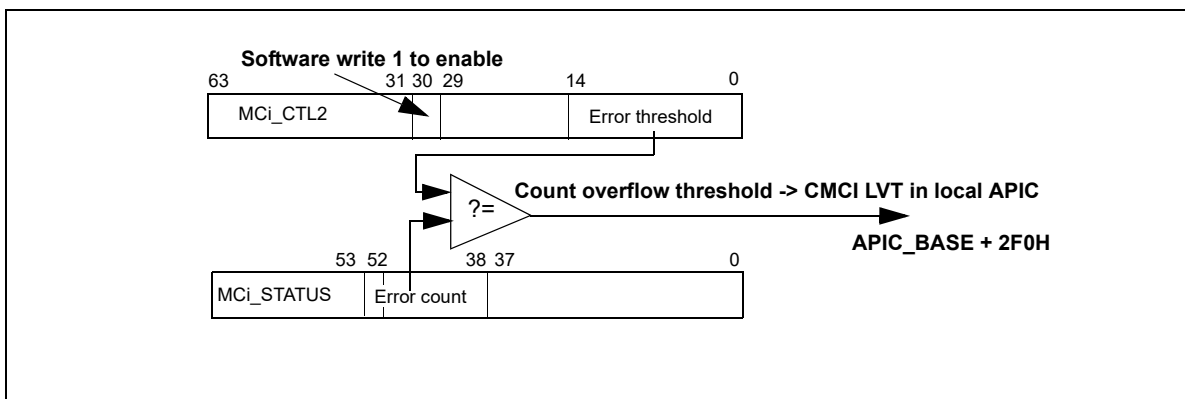


Figure 15-10. CMCI Behavior

CMCI interrupt delivery is configured by writing to the LVT CMCI register entry in the local APIC register space at default address of `APIC_BASE + 2F0H`. A CMCI interrupt can be delivered to more than one logical processors if multiple logical processors are affected by the associated MC errors. For example, if a corrected bit error in a cache shared by two logical processors caused a CMCI, the interrupt will be delivered to both logical processors sharing that microarchitectural sub-system. Similarly, package level errors may cause CMCI to be delivered to all logical processors within the package. However, system level errors will not be handled by CMCI.

See Section 10.5.1, “Local Vector Table” for details regarding the LVT CMCI register.

## 15.5.2 System Software Recommendation for Managing CMCI and Machine Check Resources

System software must enable and manage CMCI, set up interrupt handlers to service CMCI interrupts delivered to affected logical processors, program CMCI LVT entry, and query machine check banks that are shared by more than one logical processors.

This section describes techniques system software can implement to manage CMCI initialization, service CMCI interrupts in an efficient manner to minimize contentions to access shared MSR resources.

### 15.5.2.1 CMCI Initialization

Although a CMCI interrupt may be delivered to more than one logical processors depending on the nature of the corrected MC error, only one instance of the interrupt service routine needs to perform the necessary service and make queries to the machine-check banks. The following steps describes a technique that limits the amount of work the system has to do in response to a CMCI.

- To provide maximum flexibility, system software should define per-thread data structure for each logical processor to allow equal-opportunity and efficient response to interrupt delivery. Specifically, the per-thread data structure should include a set of per-bank fields to track which machine check bank it needs to access in response to a delivered CMCI interrupt. The number of banks that needs to be tracked is determined by `IA32_MCG_CAP[7:0]`.
- Initialization of per-thread data structure. The initialization of per-thread data structure must be done serially on each logical processor in the system. The sequencing order to start the per-thread initialization between different logical processor is arbitrary. But it must observe the following specific detail to satisfy the shared nature of specific MSR resources:
  - a. Each thread initializes its data structure to indicate that it does not own any MC bank registers.
  - b. Each thread examines `IA32_MCi_CTL2[30]` indicator for each bank to determine if another thread has already claimed ownership of that bank.
    - If `IA32_MCi_CTL2[30]` had been set by another thread. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
    - If `IA32_MCi_CTL2[30] = 0`, proceed to step c.
  - c. Check whether writing a 1 into `IA32_MCi_CTL2[30]` can return with 1 on a subsequent read to determine this bank can support CMCI.
    - If `IA32_MCi_CTL2[30] = 0`, this bank does not support CMCI. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
    - If `IA32_MCi_CTL2[30] = 1`, modify the per-thread data structure to indicate this thread claims ownership to the MC bank; proceed to initialize the error threshold count (bits 15:0) of that bank as described in Chapter 15, “CMCI Threshold Management”. Then proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
- After the thread has examined all of the machine check banks, it sees if it owns any MC banks to service CMCI. If any bank has been claimed by this thread:
  - Ensure that the CMCI interrupt handler has been set up as described in Chapter 15, “CMCI Interrupt Handler”.
  - Initialize the CMCI LVT entry, as described in Section 15.5.1, “CMCI Local APIC Interface”.

- Log and clear all of IA32\_MCi\_Status registers for the banks that this thread owns. This will allow new errors to be logged.

### 15.5.2.2 CMCI Threshold Management

The Corrected MC error threshold field, IA32\_MCi\_CTL2[14:0], is architecturally defined. Specifically, all these bits are writable by software, but different processor implementations may choose to implement less than 15 bits as threshold for the overflow comparison with IA32\_MCi\_STATUS[52:38]. The following describes techniques that software can manage CMCI threshold to be compatible with changes in implementation characteristics:

- Software can set the initial threshold value to 1 by writing 1 to IA32\_MCi\_CTL2[14:0]. This will cause overflow condition on every corrected MC error and generates a CMCI interrupt.
- To increase the threshold and reduce the frequency of CMCI servicing:
  - a. Find the maximum threshold value a given processor implementation supports. The steps are:
    - Write 7FFFH to IA32\_MCi\_CTL2[14:0],
    - Read back IA32\_MCi\_CTL2[14:0]; these 15 bits (14:0) contain the maximum threshold supported by the processor.
  - b. Increase the threshold to a value below the maximum value discovered using step a.

### 15.5.2.3 CMCI Interrupt Handler

The following describes techniques system software may consider to implement a CMCI service routine:

- The service routine examines its private per-thread data structure to check which set of MC banks it has ownership. If the thread does not have ownership of a given MC bank, proceed to the next MC bank. Ownership is determined at initialization time which is described in Section 15.5.2.1.

If the thread had claimed ownership to an MC bank, this technique will allow each logical processors to handle corrected MC errors independently and requires no synchronization to access shared MSR resources. Consult Example 15-5 for guidelines on logging when processing CMCI.

## 15.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS

Recovery of uncorrected recoverable machine check errors is an enhancement in machine-check architecture. The first processor that supports this feature is 45 nm Intel 64 processor on which CPUID reports DisplayFamily\_DisplayModel as 06H\_2EH (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). This allow system software to perform recovery action on certain class of uncorrected errors and continue execution.

### 15.6.1 Detection of Software Error Recovery Support

Software must use bit 24 of IA32\_MCG\_CAP (MCG\_SER\_P) to detect the presence of software error recovery support (see Figure 15-2). When IA32\_MCG\_CAP[24] is set, this indicates that the processor supports software error recovery. When this bit is clear, this indicates that there is no support for error recovery from the processor and the primary responsibility of the machine check handler is logging the machine check error information and shutting down the system.

The new class of architectural MCA errors from which system software can attempt recovery is called Uncorrected Recoverable (UCR) Errors. UCR errors are uncorrected errors that have been detected and signaled but have not corrupted the processor context. For certain UCR errors, this means that once system software has performed a certain recovery action, it is possible to continue execution on this processor. UCR error reporting provides an error containment mechanism for data poisoning. The machine check handler will use the error log information from the error reporting registers to analyze and implement specific error recovery actions for UCR errors.

## 15.6.2 UCR Error Reporting and Logging

IA32\_MCi\_STATUS MSR is used for reporting UCR errors and existing corrected or uncorrected errors. The definitions of IA32\_MCi\_STATUS, including bit fields to identify UCR errors, is shown in Figure 15-6. UCR errors can be signaled through either the corrected machine check interrupt (CMCI) or machine check exception (MCE) path depending on the type of the UCR error.

When IA32\_MCG\_CAP[24] is set, a UCR error is indicated by the following bit settings in the IA32\_MCi\_STATUS register:

- Valid (bit 63) = 1
- UC (bit 61) = 1
- PCC (bit 57) = 0

Additional information from the IA32\_MCi\_MISC and the IA32\_MCi\_ADDR registers for the UCR error are available when the ADDR\_V and the MISC\_V flags in the IA32\_MCi\_STATUS register are set (see Section 15.3.2.4). The MCA error code field of the IA32\_MCi\_STATUS register indicates the type of UCR error. System software can interpret the MCA error code field to analyze and identify the necessary recovery action for the given UCR error.

In addition, the IA32\_MCi\_STATUS register bit fields, bits 56:55, are defined (see Figure 15-6) to provide additional information to help system software to properly identify the necessary recovery action for the UCR error:

- S (Signaling) flag, bit 56 - Indicates (when set) that a machine check exception was generated for the UCR error reported in this MC bank and system software needs to check the AR flag and the MCA error code fields in the IA32\_MCi\_STATUS register to identify the necessary recovery action for this error. When the S flag in the IA32\_MCi\_STATUS register is clear, this UCR error was not signaled via a machine check exception and instead was reported as a corrected machine check (CMC). System software is not required to take any recovery action when the S flag in the IA32\_MCi\_STATUS register is clear.
- AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. This recovery action must be completed successfully before any additional work is scheduled for this processor. When the RIP\_V flag in the IA32\_MCG\_STATUS is clear, an alternative execution stream needs to be provided; when the MCA error code specific recovery specific recovery action cannot be successfully completed, system software must shut down the system. When the AR flag in the IA32\_MCi\_STATUS register is clear, system software may still take MCA error code specific recovery action but this is optional; system software can safely resume program execution at the instruction pointer saved on the stack from the machine check exception when the RIP\_V flag in the IA32\_MCG\_STATUS register is set.

Both the S and the AR flags in the IA32\_MCi\_STATUS register are defined to be sticky bits, which mean that once set, the processor does not clear them. Only software and good power-on reset can clear the S and the AR-flags. Both the S and the AR flags are only set when the processor reports the UCR errors (MCG\_CAP[24] is set).

## 15.6.3 UCR Error Classification

With the S and AR flag encoding in the IA32\_MCi\_STATUS register, UCR errors can be classified as:

- Uncorrected no action required (UCNA) - is a UCR error that is not signaled via a machine check exception and, instead, is reported to system software as a corrected machine check error. UCNA errors indicate that some data in the system is corrupted, but the data has not been consumed and the processor state is valid and you may continue execution on this processor. UCNA errors require no action from system software to continue execution. A UCNA error is indicated with UC=1, PCC=0, S=0 and AR=0 in the IA32\_MCi\_STATUS register.
- Software recoverable action optional (SRAO) - a UCR error is signaled either via a machine check exception or CMCI. System software recovery action is optional and not required to continue execution from this machine check exception. SRAO errors indicate that some data in the system is corrupt, but the data has not been consumed and the processor state is valid. SRAO errors provide the additional error information for system software to perform a recovery action. An SRAO error when signaled as a machine check is indicated with UC=1, PCC=0, S=1, EN=1 and AR=0 in the IA32\_MCi\_STATUS register. In cases when SRAO is signaled via CMCI the error signature is indicated via UC=1, PCC=0, S=0. Recovery actions for SRAO errors are MCA error code specific. The MISC\_V and the ADDR\_V flags in the IA32\_MCi\_STATUS register are set when the additional error information is available from the IA32\_MCi\_MISC and the IA32\_MCi\_ADDR registers. System software needs to inspect the MCA error code fields in the IA32\_MCi\_STATUS register to identify the specific recovery



action for a given SRAO error. If MISCV and ADDRIV are not set, it is recommended that no system software error recovery be performed however, system software can resume execution.

- Software recoverable action required (SRAR) - a UCR error that requires system software to take a recovery action on this processor before scheduling another stream of execution on this processor. SRAR errors indicate that the error was detected and raised at the point of the consumption in the execution flow. An SRAR error is indicated with UC=1, PCC=0, S=1, EN=1 and AR=1 in the IA32\_MCi\_STATUS register. Recovery actions are MCA error code specific. The MISCV and the ADDRIV flags in the IA32\_MCi\_STATUS register are set when the additional error information is available from the IA32\_MCi\_MISC and the IA32\_MCi\_ADDR registers. System software needs to inspect the MCA error code fields in the IA32\_MCi\_STATUS register to identify the specific recovery action for a given SRAR error. If MISCV and ADDRIV are not set, it is recommended that system software shutdown the system.

Table 15-7 summarizes UCR, corrected, and uncorrected errors.

**Table 15-7. MC Error Classifications**

Type of Error <sup>1</sup>	UC	EN	PCC	S	AR	Signaling	Software Action	Example
Uncorrected Error (UC)	1	1	1	x	x	MCE	If EN=1, reset the system, else log and OK to keep the system running.	
SRAR	1	1	0	1	1	MCE	For known MCACOD, take specific recovery action; For unknown MCACOD, must bugcheck. If OVER=1, reset system, else take specific recovery action.	Cache to processor load error.
SRAO	1	x <sup>2</sup>	0	x <sup>2</sup>	0	MCE/CMC	For known MCACOD, take specific recovery action; For unknown MCACOD, OK to keep the system running.	Patrol scrub and explicit writeback poison errors.
UCNA	1	x	0	0	0	CMC	Log the error and Ok to keep the system running.	Poison detection error.
Corrected Error (CE)	0	x	x	x	x	CMC	Log the error and no corrective action required.	ECC in caches and memory.

**NOTES:**

1. SRAR, SRAO and UCNA errors are supported by the processor only when IA32\_MCG\_CAP[24] (MCG\_SER\_P) is set.
2. EN=1, S=1 when signaled via MCE. EN=x, S=0 when signaled via CMC.

### 15.6.4 UCR Error Overwrite Rules

In general, the overwrite rules are as follows:

- UCR errors will overwrite corrected errors.
- Uncorrected (PCC=1) errors overwrite UCR (PCC=0) errors.
- UCR errors are not written over previous UCR errors.
- Corrected errors do not write over previous UCR errors.

Regardless of whether the 1st error is retained or the 2nd error is overwritten over the 1st error, the OVER flag in the IA32\_MCi\_STATUS register will be set to indicate an overflow condition. As the S flag and AR flag in the IA32\_MCi\_STATUS register are defined to be sticky flags, a second event cannot clear these 2 flags once set, however the MC bank information may be filled in for the 2nd error. The table below shows the overwrite rules and how to treat a second error if the first event is already logged in a MC bank along with the resulting bit setting of the UC, PCC, and AR flags in the IA32\_MCi\_STATUS register. As UCNA and SRAO errors do not require recovery action from system software to continue program execution, a system reset by system software is not required unless the AR flag or PCC flag is set for the UCR overflow case (OVER=1, VAL=1, UC=1, PCC=0).

Table 15-8 lists overwrite rules for uncorrected errors, corrected errors, and uncorrected recoverable errors.

**Table 15-8. Overwrite Rules for UC, CE, and UCR Errors**

First Event	Second Event	UC	PCC	S	AR	MCA Bank	Reset System
CE	UCR	1	0	0 if UCNA, else 1	1 if SRAR, else 0	second	yes, if AR=1
UCR	CE	1	0	0 if UCNA, else 1	1 if SRAR, else 0	first	yes, if AR=1
UCNA	UCNA	1	0	0	0	first	no
UCNA	SRAO	1	0	1	0	first	no
UCNA	SRAR	1	0	1	1	first	yes
SRAO	UCNA	1	0	1	0	first	no
SRAO	SRAO	1	0	1	0	first	no
SRAO	SRAR	1	0	1	1	first	yes
SRAR	UCNA	1	0	1	1	first	yes
SRAR	SRAO	1	0	1	1	first	yes
SRAR	SRAR	1	0	1	1	first	yes
UCR	UC	1	1	undefined	undefined	second	yes
UC	UCR	1	1	undefined	undefined	first	yes

## 15.7 MACHINE-CHECK AVAILABILITY

The machine-check architecture and machine-check exception (#MC) are model-specific features. Software can execute the CPUID instruction to determine whether a processor implements these features. Following the execution of the CPUID instruction, the settings of the MCA flag (bit 14) and MCE flag (bit 7) in EDX indicate whether the processor implements the machine-check architecture and machine-check exception.

## 15.8 MACHINE-CHECK INITIALIZATION

To use the processors machine-check architecture, software must initialize the processor to activate the machine-check exception and the error-reporting mechanism.

Example 15-1 gives pseudocode for performing this initialization. This pseudocode checks for the existence of the machine-check architecture and exception; it then enables machine-check exception and the error-reporting register banks. The pseudocode shown is compatible with the Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors.

Following power up or power cycling, IA32\_MCi\_STATUS registers are not guaranteed to have valid data until after they are initially cleared to zero by software (as shown in the initialization pseudocode in Example 15-1).

### Example 15-1. Machine-Check Initialization Pseudocode

Check CPUID Feature Flags for MCE and MCA support

IF CPU supports MCE

THEN

IF CPU supports MCA

THEN

IF (IA32\_MCG\_CAP.MCG\_CTL\_P = 1)

(\* IA32\_MCG\_CTL register is present \*)

THEN

IA32\_MCG\_CTL ← FFFFFFFFFFFFFFFFH;

(\* enables all MCA features \*)

FI

IF (IA32\_MCG\_CAP.MCG\_LMCE\_P = 1 and IA32\_FEATURE\_CONTROL.LOCK = 1 and IA32\_FEATURE\_CONTROL.LMCE\_ENABLED = 1)

(\* IA32\_MCG\_EXT\_CTL register is present and platform has enabled LMCE to permit system software to use LMCE \*)



```

THEN
  IA32_MCG_EXT_CTL ← IA32_MCG_EXT_CTL | 01H;
  (* System software enables LMCE capability for hardware to signal MCE to a single logical processor*)
FI

(* Determine number of error-reporting banks supported *)
COUNT ← IA32_MCG_CAP.Count;
MAX_BANK_NUMBER ← COUNT - 1;

IF (Processor Family is 6H and Processor EXTMODEL:MODEL is less than 1AH)
THEN
  (* Enable logging of all errors except for MCO_CTL register *)
  FOR error-reporting banks (1 through MAX_BANK_NUMBER)
  DO
    IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFH;
  OD

ELSE
  (* Enable logging of all errors including MCO_CTL register *)
  FOR error-reporting banks (0 through MAX_BANK_NUMBER)
  DO
    IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFH;
  OD
FI

(* BIOS clears all errors only on power-on reset *)
IF (BIOS detects Power-on reset)
THEN
  FOR error-reporting banks (0 through MAX_BANK_NUMBER)
  DO
    IA32_MCi_STATUS ← 0;
  OD
ELSE
  FOR error-reporting banks (0 through MAX_BANK_NUMBER)
  DO
    (Optional for BIOS and OS) Log valid errors
    (OS only) IA32_MCi_STATUS ← 0;
  OD
FI
FI

Setup the Machine Check Exception (#MC) handler for vector 18 in IDT

Set the MCE bit (bit 6) in CR4 register to enable Machine-Check Exceptions
FI

```

## 15.9 INTERPRETING THE MCA ERROR CODES

When the processor detects a machine-check error condition, it writes a 16-bit error code to the MCA error code field of one of the IA32\_MCi\_STATUS registers and sets the VAL (valid) flag in that register. The processor may also write a 16-bit model-specific error code in the IA32\_MCi\_STATUS register depending on the implementation of the machine-check architecture of the processor.

The MCA error codes are architecturally defined for Intel 64 and IA-32 processors. To determine the cause of a machine-check exception, the machine-check exception handler must read the VAL flag for each IA32\_MCi\_STATUS register. If the flag is set, the machine check-exception handler must then read the MCA error code field of the register. It is the encoding of the MCA error code field [15:0] that determines the type of error being reported and not the register bank reporting it.

There are two types of MCA error codes: simple error codes and compound error codes.

## 15.9.1 Simple Error Codes

Table 15-9 shows the simple error codes. These unique codes indicate global error information.

**Table 15-9. IA32\_MCi\_Status [15:0] Simple Error Code Encoding**

Error Code	Binary Encoding	Meaning
No Error	0000 0000 0000 0000	No error has been reported to this bank of error-reporting registers.
Unclassified	0000 0000 0000 0001	This error has not been classified into the MCA error classes.
Microcode ROM Parity Error	0000 0000 0000 0010	Parity error in internal microcode ROM
External Error	0000 0000 0000 0011	The BINIT# from another processor caused this processor to enter machine check. <sup>1</sup>
FRC Error	0000 0000 0000 0100	FRC (functional redundancy check) <i>main/secondary</i> error.
Internal Parity Error	0000 0000 0000 0101	Internal parity error.
SMM Handler Code Access Violation	0000 0000 0000 0110	An attempt was made by the SMM Handler to execute outside the ranges specified by SMRR.
Internal Timer Error	0000 0100 0000 0000	Internal timer error.
I/O Error	0000 1110 0000 1011	generic I/O error.
Internal Unclassified	0000 01xx xxxx xxxx	Internal unclassified errors. <sup>2</sup>

### NOTES:

1. BINIT# assertion will cause a machine check exception if the processor (or any processor on the same external bus) has BINIT# observation enabled during power-on configuration (hardware strapping) and if machine check exceptions are enabled (by setting CR4.MCE = 1).
2. At least one X must equal one. Internal unclassified errors have not been classified.

## 15.9.2 Compound Error Codes

Compound error codes describe errors related to the TLBs, memory, caches, bus and interconnect logic, and internal timer. A set of sub-fields is common to all of compound errors. These sub-fields describe the type of access, level in the cache hierarchy, and type of request. Table 15-10 shows the general form of the compound error codes.

**Table 15-10. IA32\_MCi\_Status [15:0] Compound Error Code Encoding**

Type	Form	Interpretation
Generic Cache Hierarchy	000F 0000 0000 11LL	Generic cache hierarchy error
TLB Errors	000F 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory Controller Errors	000F 0000 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Cache Hierarchy Errors	000F 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_ERR
Extended Memory Errors	000F 0010 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Bus and Interconnect Errors	000F 1PPT RRRR IILL	BUS{LL}_{PP}_{RRRR}_{II}_{T}_ERR

The “Interpretation” column in the table indicates the name of a compound error. The name is constructed by substituting mnemonics for the sub-field names given within curly braces. For example, the error code ICACHEL1\_RD\_ERR is constructed from the form:

{TT}CACHE{LL}\_{RRRR}\_ERR,

where {TT} is replaced by I, {LL} is replaced by L1, and {RRRR} is replaced by RD.

For more information on the “Form” and “Interpretation” columns, see Sections Section 15.9.2.1, “Correction Report Filtering (F) Bit” through Section 15.9.2.5, “Bus and Interconnect Errors”.

### 15.9.2.1 Correction Report Filtering (F) Bit

Starting with Intel Core Duo processors, bit 12 in the “Form” column in Table 15-10 is used to indicate that a particular posting to a log may be the last posting for corrections in that line/entry, at least for some time:

- 0 in bit 12 indicates “normal” filtering (original P6/Pentium4/Atom/Xeon processor meaning).
- 1 in bit 12 indicates “corrected” filtering (filtering is activated for the line/entry in the posting). Filtering means that some or all of the subsequent corrections to this entry (in this structure) will not be posted. The enhanced error reporting introduced with the Intel Core Duo processors is based on tracking the lines affected by repeated corrections (see Section 15.4, “Enhanced Cache Error reporting”). This capability is indicated by IA32\_MCG\_CAP[11]. Only the first few correction events for a line are posted; subsequent redundant correction events to the same line are not posted. Uncorrected events are always posted.

The behavior of error filtering after crossing the yellow threshold is model-specific. Filtering has meaning only for corrected errors (UC=0 in IA32\_MCi\_STATUS MSR). System software must ignore filtering bit (12) for uncorrected errors.

### 15.9.2.2 Transaction Type (TT) Sub-Field

The 2-bit TT sub-field (Table 15-11) indicates the type of transaction (data, instruction, or generic). The sub-field applies to the TLB, cache, and interconnect error conditions. Note that interconnect error conditions are primarily associated with P6 family and Pentium processors, which utilize an external APIC bus separate from the system bus. The generic type is reported when the processor cannot determine the transaction type.

**Table 15-11. Encoding for TT (Transaction Type) Sub-Field**

Transaction Type	Mnemonic	Binary Encoding
Instruction	I	00
Data	D	01
Generic	G	10

### 15.9.2.3 Level (LL) Sub-Field

The 2-bit LL sub-field (see Table 15-12) indicates the level in the memory hierarchy where the error occurred (level 0, level 1, level 2, or generic). The LL sub-field also applies to the TLB, cache, and interconnect error conditions. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support two levels in the cache hierarchy and one level in the TLBs. Again, the generic type is reported when the processor cannot determine the hierarchy level.

**Table 15-12. Level Encoding for LL (Memory Hierarchy Level) Sub-Field**

Hierarchy Level	Mnemonic	Binary Encoding
Level 0	L0	00
Level 1	L1	01
Level 2	L2	10
Generic	LG	11

### 15.9.2.4 Request (RRRR) Sub-Field

The 4-bit RRRR sub-field (see Table 15-13) indicates the type of action associated with the error. Actions include read and write operations, prefetches, cache evictions, and snoops. Generic error is returned when the type of error cannot be determined. Generic read and generic write are returned when the processor cannot determine the type of instruction or data request that caused the error. Eviction and snoop requests apply only to the caches. All of the other requests apply to TLBs, caches and interconnects.

**Table 15-13. Encoding of Request (RRRR) Sub-Field**

Request Type	Mnemonic	Binary Encoding
Generic Error	ERR	0000
Generic Read	RD	0001
Generic Write	WR	0010
Data Read	DRD	0011
Data Write	DWR	0100
Instruction Fetch	IRD	0101
Prefetch	PREFETCH	0110
Eviction	EVICT	0111
Snoop	SNOOP	1000

### 15.9.2.5 Bus and Interconnect Errors

The bus and interconnect errors are defined with the 2-bit PP (participation), 1-bit T (time-out), and 2-bit II (memory or I/O) sub-fields, in addition to the LL and RRRR sub-fields (see Table 15-14). The bus error conditions are implementation dependent and related to the type of bus implemented by the processor. Likewise, the interconnect error conditions are predicated on a specific implementation-dependent interconnect model that describes the connections between the different levels of the storage hierarchy. The type of bus is implementation dependent, and as such is not specified in this document. A bus or interconnect transaction consists of a request involving an address and a response.

**Table 15-14. Encodings of PP, T, and II Sub-Fields**

Sub-Field	Transaction	Mnemonic	Binary Encoding
PP (Participation)	Local processor* originated request	SRC	00
	Local processor* responded to request	RES	01
	Local processor* observed error as third party	OBS	10
	Generic		11
T (Time-out)	Request timed out	TIMEOUT	1
	Request did not time out	NOTIMEOUT	0
II (Memory or I/O)	Memory Access	M	00
	Reserved		01
	I/O	IO	10
	Other transaction		11

**NOTE:**

\* Local processor differentiates the processor reporting the error from other system components (including the APIC, other processors, etc.).

### 15.9.2.6 Memory Controller and Extended Memory Errors

The memory controller errors are defined with the 3-bit MMM (memory transaction type), and 4-bit CCCC (channel) sub-fields. The encodings for MMM and CCCC are defined in Table 15-15. Extended Memory errors use the same encodings and are used to report errors in memory used as a cache.

**Table 15-15. Encodings of MMM and CCCC Sub-Fields**

Sub-Field	Transaction	Mnemonic	Binary Encoding
MMM	Generic undefined request	GEN	000
	Memory read error	RD	001
	Memory write error	WR	010
	Address/Command Error	AC	011
	Memory Scrubbing Error	MS	100
	Reserved		101-111
CCCC	Channel number	CHN	0000-1110
	Channel not specified		1111

Note that the CCCC channel number may be enumerated from zero separately by each memory controller on a system. On a multi-socket system, or a system with multiple memory controllers per socket, it is necessary to also consider which machine check bank logged the error. See Chapter 16 for details on specific implementations.

### 15.9.3 Architecturally Defined UCR Errors

Software recoverable compound error code are defined in this section.

#### 15.9.3.1 Architecturally Defined SRAO Errors

The following two SRAO errors are architecturally defined.

- UCR Errors detected by memory controller scrubbing; and
- UCR Errors detected during L3 cache (L3) explicit writebacks.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 15-10). Their values and compound encoding format are given in Table 15-16.

**Table 15-16. MCA Compound Error Code Encoding for SRAO Errors**

Type	MCACOD Value	MCA Error Code Encoding <sup>1</sup>
Memory Scrubbing	COH - CFH	0000_0000_1100_CCCC 000F 0000 1MMM CCCC (Memory Controller Error), where Memory subfield MMM = 100B (memory scrubbing) Channel subfield CCCC = channel # or generic
L3 Explicit Writeback	17AH	0000_0001_0111_1010 000F 0001 RRRR TTLL (Cache Hierarchy Error) where Request subfields RRRR = 0111B (Eviction) Transaction Type subfields TT = 10B (Generic) Level subfields LL = 10B

**NOTES:**

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 15-17 lists values of relevant bit fields of IA32\_MCi\_STATUS for architecturally defined SRAO errors.

**Table 15-17. IA32\_MCi\_STATUS Values for SRAO Errors**

SRAO Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR	MCACOD
Memory Scrubbing	1	0	1	x <sup>1</sup>	1	1	0	x <sup>1</sup>	0	COH-CFH
L3 Explicit Writeback	1	0	1	x <sup>1</sup>	1	1	0	x <sup>1</sup>	0	17AH

**NOTES:**

1. When signaled as MCE, EN=1 and S=1. If error was signaled via CMC, then EN=x, and S=0.

For both the memory scrubbing and L3 explicit writeback errors, the ADDRv and MISCV flags in the IA32\_MCi\_STATUS register are set to indicate that the offending physical address information is available from the IA32\_MCi\_MISC and the IA32\_MCi\_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32\_MCi\_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32\_MCi\_MISC register should indicate the lowest valid address bit in the address information provided from the IA32\_MCi\_ADDR register.

MCE signal is broadcast to all logical processors as outlined in Section 15.10.4.1. If LMCE is supported and enabled, some errors (not limited to UCR errors) may be delivered to only a single logical processor. System software should consult IA32\_MCG\_STATUS.LMCE\_S to determine if the MCE signaled is only to this logical processor.

IA32\_MCi\_STATUS banks can be shared by logical processors within a core or within the same package. So several logical processors may find an SRAO error in the shared IA32\_MCi\_STATUS bank but other processors do not find it in any of the IA32\_MCi\_STATUS banks. Table 15-18 shows the RIPV and EIPV flag indication in the IA32\_MCG\_STATUS register for the memory scrubbing and L3 explicit writeback errors on both the reporting and non-reporting logical processors.

**Table 15-18. IA32\_MCG\_STATUS Flag Indication for SRAO Errors**

SRAO Type	Reporting Logical Processors		Non-reporting Logical Processors	
	RIPV	EIPV	RIPV	EIPV
Memory Scrubbing	1	0	1	0
L3 Explicit Writeback	1	0	1	0

### 15.9.3.2 Architecturally Defined SRAR Errors

The following two SRAR errors are architecturally defined.

- UCR Errors detected on data load; and
- UCR Errors detected on instruction fetch.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 15-10). Their values and compound encoding format are given in Table 15-19.

**Table 15-19. MCA Compound Error Code Encoding for SRAR Errors**

Type	MCACOD Value	MCA Error Code Encoding <sup>1</sup>
Data Load	134H	0000_0001_0011_0100 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0011B (Data Load) Transaction Type subfield TT= 01B (Data) Level subfield LL = 00B (Level 0)
Instruction Fetch	150H	0000_0001_0101_0000 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0101B (Instruction Fetch) Transaction Type subfield TT= 00B (Instruction) Level subfield LL = 00B (Level 0)

**NOTES:**

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 15-20 lists values of relevant bit fields of IA32\_MCi\_STATUS for architecturally defined SRAR errors.

**Table 15-20. IA32\_MCi\_STATUS Values for SRAR Errors**

SRAR Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR	MCACOD
Data Load	1	0	1	1	1	1	0	1	1	134H
Instruction Fetch	1	0	1	1	1	1	0	1	1	150H

For both the data load and instruction fetch errors, the ADDRv and MISCV flags in the IA32\_MCi\_STATUS register are set to indicate that the offending physical address information is available from the IA32\_MCi\_MISC and the IA32\_MCi\_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32\_MCi\_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32\_MCi\_MISC register should indicate the lowest valid address bit in the address information provided from the IA32\_MCi\_ADDR register.

MCE signal is broadcast to all logical processors on the system on which the UCR errors are supported, except when the processor supports LMCE and LMCE is enabled by system software (see Section 15.3.1.5). The IA32\_MCG\_STATUS MSR allows system software to distinguish the affected logical processor of an SRAR error amongst logical processors that observed SRAR via MCI\_STATUS bank.

Table 15-21 shows the RIPV and EIPV flag indication in the IA32\_MCG\_STATUS register for the data load and instruction fetch errors on both the reporting and non-reporting logical processors. The recoverable SRAR error reported by a processor may be continuable, where the system software can interpret the context of continuable as follows: the error was isolated, contained. If software can rectify the error condition in the current instruction stream, the execution context on that logical processor can be continued without loss of information.

**Table 15-21. IA32\_MCG\_STATUS Flag Indication for SRAR Errors**

SRAR Type	Affected Logical Processor			Non-Affected Logical Processors		
	RIPV	EIPV	Continuable	RIPV	EIPV	Continuable
Recoverable-continuable	1	1	Yes <sup>1</sup>	1	0	Yes
Recoverable-not-continuable	0	x	No			

**NOTES:**

1. see the definition of the context of "continuable" above and additional detail below.

**SRAR Error And Affected Logical Processors**

The affected logical processor is the one that has detected and raised an SRAR error at the point of the consumption in the execution flow. The affected logical processor should find the Data Load or the Instruction Fetch error information in the IA32\_MCi\_STATUS register that is reporting the SRAR error.

Table 15-21 list the actionable scenarios that system software can respond to an SRAR error on an affected logical processor according to RIPV and EIPV values:

- Recoverable-Continuable SRAR Error (RIPV=1, EIPV=1):  
 For Recoverable-Continuable SRAR errors, the affected logical processor should find that both the IA32\_MCG\_STATUS.RIPV and the IA32\_MCG\_STATUS.EIPV flags are set, indicating that system software may be able to restart execution from the interrupted context if it is able to rectify the error condition. If system software cannot rectify the error condition then it must treat the error as a recoverable error where restarting execution with the interrupted context is not possible. Restarting without rectifying the error condition will result in most cases with another SRAR error on the same instruction.

- Recoverable-not-continuable SRAR Error (RIPV=0, EIPV=x):

For Recoverable-not-continuable errors, the affected logical processor should find that either

- IA32\_MCG\_STATUS.RIPV= 0, IA32\_MCG\_STATUS.EIPV=1, or
- IA32\_MCG\_STATUS.RIPV= 0, IA32\_MCG\_STATUS.EIPV=0.

In either case, this indicates that the error is detected at the instruction pointer saved on the stack for this machine check exception and restarting execution with the interrupted context is not possible. System software may take the following recovery actions for the affected logical processor:

- The current executing thread cannot be continued. System software must terminate the interrupted stream of execution and provide a new stream of execution on return from the machine check handler for the affected logical processor.

### SRAR Error And Non-Affected Logical Processors

The logical processors that observed but not affected by an SRAR error should find that the RIPV flag in the IA32\_MCG\_STATUS register is set and the EIPV flag in the IA32\_MCG\_STATUS register is cleared, indicating that it is safe to restart the execution at the instruction saved on the stack for the machine check exception on these processors after the recovery action is successfully taken by system software.

## 15.9.4 Multiple MCA Errors

When multiple MCA errors are detected within a certain detection window, the processor may aggregate the reporting of these errors together as a single event, i.e. a single machine exception condition. If this occurs, system software may find multiple MCA errors logged in different MC banks on one logical processor or find multiple MCA errors logged across different processors for a single machine check broadcast event. In order to handle multiple UCR errors reported from a single machine check event and possibly recover from multiple errors, system software may consider the following:

- Whether it can recover from multiple errors is determined by the most severe error reported on the system. If the most severe error is found to be an unrecoverable error (VAL=1, UC=1, PCC=1 and EN=1) after system software examines the MC banks of all processors to which the MCA signal is broadcast, recovery from the multiple errors is not possible and system software needs to reset the system.
- When multiple recoverable errors are reported and no other fatal condition (e.g. overflowed condition for SRAR error) is found for the reported recoverable errors, it is possible for system software to recover from the multiple recoverable errors by taking necessary recovery action for each individual recoverable error. However, system software can no longer expect one to one relationship with the error information recorded in the IA32\_MCi\_STATUS register and the states of the RIPV and EIPV flags in the IA32\_MCG\_STATUS register as the states of the RIPV and the EIPV flags in the IA32\_MCG\_STATUS register may indicate the information for the most severe error recorded on the processor. System software is required to use the RIPV flag indication in the IA32\_MCG\_STATUS register to make a final decision of recoverability of the errors and find the restart-ability requirement after examining each IA32\_MCi\_STATUS register error information in the MC banks.

In certain cases where system software observes more than one SRAR error logged for a single logical processor, it can no longer rely on affected threads as specified in Table 15-20 above. System software is recommended to reset the system if this condition is observed.

## 15.9.5 Machine-Check Error Codes Interpretation

Chapter 16, "Interpreting Machine-Check Error Codes," provides information on interpreting the MCA error code, model-specific error code, and other information error code fields. For P6 family processors, information has been included on decoding external bus errors. For Pentium 4 and Intel Xeon processors; information is included on external bus, internal timer and cache hierarchy errors.



## 15.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE

The machine-check architecture and error logging can be used in three different ways:

- To detect machine errors during normal instruction execution, using the machine-check exception (#MC).
- To periodically check and log machine errors.
- To examine recoverable UCR errors, determine software recoverability and perform recovery actions via a machine-check exception handler or a corrected machine-check interrupt handler.

To use the machine-check exception, the operating system or executive software must provide a machine-check exception handler. This handler may need to be designed specifically for each family of processors.

A special program or utility is required to log machine errors.

Guidelines for writing a machine-check exception handler or a machine-error logging utility are given in the following sections.

### 15.10.1 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When IA32\_MCG\_CAP[24] is clear, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32\_MCG\_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.
- The VAL (valid) flag in each IA32\_MCi\_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32\_MCi\_STATUS register should be checked. See Section 15.9, "Interpreting the MCA Error Codes," for information that can be used to write an algorithm to interpret this field.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32\_MCi\_STATUS register indicates whether the processor automatically corrected an error.
- The RIPV, PCC, and OVER flags in each IA32\_MCi\_STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- The RIPV flag in the IA32\_MCG\_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32\_MCG\_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.
- The MCIP flag in the IA32\_MCG\_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture

does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

Example 15-2 gives typical steps carried out by a machine-check exception handler.

### Example 15-2. Machine-Check Exception Handler Pseudocode

```

IF CPU supports MCE
  THEN
    IF CPU supports MCA
      THEN
        call errorlogging routine; (* returns restartability *)
      FI;
    ELSE (* Pentium(R) processor compatible *)
      READ P5_MC_ADDR
      READ P5_MC_TYPE;
      report RESTARTABILITY to console;
    FI;
  IF error is not restartable
    THEN
      report RESTARTABILITY to console;
      abort system;
    FI;
  CLEAR MCIP flag in IA32_MCG_STATUS;

```

## 15.10.2 Pentium Processor Machine-Check Exception Handling

Machine-check exception handler on P6 family, Intel Atom and later processor families, should follow the guidelines described in Section 15.10.1 and Example 15-2 that check the processor's support of MCA.

### NOTE

On processors that support MCA (CPUID.1.EDX.MCA = 1) reading the P5\_MC\_TYPE and P5\_MC\_ADDR registers may produce invalid data.

When machine-check exceptions are enabled for the Pentium processor (MCE flag is set in control register CR4), the machine-check exception handler uses the RDMSR instruction to read the error type from the P5\_MC\_TYPE register and the machine check address from the P5\_MC\_ADDR register. The handler then normally reports these register values to the system console before aborting execution (see Example 15-2).

## 15.10.3 Logging Correctable Machine-Check Errors

The error handling routine for servicing the machine-check exceptions is responsible for logging uncorrected errors.

If a machine-check error is correctable, the processor does not generate a machine-check exception for it. To detect correctable machine-check errors, a utility program must be written that reads each of the machine-check error-reporting register banks and logs the results in an accounting file or data structure. This utility can be implemented in either of the following ways.

- A system daemon that polls the register banks on an infrequent basis, such as hourly or daily.
- A user-initiated application that polls the register banks and records the exceptions. Here, the actual polling service is provided by an operating-system driver or through the system call interface.
- An interrupt service routine servicing CMCI can read the MC banks and log the error. Please refer to Section 15.10.4.2 for guidelines on logging correctable machine checks.

Example 15-3 gives pseudocode for an error logging utility.

**Example 15-3. Machine-Check Error Logging Pseudocode**

```

Assume that execution is restartable;
IF the processor supports MCA
  THEN
    FOR each bank of machine-check registers
      DO
        READ IA32_MCi_STATUS;
        IF VAL flag in IA32_MCi_STATUS = 1
          THEN
            IF ADDR_V flag in IA32_MCi_STATUS = 1
              THEN READ IA32_MCi_ADDR;
            FI;
            IF MISC_V flag in IA32_MCi_STATUS = 1
              THEN READ IA32_MCi_MISC;
            FI;
            IF MCIP flag in IA32_MCG_STATUS = 1
              (* Machine-check exception is in progress *)
              AND PCC flag in IA32_MCi_STATUS = 1
              OR RIPV flag in IA32_MCG_STATUS = 0
              (* execution is not restartable *)
              THEN
                RESTARTABILITY = FALSE;
                return RESTARTABILITY to calling procedure;
            FI;
            Save time-stamp counter and processor ID;
            Set IA32_MCi_STATUS to all 0s;
            Execute serializing instruction (i.e., CPUID);
          FI;
      OD;
    FI;

```

If the processor supports the machine-check architecture, the utility reads through the banks of error-reporting registers looking for valid register entries. It then saves the values of the IA32\_MCi\_STATUS, IA32\_MCi\_ADDR, IA32\_MCi\_MISC and IA32\_MCG\_STATUS registers for each bank that is valid. The routine minimizes processing time by recording the raw data into a system data structure or file, reducing the overhead associated with polling. User utilities analyze the collected data in an off-line environment.

When the MCIP flag is set in the IA32\_MCG\_STATUS register, a machine-check exception is in progress and the machine-check exception handler has called the exception logging routine.

Once the logging process has been completed the exception-handling routine must determine whether execution can be restarted, which is usually possible when damage has not occurred (The PCC flag is clear, in the IA32\_MCi\_STATUS register) and when the processor can guarantee that execution is restartable (the RIPV flag is set in the IA32\_MCG\_STATUS register). If execution cannot be restarted, the system is not recoverable and the exception-handling routine should signal the console appropriately before returning the error status to the Operating System kernel for subsequent shutdown.

The machine-check architecture allows buffering of exceptions from a given error-reporting bank although the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors do not implement this feature. The error logging routine should provide compatibility with future processors by reading each hardware error-reporting bank's IA32\_MCi\_STATUS register and then writing 0s to clear the OVER and VAL flags in this register. The error logging utility should re-read the IA32\_MCi\_STATUS register for the bank ensuring that the valid bit is clear. The processor will write the next error into the register bank and set the VAL flags.

Additional information that should be stored by the exception-logging routine includes the processor's time-stamp counter value, which provides a mechanism to indicate the frequency of exceptions. A multiprocessing operating system stores the identity of the processor node incurring the exception using a unique identifier, such as the processor's APIC ID (see Section 10.8, "Handling Interrupts").

The basic algorithm given in Example 15-3 can be modified to provide more robust recovery techniques. For example, software has the flexibility to attempt recovery using information unavailable to the hardware. Specifically, the machine-check exception handler can, after logging carefully analyze the error-reporting registers when the error-logging routine reports an error that does not allow execution to be restarted. These recovery techniques

can use external bus related model-specific information provided with the error report to localize the source of the error within the system and determine the appropriate recovery strategy.

## 15.10.4 Machine-Check Software Handler Guidelines for Error Recovery

### 15.10.4.1 Machine-Check Exception Handler for Error Recovery

When writing a machine-check exception (MCE) handler to support software recovery from Uncorrected Recoverable (UCR) errors, consider the following:

- When IA32\_MCG\_CAP [24] is zero, there are no recoverable errors supported and all machine-check are fatal exceptions. The logging of status and error information is therefore a baseline implementation requirement.
- When IA32\_MCG\_CAP [24] is 1, certain uncorrected errors called uncorrected recoverable (UCR) errors may be software recoverable. The handler can analyze the reported error information, and in some cases attempt to recover from the uncorrected error and continue execution.
- For processors on which CPUID reports DisplayFamily\_DisplayModel as 06H\_0EH and onward, an MCA signal is broadcast to all logical processors in the system (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). Due to the potentially shared machine check MSR resources among the logical processors on the same package/core, the MCE handler may be required to synchronize with the other processors that received a machine check error and serialize access to the machine check registers when analyzing, logging and clearing the information in the machine check registers.
  - On processors that indicate ability for local machine-check exception (MCG\_LMCE\_P), hardware can choose to report the error to only a single logical processor if system software has enabled LMCE by setting IA32\_MCG\_EXT\_CTL[LMCE\_EN] = 1 as outlined in Section 15.3.1.5.
- The VAL (valid) flag in each IA32\_MCi\_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and should not be checked.
- The MCE handler is primarily responsible for processing uncorrected errors. The UC flag in each IA32\_MCi\_Status register indicates whether the reported error was corrected (UC=0) or uncorrected (UC=1). The MCE handler can optionally log and clear the corrected errors in the MC banks if it can implement software algorithm to avoid the undesired race conditions with the CMCI or CMC polling handler.
- For uncorrectable errors, the EIPV flag in the IA32\_MCG\_STATUS register indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- The MCIP flag in the IA32\_MCG\_STATUS register indicates whether a machine-check exception was generated. When a machine check exception is generated, it is expected that the MCIP flag in the IA32\_MCG\_STATUS register is set to 1. If it is not set, this machine check was generated by either an INT 18 instruction or some piece of hardware signaling an interrupt with vector 18.

When IA32\_MCG\_CAP [24] is 1, the following rules can apply when writing a machine check exception (MCE) handler to support software recovery:

- The PCC flag in each IA32\_MCi\_STATUS register indicates whether recovery from the error is possible for uncorrected errors (UC=1). If the PCC flag is set for enabled uncorrected errors (UC=1 and EN=1), recovery is not possible. When recovery is not possible, the MCE handler typically records the error information and signals the operating system to reset the system.
- The RIPV flag in the IA32\_MCG\_STATUS register indicates whether restarting the program execution from the instruction pointer saved on the stack for the machine check exception is possible. When the RIPV is set, program execution can be restarted reliably when recovery is possible. If the RIPV flag is not set, program execution cannot be restarted reliably. In this case the recovery algorithm may involve terminating the current program execution and resuming an alternate thread of execution upon return from the machine check handler when recovery is possible. When recovery is not possible, the MCE handler signals the operating system to reset the system.

- When the EN flag is zero but the VAL and UC flags are one in the IA32\_MCi\_STATUS register, the reported uncorrected error in this bank is not enabled. As uncorrected errors with the EN flag = 0 are not the source of machine check exceptions, the MCE handler should log and clear non-enabled errors when the S bit is set and should continue searching for enabled errors from the other IA32\_MCi\_STATUS registers. Note that when IA32\_MCG\_CAP [24] is 0, any uncorrected error condition (VAL =1 and UC=1) including the one with the EN flag cleared are fatal and the handler must signal the operating system to reset the system. For the errors that do not generate machine check exceptions, the EN flag has no meaning.
- When the VAL flag is one, the UC flag is one, the EN flag is one and the PCC flag is zero in the IA32\_MCi\_STATUS register, the error in this bank is an uncorrected recoverable (UCR) error. The MCE handler needs to examine the S flag and the AR flag to find the type of the UCR error for software recovery and determine if software error recovery is possible.
- When both the S and the AR flags are clear in the IA32\_MCi\_STATUS register for the UCR error (VAL=1, UC=1, EN=x and PCC=0), the error in this bank is an uncorrected no-action required error (UCNA). UCNA errors are uncorrected but do not require any OS recovery action to continue execution. These errors indicate that some data in the system is corrupt, but that data has not been consumed and may not be consumed. If that data is consumed a non-UCNA machine check exception will be generated. UCNA errors are signaled in the same way as corrected machine check errors and the CMCI and CMC polling handler is primarily responsible for handling UCNA errors. Like corrected errors, the MCA handler can optionally log and clear UCNA errors as long as it can avoid the undesired race condition with the CMCI or CMC polling handler. As UCNA errors are not the source of machine check exceptions, the MCA handler should continue searching for uncorrected or software recoverable errors in all other MC banks.
- When the S flag in the IA32\_MCi\_STATUS register is set for the UCR error ((VAL=1, UC=1, EN=1 and PCC=0), the error in this bank is software recoverable and it was signaled through a machine-check exception. The AR flag in the IA32\_MCi\_STATUS register further clarifies the type of the software recoverable errors.
- When the AR flag in the IA32\_MCi\_STATUS register is clear for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action optional (SRAO) error. The MCE handler and the operating system can analyze the IA32\_MCi\_STATUS [15:0] to implement MCA error code specific optional recovery action, but this recovery action is optional. System software can resume the program execution from the instruction pointer saved on the stack for the machine check exception when the RIPV flag in the IA32\_MCG\_STATUS register is set.
- Even if the OVER flag in the IA32\_MCi\_STATUS register is set for the SRAO error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=0), the MCE handler can take recovery action for the SRAO error logged in the IA32\_MCi\_STATUS register. Since the recovery action for SRAO errors is optional, restarting the program execution from the instruction pointer saved on the stack for the machine check exception is still possible for the overflowed SRAO error if the RIPV flag in the IA32\_MCG\_STATUS is set.
- When the AR flag in the IA32\_MCi\_STATUS register is set for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action required (SRAR) error. The MCE handler and the operating system must take recovery action in order to continue execution after the machine-check exception. The MCA handler and the operating system need to analyze the IA32\_MCi\_STATUS [15:0] to determine the MCA error code specific recovery action. If no recovery action can be performed, the operating system must reset the system.
- When the OVER flag in the IA32\_MCi\_STATUS register is set for the SRAR error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=1), the MCE handler cannot take recovery action as the information of the SRAR error in the IA32\_MCi\_STATUS register was potentially lost due to the overflow condition. Since the recovery action for SRAR errors must be taken, the MCE handler must signal the operating system to reset the system.
- When the MCE handler cannot find any uncorrected (VAL=1, UC=1 and EN=1) or any software recoverable errors (VAL=1, UC=1, EN=1, PCC=0 and S=1) in any of the IA32\_MCi banks of the processors, this is an unexpected condition for the MCE handler and the handler should signal the operating system to reset the system.
- Before returning from the machine-check exception handler, software must clear the MCIP flag in the IA32\_MCG\_STATUS register. The MCIP flag is used to detect recursion. The machine-check architecture does not support recursion. When the processor receives a machine check when MCIP is set, it automatically enters the shutdown state.

Example 15-4 gives pseudocode for an MC exception handler that supports recovery of UCR.

**Example 15-4. Machine-Check Error Handler Pseudocode Supporting UCR**

```

MACHINE CHECK HANDLER: (* Called from INT 18 handler *)
NOERROR = TRUE;
ProcessorCount = 0;
IF CPU supports MCA
  THEN
    RESTARTABILITY = TRUE;
    IF (Processor Family = 6 AND DisplayModel ≥ 0EH) OR (Processor Family > 6)
      THEN
        IF ( MCG_LMCE = 1)
          MCA_BROADCAST = FALSE;
        ELSE
          MCA_BROADCAST = TRUE;
        FI;
        Acquire SpinLock;
        ProcessorCount++; (* Allowing one logical processor at a time to examine machine check registers *)
        CALL MCA ERROR PROCESSING; (* returns RESTARTABILITY and NOERROR *)
      ELSE
        MCA_BROADCAST = FALSE;
        (* Implement a rendezvous mechanism with the other processors if necessary *)
        CALL MCA ERROR PROCESSING;
      FI;
    ELSE (* Pentium(R) processor compatible *)
      READ P5_MC_ADDR
      READ P5_MC_TYPE;
      RESTARTABILITY = FALSE;
    FI;
  FI;

IF NOERROR = TRUE
  THEN
    IF NOT (MCG_RIPV = 1 AND MCG_EIPV = 0)
      THEN
        RESTARTABILITY = FALSE;
      FI
    FI;

IF RESTARTABILITY = FALSE
  THEN
    Report RESTARTABILITY to console;
    Reset system;
  FI;

IF MCA_BROADCAST = TRUE
  THEN
    IF ProcessorCount = MAX_PROCESSORS
      AND NOERROR = TRUE
      THEN
        Report RESTARTABILITY to console;
        Reset system;
      FI;
    Release SpinLock;
    Wait till ProcessorCount = MAX_PROCESSORS on system;
    (* implement a timeout and abort function if necessary *)
  FI;
CLEAR IA32_MCG_STATUS;
RESUME Execution;
(* End of MACHINE CHECK HANDLER*)

```

```

MCA ERROR PROCESSING: (* MCA Error Processing Routine called from MCA Handler *)
IF MCIP flag in IA32_MCG_STATUS = 0
  THEN (* MCIP=0 upon MCA is unexpected *)
    RESTARTABILITY = FALSE;
  FI;

```

## MACHINE-CHECK ARCHITECTURE

FOR each bank of machine-check registers

DO

CLEAR\_MC\_BANK = FALSE;

READ IA32\_MCI\_STATUS;

IF VAL Flag in IA32\_MCI\_STATUS = 1

THEN

IF UC Flag in IA32\_MCI\_STATUS = 1

THEN

IF Bit 24 in IA32\_MCG\_CAP = 0

THEN (\* the processor does not support software error recovery \*)

RESTARTABILITY = FALSE;

NOERROR = FALSE;

GOTO LOG MCA REGISTER;

FI;

(\* the processor supports software error recovery \*)

IF EN Flag in IA32\_MCI\_STATUS = 0 AND OVER Flag in IA32\_MCI\_STATUS=0

THEN (\* It is a spurious MCA Log. Log and clear the register \*)

CLEAR\_MC\_BANK = TRUE;

GOTO LOG MCA REGISTER;

FI;

IF PCC = 1 and EN = 1 in IA32\_MCI\_STATUS

THEN (\* processor context might have been corrupted \*)

RESTARTABILITY = FALSE;

ELSE (\* It is a uncorrected recoverable (UCR) error \*)

IF S Flag in IA32\_MCI\_STATUS = 0

THEN

IF AR Flag in IA32\_MCI\_STATUS = 0

THEN (\* It is a uncorrected no action required (UCNA) error \*)

GOTO CONTINUE; (\* let CMCI and CMC polling handler to process \*)

ELSE

RESTARTABILITY = FALSE; (\* S=0, AR=1 is illegal \*)

FI

FI;

IF RESTARTABILITY = FALSE

THEN (\* no need to take recovery action if RESTARTABILITY is already false \*)

NOERROR = FALSE;

GOTO LOG MCA REGISTER;

FI;

(\* S in IA32\_MCI\_STATUS = 1 \*)

IF AR Flag in IA32\_MCI\_STATUS = 1

THEN (\* It is a software recoverable and action required (SRAR) error \*)

IF OVER Flag in IA32\_MCI\_STATUS = 1

THEN

RESTARTABILITY = FALSE;

NOERROR = FALSE;

GOTO LOG MCA REGISTER;

FI

IF MCACOD Value in IA32\_MCI\_STATUS is recognized

AND Current Processor is an Affected Processor

THEN

Implement MCACOD specific recovery action;

CLEAR\_MC\_BANK = TRUE;

ELSE

RESTARTABILITY = FALSE;

FI;

ELSE (\* It is a software recoverable and action optional (SRAO) error \*)

IF OVER Flag in IA32\_MCI\_STATUS = 0 AND

MCACOD in IA32\_MCI\_STATUS is recognized

THEN

Implement MCACOD specific recovery action;

FI;

CLEAR\_MC\_BANK = TRUE;

FI; AR

FI; PCC

NOERROR = FALSE;

```

        GOTO LOG MCA REGISTER;
    ELSE (* It is a corrected error; continue to the next IA32_MCi_STATUS *)
        GOTO CONTINUE;
    FI; UC
    FI; VAL
LOG MCA REGISTER:
    SAVE IA32_MCi_STATUS;
    If MISCV in IA32_MCi_STATUS
        THEN
            SAVE IA32_MCi_MISC;
        FI;
    IF ADDRv in IA32_MCi_STATUS
        THEN
            SAVE IA32_MCi_ADDR;
        FI;
    IF CLEAR_MC_BANK = TRUE
        THEN
            SET all 0 to IA32_MCi_STATUS;
            If MISCV in IA32_MCi_STATUS
                THEN
                    SET all 0 to IA32_MCi_MISC;
                FI;
            IF ADDRv in IA32_MCi_STATUS
                THEN
                    SET all 0 to IA32_MCi_ADDR;
                FI;
        FI;
    CONTINUE:
    OD;
(*END FOR *)
RETURN;
(* End of MCA ERROR PROCESSING*)

```

#### 15.10.4.2 Corrected Machine-Check Handler for Error Recovery

When writing a corrected machine check handler, which is invoked as a result of CMCI or called from an OS CMC Polling dispatcher, consider the following:

- The VAL (valid) flag in each IA32\_MCi\_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank does not contain valid error information and does not need to be checked.
- The CMCI or CMC polling handler is responsible for logging and clearing corrected errors. The UC flag in each IA32\_MCi\_Status register indicates whether the reported error was corrected (UC=0) or not (UC=1).
- When IA32\_MCG\_CAP [24] is one, the CMC handler is also responsible for logging and clearing uncorrected no-action required (UCNA) errors. When the UC flag is one but the PCC, S, and AR flags are zero in the IA32\_MCi\_STATUS register, the reported error in this bank is an uncorrected no-action required (UCNA) error. In cases when SRAO error are signaled as UCNA error via CMCI, software can perform recovery for those errors identified in Table 15-16.
- In addition to corrected errors and UCNA errors, the CMC handler optionally logs uncorrected (UC=1 and PCC=1), software recoverable machine check errors (UC=1, PCC=0 and S=1), but should avoid clearing those errors from the MC banks. Clearing these errors may result in accidentally removing these errors before these errors are actually handled and processed by the MCE handler for attempted software error recovery.

Example 15-5 gives pseudocode for a CMCI handler with UCR support.



**Example 15-5. Corrected Error Handler Pseudocode with UCR Support**

Corrected Error HANDLER: (\* Called from CMCI handler or OS CMC Polling Dispatcher\*)  
 IF CPU supports MCA

```

  THEN
    FOR each bank of machine-check registers
      DO
        READ IA32_MCI_STATUS;
        IF VAL flag in IA32_MCI_STATUS = 1
          THEN
            IF UC Flag in IA32_MCI_STATUS = 0 (* It is a corrected error *)
              THEN
                GOTO LOG CMC ERROR;
            ELSE
              IF Bit 24 in IA32_MCG_CAP = 0
                THEN
                  GOTO CONTINUE;
              FI;
              IF S Flag in IA32_MCI_STATUS = 0 AND AR Flag in IA32_MCI_STATUS = 0
                THEN (* It is a uncorrected no action required error *)
                  GOTO LOG CMC ERROR
              FI
              IF EN Flag in IA32_MCI_STATUS = 0
                THEN (* It is a spurious MCA error *)
                  GOTO LOG CMC ERROR
              FI;
            FI;
          FI;
          GOTO CONTINUE;
        LOG CMC ERROR:
        SAVE IA32_MCI_STATUS;
        If MISCV Flag in IA32_MCI_STATUS
          THEN
            SAVE IA32_MCI_MISC;
            SET all 0 to IA32_MCI_MISC;
          FI;
        IF ADDR_V Flag in IA32_MCI_STATUS
          THEN
            SAVE IA32_MCI_ADDR;
            SET all 0 to IA32_MCI_ADDR
          FI;
        SET all 0 to IA32_MCI_STATUS;
        CONTINUE:
      OD;
    (*END FOR *)
  FI;

```

## 19. Updates to Chapter 16, Volume 3B

Change bars and green text show changes to Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes to this chapter:

Changed "master" to "main" and "slave" to "secondary".

Intel is committed to ensuring technology is fully inclusive. In the spirit of this effort, we are actively working to replace instances of non-inclusive terms (e.g., Master / Slave) with more accurate and descriptive inclusive terms

# CHAPTER 16

## INTERPRETING MACHINE-CHECK ERROR CODES

Encoding of the model-specific and other information fields is different across processor families. The differences are documented in the following sections.

### 16.1 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H MACHINE ERROR CODES FOR MACHINE CHECK

Section 16.1 provides information for interpreting additional model-specific fields for external bus errors relating to processor family 06H. The references to processor family 06H refers to only IA-32 processors with CPUID signatures listed in Table 16-1.

**Table 16-1. CPUID DisplayFamily\_DisplayModel Signatures for Processor Family 06H**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_09H	Intel Pentium M processor
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon Processor, Intel Pentium III Processor
06_03H, 06_05H	Intel Pentium II Xeon Processor, Intel Pentium II Processor
06_01H	Intel Pentium Pro Processor

These errors are reported in the IA32\_MCi\_STATUS MSRs. They are reported architecturally as compound errors with a general form of *0000 1PPT RRRR IILL* in the MCA error code field. See Chapter 15 for information on the interpretation of compound error codes. Incremental decoding information is listed in Table 16-2.

**Table 16-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0		
Model specific errors	18:16	Reserved	Reserved
Model specific errors	24:19	Bus queue request type	000000 for BQ_DCU_READ_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error

**Table 16-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check**

Type	Bit No.	Bit Function	Bit Description
			001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSH2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error
Model specific errors	27:25	Bus queue error type	000 for BQ_ERR_HARD_TYPE error 001 for BQ_ERR_DOUBLE_TYPE error 010 for BQ_ERR_AERR2_TYPE error 100 for BQ_ERR_SINGLE_TYPE error 101 for BQ_ERR_AERR1_TYPE error
Model specific errors	28	FRC error	1 if FRC error active
	29	BERR	1 if BERR is driven
	30	Internal BINIT	1 if BINIT driven for this processor
	31	Reserved	Reserved
Other information	34:32	Reserved	Reserved
	35	External BINIT	1 if BINIT is received from external bus.
	36	Response parity error	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	Bus BINIT	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has received a hard error response on a split transaction one access that has needed to be split across the 64-bit external bus interface into two accesses).
	38	Timeout BINIT	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time.  A ROB time-out occurs when the 15-bit ROB time-out counter carries a 1 out of its high order bit. <sup>2</sup> The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs.  The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	41:39	Reserved	Reserved
	42	Hard error	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.

**Table 16-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check**

Type	Bit No.	Bit Function	Bit Description
	43	IERR	This bit is asserted in IA32_MCi_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.
	44	AERR	This bit is asserted in IA32_MCi_STATUS if this component has initiated 2 failing bus transactions which have failed due to Address Parity Errors AERR asserted). While this bit is asserted, it cannot be overwritten.
	45	UECC	The Uncorrectable ECC error bit is asserted in IA32_MCi_STATUS for uncorrected ECC errors. While this bit is asserted, the ECC syndrome field will not be overwritten.
	46	CECC	The correctable ECC error bit is asserted in IA32_MCi_STATUS for corrected ECC errors.
	54:47	ECC syndrome	The ECC syndrome field in IA32_MCi_STATUS contains the 8-bit ECC syndrome only if the error was a correctable/uncorrectable ECC error and there wasn't a previous valid ECC error syndrome logged in IA32_MCi_STATUS.  A previous valid ECC error in IA32_MCi_STATUS is indicated by IA32_MCi_STATUS.bit45 (uncorrectable error occurred) being asserted. After processing an ECC error, machine-check handling software should clear IA32_MCi_STATUS.bit45 so that future ECC error syndromes can be logged.
	56:55	Reserved	Reserved.
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.
2. For processors with a CPUID signature of 06\_OEH, a ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit.

## 16.2 INCREMENTAL DECODING INFORMATION: INTEL CORE 2 PROCESSOR FAMILY MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-4 provides information for interpreting additional model-specific fields for external bus errors relating to processor based on Intel Core microarchitecture, which implements the P4 bus specification. Table 16-3 lists the CPUID signatures for Intel 64 processors that are covered by Table 16-4. These errors are reported in the IA32\_MCi\_STATUS MSR. They are reported architecturally as compound errors with a general form of *0000 1PPT RRRR IILL* in the MCA error code field. See Chapter 15 for information on the interpretation of compound error codes.

**Table 16-3. CPUID DisplayFamily\_DisplayModel Signatures for Processors Based on Intel Core Microarchitecture**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_1DH	Intel Xeon Processor 7400 series.
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processor Q9650.
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad, Intel Core 2 Extreme, Intel Core 2 Duo processors, Intel Pentium dual-core processors.

**Table 16-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel Core Microarchitecture**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0		
Model specific errors	18:16	Reserved	Reserved
Model specific errors	24:19	Bus queue request type	'000001 for BQ_PREF_READ_TYPE error 000000 for BQ_DCU_READ_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error 001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSHL2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error 100100 for BQ_L2_WI_RFO_TYPE error 100110 for BQ_L2_WI_ITOM_TYPE error
Model specific errors	27:25	Bus queue error type	'001 for Address Parity Error '010 for Response Hard Error '011 for Response Parity Error
Model specific errors	28	MCE Driven	1 if MCE is driven
	29	MCE Observed	1 if MCE is observed
	30	Internal BINIT	1 if BINIT driven for this processor
	31	BINIT Observed	1 if BINIT is observed for this processor
Other information	33:32	Reserved	Reserved
	34	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	35	Reserved	Reserved

**Table 16-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel Core Microarchitecture (Contd.)**

Type	Bit No.	Bit Function	Bit Description
	36	Response parity error	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	38	Timeout BINIT	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time.  A ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs.  The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	41:39	Reserved	Reserved
	42	Hard error	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.
	43	IERR	This bit is asserted in IA32_MC <sub>i</sub> _STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.
	44	Reserved	Reserved
	45	Reserved	Reserved
	46	Reserved	Reserved
	54:47	Reserved	Reserved
	56:55	Reserved	Reserved.
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

**16.2.1 Model-Specific Machine Check Error Codes for Intel Xeon Processor 7400 Series**

Intel Xeon processor 7400 series has machine check register banks that generally follows the description of Chapter 15 and Section 16.2. Additional error codes specific to Intel Xeon processor 7400 series is describe in this section.

MC4\_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors for Intel Xeon processor 7400 series. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

### 16.2.1.1 Processor Machine Check Status Register Incremental MCA Error Code Definition

Intel Xeon processor 7400 series use compound MCA Error Codes for logging its Bus internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines incremental Machine Check error types (IA32\_MC6\_STATUS[15:0]) beyond those defined in Chapter 15. Table 16-5 lists these incremental MCA error code types that apply to IA32\_MC6\_STATUS. Error code details are specified in MC6\_STATUS [31:16] (see Section 16.2.2), the "Model Specific Error Code" field. The information in the "Other\_Info" field (MC4\_STATUS[56:32]) is common to the three processor error types and contains a correctable event count and specifies the MC6\_MISC register format.

**Table 16-5. Incremental MCA Error Code Types for Intel Xeon Processor 7400**

Processor MCA_Error_Code (MC6_STATUS[15:0])			
Type	Error Code	Binary Encoding	Meaning
C	Internal Error	0000 0100 0000 0000	Internal Error Type Code
B	Bus and Interconnect Error	0000 100x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 101x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 110x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 1110 0000 1111	Bus and Interconnection Error Type Code
		0000 1111 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations

The **Bold faced** binary encodings are the only encodings used by the processor for MC4\_STATUS[15:0].

## 16.2.2 Intel Xeon Processor 7400 Model Specific Error Code Field

### 16.2.2.1 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error

Note: The Model Specific Error Code field in MC6\_STATUS (bits 31:16).

**Table 16-6. Type B Bus and Interconnect Error Codes**

Bit Num	Sub-Field Name	Description
16	FSB Request Parity	Parity error detected during FSB request phase
19:17		Reserved
20	FSB Hard Fail Response	"Hard Failure" response received for a local transaction
21	FSB Response Parity	Parity error on FSB response field detected
22	FSB Data Parity	FSB data parity error on inbound data detected
31:23	---	Reserved



### 16.2.2.2 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error

**Table 16-7. Type C Cache Bus Controller Error Codes**

MC4_STATUS[31:16] (MSCE) Value	Error Description
0000_0000_0000_0001 0001H	Inclusion Error from Core 0
0000_0000_0000_0010 0002H	Inclusion Error from Core 1
0000_0000_0000_0011 0003H	Write Exclusive Error from Core 0
0000_0000_0000_0100 0004H	Write Exclusive Error from Core 1
0000_0000_0000_0101 0005H	Inclusion Error from FSB
0000_0000_0000_0110 0006H	SNP Stall Error from FSB
0000_0000_0000_0111 0007H	Write Stall Error from FSB
0000_0000_0000_1000 0008H	FSB Arb Timeout Error
0000_0000_0000_1010 000AH	Inclusion Error from Core 2
0000_0000_0000_1011 000BH	Write Exclusive Error from Core 2
0000_0010_0000_0000 0200H	Internal Timeout error
0000_0011_0000_0000 0300H	Internal Timeout Error
0000_0100_0000_0000 0400H	Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow
0000_0101_0000_0000 0500H	Quiet cycle Timeout Error (correctable)
1100_0000_0000_0010 C002H	Correctable ECC event on outgoing Core 0 data
1100_0000_0000_0100 C004H	Correctable ECC event on outgoing Core 1 data
1100_0000_0000_1000 C008H	Correctable ECC event on outgoing Core 2 data
1110_0000_0000_0010 E002H	Uncorrectable ECC error on outgoing Core 0 data
1110_0000_0000_0100 E004H	Uncorrectable ECC error on outgoing Core 1 data
1110_0000_0000_1000 E008H	Uncorrectable ECC error on outgoing Core 2 data
— all other encodings —	Reserved

## 16.3 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR 3400, 3500, 5500 SERIES, MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-8 through Table 16-12 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor 3400, 3500, 5500 series with CPUID DisplayFamily\_DisplaySignature 06\_1AH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32\_MC0 and IA32\_MC1, incremental error codes for internal machine check is reported in the register bank IA32\_MC7, and incremental error codes for the memory controller unit is reported in the register banks IA32\_MC8.

### 16.3.1 Intel QPI Machine Check Errors

**Table 16-8. Intel QPI Machine Check Error Codes for IA32\_MC0\_STATUS and IA32\_MC1\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors	16	Header Parity	If 1, QPI Header had bad parity
	17	Data Parity	If 1, QPI Data packet had bad parity
	18	Retries Exceeded	If 1, number of QPI retries was exceeded
	19	Received Poison	If 1, Received a data packet that was marked as poisoned by the sender
	21:20	Reserved	Reserved
	22	Unsupported Message	If 1, QPI received a message encoding it does not support
	23	Unsupported Credit	If 1, QPI credit type is not supported.
	24	Receive Flit Overrun	If 1, Sender sent too many QPI flits to the receiver.
	25	Received Failed Response	If 1, Indicates that sender sent a failed response to receiver.
	26	Receiver Clock Jitter	If 1, clock jitter detected in the internal QPI clocking
	56:27	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

**Table 16-9. Intel QPI Machine Check Error Codes for IA32\_MC0\_MISC and IA32\_MC1\_MISC**

Type	Bit No.	Bit Function	Bit Description
Model specific errors <sup>1</sup>	7:0	QPI Opcode	Message class and opcode from the packet with the error
	13:8	RTID	QPI Request Transaction ID
	15:14	Reserved	Reserved
	18:16	RHNID	QPI Requestor/Home Node ID
	23:19	Reserved	Reserved
	24	IIB	QPI Interleave/Head Indication Bit

**NOTES:**

1. Which of these fields are valid depends on the error type.

### 16.3.2 Internal Machine Check Errors

**Table 16-10. Machine Check Error Codes for IA32\_MC7\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
Model specific errors			

Type	Bit No.	Bit Function	Bit Description
	23:16	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 03h - Reset firmware did not complete 08h - Received an invalid CMPD 0Ah - Invalid Power Management Request 0Dh - Invalid S-state transition 11h - VID controller does not match POC controller selected 1Ah - MSID from POC does not match CPU MSID
	56:32	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

### 16.3.3 Memory Controller Errors

**Table 16-11. Incremental Memory Controller Error Codes of Machine Check for IA32\_MC8\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory error format: 1MMMCCCC
Model specific errors			
	16	Read ECC error	If 1, ECC occurred on a read
	17	RAS ECC error	If 1, ECC occurred on a scrub
	18	Write parity error	If 1, bad parity on a write
	19	Redundancy loss	If 1, Error in half of redundant memory
	20	Reserved	Reserved
	21	Memory range error	If 1, Memory access out of range
	22	RTID out of range	If 1, Internal ID invalid
	23	Address parity error	If 1, bad address parity
	24	Byte enable parity error	If 1, bad enable parity
Other information	37:25	Reserved	Reserved
	52:38	CORE_ERR_CNT	Corrected error count
	56:53	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

**Table 16-12. Incremental Memory Controller Error Codes of Machine Check for IA32\_MC8\_MISC**

Type	Bit No.	Bit Function	Bit Description
Model specific errors <sup>1</sup>			
	7:0	RTId	Transaction Tracker ID
	15:8	Reserved	Reserved
	17:16	DIMM	DIMM ID which got the error
	19:18	Channel	Channel ID which got the error
	31:20	Reserved	Reserved
	63:32	Syndrome	ECC Syndrome

**NOTES:**

1. Which of these fields are valid depends on the error type.

## 16.4 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-13 through Table 16-15 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor E5 Family with CPUID DisplayFamily\_DisplaySignature 06\_2DH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32\_MC6 and IA32\_MC7, incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32\_MC4, and incremental error codes for the memory controller unit is reported in the register banks IA32\_MC8-IA32\_MC11.

### 16.4.1 Internal Machine Check Errors

**Table 16-13. Machine Check Error Codes for IA32\_MC4\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
Model specific errors	19:16	Reserved except for the following	0000b - No Error 0001b - Non_IMem_Sel 0010b - I_Parity_Error 0011b - Bad_OpCode 0100b - I_Stack_Underflow 0101b - I_Stack_Overflow 0110b - D_Stack_Underflow 0111b - D_Stack_Overflow 1000b - Non-DMem_Sel 1001b - D_Parity_Error

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 0Dh - MC_IMC_FORCE_SR_S3_TIMEOUT 0Eh - MC_CPD_UNCPD_ST_TIMEOUT 0Fh - MC_PKGS_SAFE_WP_TIMEOUT 43h - MC_PECI_MAILBOX QUIESCE_TIMEOUT 5Ch - MC_MORE_THAN_ONE_LT_AGENT 60h - MC_INVALID_PKGS_REQ_PCH 61h - MC_INVALID_PKGS_REQ_QPI 62h - MC_INVALID_PKGS_RES_QPI 63h - MC_INVALID_PKGC_RES_PCH 64h - MC_INVALID_PKG_STATE_CONFIG 70h - MC_WATCHDG_TIMEOUT_PKGC_SECONDARY 71h - MC_WATCHDG_TIMEOUT_PKGC_MAIN 72h - MC_WATCHDG_TIMEOUT_PKGS_MAIN 7ah - MC_HA_FAILSTS_CHANGE_DETECTED 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.4.2 Intel QPI Machine Check Errors

**Table 16-14. Intel QPI MC Error Codes for IA32\_MC6\_STATUS and IA32\_MC7\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors			
	56:16	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.4.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC8\_STATUS-IA32\_MC11\_STATUS. The supported error codes are follows the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture,"). MSR\_ERROR\_CONTROL.[bit 1] can enable additional informa-

tion logging of the IMC. The additional error information logged by the IMC is stored in IA32\_MCi\_STATUS and IA32\_MCi\_MISC; (i = 8, 11).

**Table 16-15. Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 8, 11)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors	31:16	Reserved except for the following	001H - Address parity error 002H - HA Wrt buffer Data parity error 004H - HA Wrt byte enable parity error 008H - Corrected patrol scrub error 010H - Uncorrected patrol scrub error 020H - Corrected spare error 040H - Uncorrected spare error
Model specific errors	36:32	Other info	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first device error when corrected error is detected during normal read.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

**Table 16-16. Intel IMC MC Error Codes for IA32\_MCi\_MISC (i= 8, 11)**

Type	Bit No.	Bit Function	Bit Description
MCA addr info <sup>1</sup>	8:0		See Chapter 15, "Machine-Check Architecture,"
Model specific errors	13:9		<ul style="list-style-type: none"> <li>When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second device error when corrected error is detected during normal read.</li> <li>Otherwise contain parity error if MCI_Status indicates HA_WB_Data or HA_W_BE parity error.</li> </ul>
Model specific errors	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error bit mask.
Model specific errors	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error failing rank.
	58:56	Reserved	Reserved
	61:59	Reserved	Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data from the first correctable error in a memory device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

## 16.5 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V2 AND INTEL® XEON® PROCESSOR E7 V2 FAMILIES, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v2 family and the Intel® Xeon® processor E7 v2 family are based on the Ivy Bridge-EP microarchitecture and can be identified with CPUID DisplayFamily\_DisplaySignature 06\_3EH. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32\_MC4, Table lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32\_MC5. Information listed in Table 16-14 for QPI MC error code apply to IA32\_MC5\_STATUS. Incremental error codes for the memory controller unit is reported in the register banks IA32\_MC9-IA32\_MC16. Table 16-18 lists model-specific error codes apply to IA32\_MCi\_STATUS, i = 9-16.

### 16.5.1 Internal Machine Check Errors

Table 16-17. Machine Check Error Codes for IA32\_MC4\_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
Model specific errors	19:16	Reserved except for the following	0000b - No Error 0001b - Non_IMem_Sel 0010b - I_Parity_Error 0011b - Bad_OpCode 0100b - I_Stack_Underflow 0101b - I_Stack_Overflow 0110b - D_Stack_Underflow 0111b - D_Stack_Overflow 1000b - Non-DMem_Sel 1001b - D_Parity_Error
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 0Dh - MC_IMC_FORCE_SR_S3_TIMEOUT 0Eh - MC_CPD_UNCPD_ST_TIMEOUT 0Fh - MC_PKGS_SAFE_WP_TIMEOUT 43h - MC_PECI_MAILBOX QUIESCE_TIMEOUT 44h - MC_CRITICAL_VR_FAILED 45h - MC_ICC_MAX-NOTSUPPORTED 5Ch - MC_MORE_THAN_ONE_LT_AGENT 60h - MC_INVALID_PKGS_REQ_PCH 61h - MC_INVALID_PKGS_REQ_QPI 62h - MC_INVALID_PKGS_RES_QPI 63h - MC_INVALID_PKGC_RES_PCH 64h - MC_INVALID_PKG_STATE_CONFIG 70h - MC_WATCHDG_TIMEOUT_PKGC_SECONDARY 71h - MC_WATCHDG_TIMEOUT_PKGC_MAIN 72h - MC_WATCHDG_TIMEOUT_PKGS_MAIN

Type	Bit No.	Bit Function	Bit Description
			7Ah - MC_HA_FAILSTS_CHANGE_DETECTED 7Bh - MC_PCIE_R2PCIE-RW_BLOCK_ACK_TIMEOUT 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.5.2 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC9\_STATUS-IA32\_MC16\_STATUS. The supported error codes are follows the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

MSR\_ERROR\_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32\_MCi\_STATUS and IA32\_MCi\_MISC; (i = 9-16).

IA32\_MCi\_STATUS (i=9-12) log errors from the first memory controller. The second memory controller logs into IA32\_MCi\_STATUS (i=13-16).

**Table 16-18. Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 9-16)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory Controller error format: 000F 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	001H - Address parity error 002H - HA Wrt buffer Data parity error 004H - HA Wrt byte enable parity error 008H - Corrected patrol scrub error
			010H - Uncorrected patrol scrub error 020H - Corrected spare error 040H - Uncorrected spare error 080H - Corrected memory read error. (Only applicable with iMC's "Additional Error logging" Mode-1 enabled.) 100H - iMC, WDB, parity errors
	36:32	Other info	When MSR_ERROR_CONTROL.[1] is set, logs an encoded value from the first error device.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.



**Table 16-19. Intel IMC MC Error Codes for IA32\_MCi\_MISC (i= 9-16)**

Type	Bit No.	Bit Function	Bit Description
MCA addr info <sup>1</sup>	8:0		See Chapter 15, "Machine-Check Architecture,"
Model specific errors	13:9		If the error logged is MCWrDataPar error or MCWrBEP error, this field is the WDB ID that has the parity error. OR if the second error logged is a correctable read error, MC logs the second error device in this field.
Model specific errors	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
Model specific errors	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	61:56		Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from a correctable error from memory read associated with first error device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.5.3 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32\_MC7\_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32\_MC8\_{STATUS,ADDR,MISC}.

## 16.6 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V3 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v3 family is based on the Haswell-E microarchitecture and can be identified with CPUID DisplayFamily\_DisplaySignature 06\_3FH. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32\_MC4, Table 16-20 lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32\_MC5, IA32\_MC20, and IA32\_MC21. Information listed in Table 16-21 for QPI MC error codes. Incremental error codes for the memory controller unit is reported in the register banks IA32\_MC9-IA32\_MC16. Table 16-22 lists model-specific error codes apply to IA32\_MCi\_STATUS, i = 9-16.

### 16.6.1 Internal Machine Check Errors

**Table 16-20. Machine Check Error Codes for IA32\_MC4\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
MCACOD <sup>2</sup>	15:0	Internal Errors	0402h - PCU internal Errors 0403h - PCU internal Errors 0406h - Intel TXT Errors 0407h - Other UBOX internal Errors.  On an IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS (After a 3-strike, the core MCA banks will be unavailable).
Model specific errors	19:16	Reserved except for the following	0000b - No Error 00xxb - PCU internal error
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 09h - MC_MESSAGE_CHANNEL_TIMEOUT 13h - MC_DMI_TRAINING_TIMEOUT 15h - MC_DMI_CPU_RESET_ACK_TIMEOUT 1Eh - MC_VR_ICC_MAX_LT_FUSED_ICC_MAX 25h - MC_SVID_COMMAND_TIMEOUT 29h - MC_VR_VOUT_MAC_LT_FUSED_SVID 2Bh - MC_PKGC_WATCHDOG_HANG_CBZ_DOWN 2Ch - MC_PKGC_WATCHDOG_HANG_CBZ_UP  44h - MC_CRITICAL_VR_FAILED 46h - MC_VID_RAMP_DOWN_FAILED 49h - MC_SVID_WRITE_REG_VOUT_MAX_FAILED 4Bh - MC_BOOT_VID_TIMEOUT. Timeout setting boot VID for DRAM 0. 4Fh - MC_SVID_COMMAND_ERROR. 52h - MC_FIVR_CATAS_OVERVOL_FAULT. 53h - MC_FIVR_CATAS_OVERCUR_FAULT. 57h - MC_SVID_PKGC_REQUEST_FAILED 58h - MC_SVID_IMON_REQUEST_FAILED 59h - MC_SVID_ALERT_REQUEST_FAILED 62h - MC_INVALID_PKGS_RSP_QPI 64h - MC_INVALID_PKG_STATE_CONFIG 67h - MC_HA_IMC_Rw_BLOCK_ACK_TIMEOUT 6Ah - MC_MSGCH_PMREQ_CMP_TIMEOUT 72h - MC_WATCHDG_TIMEOUT_PKGS_MASTER 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

2. The internal error codes may be model-specific.

### 16.6.2 Intel QPI Machine Check Errors

MC error codes associated with the Intel QPI agents are reported in the MSRs IA32\_MC5\_STATUS, IA32\_MC20\_STATUS, and IA32\_MC21\_STATUS. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL (see Chapter 15, “Machine-Check Architecture,”).

Table 16-21 lists model-specific fields to interpret error codes applicable to IA32\_MC5\_STATUS, IA32\_MC20\_STATUS, and IA32\_MC21\_STATUS.

**Table 16-21. Intel QPI MC Error Codes for IA32\_MCi\_STATUS (i = 5, 20, 21)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors	31:16	MSCOD	02h - Intel QPI physical layer detected drift buffer alarm.
			03h - Intel QPI physical layer detected latency buffer rollover.
			10h - Intel QPI link layer detected control error from R3QPI.
			11h - Rx entered LLR abort state on CRC error.
			12h - Unsupported or undefined packet.
			13h - Intel QPI link layer control error.
			15h - RBT used un-initialized value.
			20h - Intel QPI physical layer detected a QPI in-band reset but aborted initialization
			21h - Link failover data self-healing
			22h - Phy detected in-band reset (no width change).
			23h - Link failover clock failover
			30h -Rx detected CRC error - successful LLR after Phy re-init.
			31h -Rx detected CRC error - successful LLR without Phy re-init.
			All other values are reserved.
	37:32	Reserved	Reserved
	52:38	Corrected Error Cnt	
	56:53	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

### 16.6.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC9\_STATUS-IA32\_MC16\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture,”).

MSR\_ERROR\_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32\_MCi\_STATUS and IA32\_MCi\_MISC; (i = 9-16).

IA32\_MCi\_STATUS (i=9-12) log errors from the first memory controller. The second memory controller logs into IA32\_MCi\_STATUS (i=13-16).

**Table 16-22. Intel IMC MC Error Codes for IA32\_MCI\_STATUS (i= 9-16)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - DDR3 address parity error 0002H - Uncorrected HA write data error  0004H - Uncorrected HA data byte enable error 0008H - Corrected patrol scrub error 0010H - Uncorrected patrol scrub error 0020H - Corrected spare error 0040H - Uncorrected spare error 0080H - Corrected memory read error. (Only applicable with iMC's "Additional Error logging" Mode-1 enabled.) 0100H - iMC, write data buffer parity errors 0200H - DDR4 command address parity error
	36:32	Other info	When MSR_ERROR_CONTROL.[1] is set, logs an encoded value from the first error device.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

**Table 16-23. Intel IMC MC Error Codes for IA32\_MCI\_MISC (i= 9-16)**

Type	Bit No.	Bit Function	Bit Description
MCA addr info <sup>1</sup>	8:0		See Chapter 15, "Machine-Check Architecture,"
Model specific errors	13:9		If the error logged is MCWrDataPar error or MCWrBEPPar error, this field is the WDB ID that has the parity error. OR if the second error logged is a correctable read error, MC logs the second error device in this field.
Model specific errors	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
Model specific errors	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	61:56		Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from a correctable error from memory read associated with first error device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

**16.6.4 Home Agent Machine Check Errors**

Memory errors from the first memory controller may be logged in the IA32\_MC7\_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32\_MC8\_{STATUS,ADDR,MISC}.

**16.7 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR D FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK**

The Intel® Xeon® processor D family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily\_DisplaySignature 06\_56H. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32\_MC4, Table 16-24 lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS. Incremental error codes for the memory controller unit is reported in the register banks IA32\_MC9-IA32\_MC10. Table 16-18 lists model-specific error codes apply to IA32\_MCi\_STATUS, i = 9-10.

**16.7.1 Internal Machine Check Errors**

**Table 16-24. Machine Check Error Codes for IA32\_MC4\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
MCACOD <sup>2</sup>	15:0	internal Errors	0402h - PCU internal Errors 0403h - internal Errors 0406h - Intel TXT Errors 0407h - Other UBOX internal Errors. On an IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS (After a 3-strike, the core MCA banks will be unavailable).
Model specific errors	19:16	Reserved except for the following	0000b - No Error 00x1b - PCU internal error 001xb - PCU internal error

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved except for the following	x1xxb - UBOX error
	31:24	Reserved except for the following	00h - No Error 09h - MC_MESSAGE_CHANNEL_TIMEOUT 13h - MC_DMI_TRAINING_TIMEOUT 15h - MC_DMI_CPU_RESET_ACK_TIMEOUT 1Eh - MC_VR_ICC_MAX_LT_FUSED_ICC_MAX 25h - MC_SVID_COMMAND_TIMEOUT 26h - MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT 29h - MC_VR_VOUT_MAC_LT_FUSED_SVID 2Bh - MC_PKGC_WATCHDOG_HANG_CBZ_DOWN 2Ch - MC_PKGC_WATCHDOG_HANG_CBZ_UP 44h - MC_CRITICAL_VR_FAILED 46h - MC_VID_RAMP_DOWN_FAILED 49h - MC_SVID_WRITE_REG_VOUT_MAX_FAILED 4Bh - MC_PP1_BOOT_VID_TIMEOUT. Timeout setting boot VID for DRAM 0. 4Fh - MC_SVID_COMMAND_ERROR. 52h - MC_FIVR_CATAS_OVERVOL_FAULT. 53h - MC_FIVR_CATAS_OVERCUR_FAULT. 57h - MC_SVID_PKGC_REQUEST_FAILED 58h - MC_SVID_IMON_REQUEST_FAILED 59h - MC_SVID_ALERT_REQUEST_FAILED 62h - MC_INVALID_PKGS_RSP_QPI 64h - MC_INVALID_PKG_STATE_CONFIG 67h - MC_HA_IMC_Rw_BLOCK_ACK_TIMEOUT 6Ah - MC_MSGCH_PMREQ_CMP_TIMEOUT 72h - MC_WATCHDG_TIMEOUT_PKGS_MASTER 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.
2. The internal error codes may be model-specific.

### 16.7.2 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC9\_STATUS-IA32\_MC10\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture,").

MSR\_ERROR\_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32\_MCi\_STATUS and IA32\_MCi\_MISC; (i = 9-10).

**Table 16-25. Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 9-10)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - DDR3 address parity error
			0002H - Uncorrected HA write data error
			0004H - Uncorrected HA data byte enable error
			0008H - Corrected patrol scrub error
			0010H - Uncorrected patrol scrub error
			0100H - iMC, write data buffer parity errors
			0200H - DDR4 command address parity error
	36:32	Other info	Reserved
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

## 16.8 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V4 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v4 family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily\_DisplaySignature 06\_4FH. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32\_MC4, Table 16-20 in Section 16.6.1 lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS.

Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32\_MC5, IA32\_MC20, and IA32\_MC21. Information listed in Table 16-21 of Section 16.6.1 covers QPI MC error codes.

### 16.8.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC9\_STATUS-IA32\_MC16\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

Table 16-26 lists model-specific error codes apply to IA32\_MCi\_STATUS, i = 9-16.

IA32\_MCi\_STATUS (i=9-12) log errors from the first memory controller. The second memory controller logs into IA32\_MCi\_STATUS (i=13-16).

**Table 16-26. Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 9-16)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - DDR3 address parity error 0002H - Uncorrected HA write data error 0004H - Uncorrected HA data byte enable error 0008H - Corrected patrol scrub error 0010H - Uncorrected patrol scrub error 0020H - Corrected spare error 0040H - Uncorrected spare error 0100H - iMC, write data buffer parity errors 0200H - DDR4 command address parity error
	36:32	Other info	Reserved
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.8.2 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the MSRs IA32\_MC7\_MISC and IA32\_MC8\_MISC. Table 16-27 lists model-specific error codes apply to IA32\_MCi\_MISC, i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32\_MC7\_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32\_MC8\_{STATUS,ADDR,MISC}.

**Table 16-27. Intel HA MC Error Codes for IA32\_MCi\_MISC (i= 7, 8)**

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 15-8.
8:6	Address Mode	See Table 15-3.
40:9	Reserved	Reserved
41	Failover	Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.
42	Mirrorcorr	Error was corrected by mirroring and primary channel scrubbed successfully.
63:43	Reserved	Reserved

## 16.9 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR SCALABLE FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the Intel® Xeon® Processor Scalable Family with CPUID DisplayFamily\_DisplaySignature 06\_55H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32\_MC4. Table 16-28 in Section 16.9.1 lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS.



## 16.9.1 Internal Machine Check Errors

**Table 16-28. Machine Check Error Codes for IA32\_MC4\_STATUS**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
MCACOD <sup>2</sup>	15:0	Internal Errors	0402h - PCU internal Errors 0403h - PCU internal Errors 0406h - Intel TXT Errors 0407h - Other UBOX internal Errors.  On an IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS (After a 3-strike, the core MCA banks will be unavailable).
Model specific errors	19:16	Reserved except for the following	0000b - No Error 00xxb - PCU internal error
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 0Dh - MCA_DMI_TRAINING_TIMEOUT 0Fh - MCA_DMI_CPU_RESET_ACK_TIMEOUT 10h - MCA_MORE_THAN_ONE_LT_AGENT 1Eh - MCA_BIOS_RST_CPL_INVALID_SEQ 1Fh - MCA_BIOS_INVALID_PKG_STATE_CONFIG 25h - MCA_MESSAGE_CHANNEL_TIMEOUT 27h - MCA_MSGCH_PMREQ_CMP_TIMEOUT 30h - MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT 31h - MCA_PKGC_INVALID_RSP_PCH 33h - MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 34h - MCA_PKGC_WATCHDOG_HANG_CBZ_UP 38h - MCA_PKGC_WATCHDOG_HANG_C3_UP_SF 40h - MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 41h - MCA_SVID_COMMAND_TIMEOUT 42h - MCA_SVID_VCCIN_VR_VOUT_MAX_FAILURE 43h - MCA_SVID_CPU_VR_CAPABILITY_ERROR 44h - MCA_SVID_CRITICAL_VR_FAILED 45h - MCA_SVID_SA_ITD_ERROR 46h - MCA_SVID_READ_REG_FAILED 47h - MCA_SVID_WRITE_REG_FAILED 48h - MCA_SVID_PKGC_INIT_FAILED

Type	Bit No.	Bit Function	Bit Description
			49h - MCA_SVID_PKG_CONFIG_FAILED 4Ah - MCA_SVID_PKG_REQUEST_FAILED 4Bh - MCA_SVID_IMON_REQUEST_FAILED 4Ch - MCA_SVID_ALERT_REQUEST_FAILED 4Dh - MCA_SVID_MCP_VP_ABSENT_OR_RAMP_ERROR 4Eh - MCA_SVID_UNEXPECTED_MCP_VP_DETECTED 51h - MCA_FIVR_CATA_OVERVOL_FAULT 52h - MCA_FIVR_CATA_OVERCUR_FAULT 58h - MCA_WATCHDG_TIMEOUT_PKG_SECONDARY 59h - MCA_WATCHDG_TIMEOUT_PKG_MAIN 5Ah - MCA_WATCHDG_TIMEOUT_PKGS_MAIN 61h - MCA_PKGS_CPD_UNPCD_TIMEOUT 63h - MCA_PKGS_INVALID_REQ_PCH 64h - MCA_PKGS_INVALID_REQ_INTERNAL 65h - MCA_PKGS_INVALID_RSP_INTERNAL 6Bh - MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	52:32	Reserved	Reserved
	54:53	CORR_ERR_STATUS	Reserved
	56:55	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.
2. The internal error codes may be model-specific.

### 16.9.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the MSRs IA32\_MC5\_STATUS, IA32\_MC12\_STATUS, IA32\_MC19\_STATUS. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL (see Chapter 15, "Machine-Check Architecture").

Table 16-29 lists model-specific fields to interpret error codes applicable to IA32\_MCi\_STATUS, i= 5, 12, 19.

**Table 16-29. Interconnect MC Error Codes for IA32\_MCi\_STATUS, i = 5, 12, 19**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: - 0x0COF - Unsupported/Undefined Packet - 0x0EOF - For all other corrected and uncorrected errors

Type	Bit No.	Bit Function	Bit Description
Model specific errors	21:16	MSCOD	<p>The encoding of Uncorrectable (UC) errors are:</p> <p>00h - UC Phy Initialization Failure.</p> <p>01h - UC Phy detected drift buffer alarm.</p> <p>02h - UC Phy detected latency buffer rollover.</p> <p>10h - UC link layer Rx detected CRC error: unsuccessful LLR entered abort state</p> <p>11h - UC LL Rx unsupported or undefined packet.</p> <p>12h - UC LL or Phy control error.</p> <p>13h - UC LL Rx parameter exchange exception.</p> <p>1fh - UC LL detected control error from the link-mesh interface</p> <p>The encoding of correctable (COR) errors are:</p> <p>20h - COR Phy initialization abort</p> <p>21h - COR Phy reset</p> <p>22h - COR Phy lane failure, recovery in x8 width.</p> <p>23h - COR Phy LOc error corrected without Phy reset</p> <p>24h - COR Phy LOc error triggering Phy reset</p> <p>25h - COR Phy LOp exit error corrected with Phy reset</p> <p>30h - COR LL Rx detected CRC error - successful LLR without Phy re-init.</p> <p>31h - COR LL Rx detected CRC error - successful LLR with Phy re-init.</p> <p>All other values are reserved.</p>
	31:22	MSCOD_SPARE	<p>The definition below applies to MSCOD 12h (UC LL or Phy Control Errors)</p> <p>[Bit 22] : Phy Control Error</p> <p>[Bit 23] : Unexpected Retry.Ack flit</p> <p>[Bit 24] : Unexpected Retry.Req flit</p> <p>[Bit 25] : RF parity error</p> <p>[Bit 26] : Routeback Table error</p> <p>[Bit 27] : unexpected Tx Protocol flit (EOP, Header or Data)</p> <p>[Bit 28] : Rx Header-or-Credit BGF credit overflow/underflow</p> <p>[Bit 29] : Link Layer Reset still in progress when Phy enters LO (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL.LinkLayerReset going back to 0).</p> <p>[Bit 30] : Link Layer reset initiated while protocol traffic not idle</p> <p>[Bit 31] : Link Layer Tx Parity Error</p>
	37:32	Reserved	Reserved
	52:38	Corrected Error Cnt	
	56:53	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.9.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC13\_STATUS-IA32\_MC18\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

IA32\_MCi\_STATUS (i=13,14,17) log errors from the first memory controller. The second memory controller logs into IA32\_MCi\_STATUS (i=15,16,18).

**Table 16-30. Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 13-18)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - Address parity error
			0002H - HA write data parity error
			0004H - HA write byte enable parity error
			0008H - Corrected patrol scrub error
			0010H - Uncorrected patrol scrub error
			0020H - Corrected spare error
			0040H - Uncorrected spare error
			0080H - Any HA read error
			0100H - WDB read parity error
			0200H - DDR4 command address parity error
			0400H - Uncorrected address parity error
			0800H - Unrecognized request type
			0801H - Read response to an invalid scoreboard entry
			0802H - Unexpected read response
			0803H - DDR4 completion to an invalid scoreboard entry
			0804H - Completion to an invalid scoreboard entry
			0805H - Completion FIFO overflow
			0806H - Correctable parity error
			0807H - Uncorrectable error
			0808H - Interrupt received while outstanding interrupt was not ACKed
			0809H - ERID FIFO overflow
			080aH - Error on Write credits
			080bH - Error on Read credits
			080cH - Scheduler error
			080dH - Error event
	36:32	Other info	MC logs the first error device. This is an encoded 5-bit value of the device.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

## 16.9.4 M2M Machine Check Errors

MC error codes associated with M2M are reported in the MSRs IA32\_MC7\_STATUS, IA32\_MC8\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture,”).

**Table 16-31. M2M MC Error Codes for IA32\_MCi\_STATUS (i= 7-8)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model specific errors	16	MscodDataRdErr	Logged an MC read data error
	17	Reserved	Reserved
	18	MscodPtlWrErr	Logged an MC partial write data error
	19	MscodFullWrErr	Logged a full write data error
	20	MscodBgfErr	Logged an M2M clock-domain-crossing buffer (BGF) error
	21	MscodTimeOut	Logged an M2M time out
	22	MscodParErr	Logged an M2M tracker parity error
	23	MscodBucket1Err	Logged a fatal Bucket1 error
	31:24	Reserved	Reserved
	36:32	Other info	MC logs the first error device. This is an encoded 5-bit value of the device.
37	Reserved	Reserved	
56:38		See Chapter 15, “Machine-Check Architecture,”	
Status register validity indicators <sup>1</sup>	63:57		

### NOTES:

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

## 16.9.5 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the MSRs IA32\_MC7\_MISC and IA32\_MC8\_MISC. Table 16-32 lists model-specific error codes apply to IA32\_MCi\_MISC, i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32\_MC7\_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32\_MC8\_{STATUS,ADDR,MISC}.

**Table 16-32. Intel HA MC Error Codes for IA32\_MCi\_MISC (i= 7, 8)**

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 15-8.
8:6	Address Mode	See Table 15-3.
40:9	Reserved	Reserved
61:41	Reserved	Reserved
62	Mirrorcorr	Error was corrected by mirroring and primary channel scrubbed successfully.
63	Failover	Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.

## 16.10 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY\_DISPLAYMODEL SIGNATURE 06\_5FH, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel® Atom® processors based on Goldmont Microarchitecture with CPUID DisplayFamily\_DisplaySignature 06\_5FH (code name Denverton), incremental error codes for the memory controller unit are reported in the register banks IA32\_MC6 and IA32\_MC7. Table 16-33 in Section 16.10.1 lists model-specific fields to interpret error codes applicable to IA32\_MCi\_STATUS, i = 6, 7.

### 16.10.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC6\_STATUS and IA32\_MC7\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

**Table 16-33. Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 6, 7)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	
Model specific errors	31:16	Reserved except for the following	01h - Cmd/Addr parity 02h - Corrected Demand/Patrol Scrub Error 04h - Uncorrected patrol scrub error 08h - Uncorrected demand read error 10h - WDB read ECC
	36:32	Other info	
	37	Reserved	
	56:38		See Chapter 15, “Machine-Check Architecture”.
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

## 16.11 INCREMENTAL DECODING INFORMATION: FUTURE INTEL® XEON® PROCESSORS WITH CPUID DISPLAYFAMILY\_DISPLAYMODEL SIGNATURES 06\_6AH AND 06\_6CH, MACHINE ERROR CODES FOR MACHINE CHECK

Future Intel® Xeon® processor with CPUID DisplayFamily\_DisplaySignature of 06\_6AH and 06\_6CH, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32\_MC4. Table 16-34 in Section 16.11.1 lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS.

### 16.11.1 Internal Machine Check Errors

**Table 16-34. Machine Check Error Codes for IA32\_MC4\_STATUS**

Type	Bit No.	Bit Function	Bit Description
Machine Check Error Codes <sup>1</sup>	15:0	MCCOD	
MCCOD	15:0	Internal Errors	The value of this field will be 0402h for the PCU and 0406h for internal firmware errors. This applies for any logged error.
Model specific errors	19:16	Reserved except for the following	Model specific error code bits 19:16. This logs the type of HW UC (PCU/VCU) error that has occurred. There are 7 errors defined. 01h - Instruction address out of valid space. 02h - Double bit RAM error on Instruction Fetch. 03h - Invalid OpCode seen. 04h - Stack Underflow. 05h - Stack Overflow. 06h - Data address out of valid space. 07h - Double bit RAM error on Data Fetch.
	23:20	Reserved except for the following	Model specific error code bits 23:20. This logs the type of HW FSM error that has occurred. There are 3 errors defined. 04h - Clock/power IP response timeout. 05h - SMBus controller raised SMI. 09h - PM controller received invalid transaction.
	31:24	Reserved except for the following	0Dh - MCA_LLC_BIST_ACTIVE_TIMEOUT 0Eh - MCA_DMI_TRAINING_TIMEOUT 0Fh - MCA_DMI_STRAP_SET_ARRIVAL_TIMEOUT 10h - MCA_DMI_CPU_RESET_ACK_TIMEOUT 11h - MCA_MORE_THAN_ONE_LT_AGENT 14h - MCA_INCOMPATIBLE_PCH_TYPE 1Eh - MCA_BIOS_RST_CPL_INVALID_SEQ 1Fh - MCA_BIOS_INVALID_PKG_STATE_CONFIG 2Dh - MCA_PCU_PMAX_CALIB_ERROR 2Eh - MCA_TSC100_SYNC_TIMEOUT 3Ah - MCA_GPSB_TIMEOUT 3Bh - MCA_PMSB_TIMEOUT 3Eh - MCA_IOSFSB_PMREQ_CMP_TIMEOUT 40h - MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 42h - MCA_SVID_VCCIN_VR_VOUT_FAILURE 43h - MCA_SVID_CPU_VR_CAPABILITY_ERROR 44h - MCA_SVID_CRITICAL_VR_FAILED 45h - MCA_SVID_SA_ITD_ERROR 46h - MCA_SVID_READ_REG_FAILED 47h - MCA_SVID_WRITE_REG_FAILED 4Ah - MCA_SVID_PKGC_REQUEST_FAILED

Type	Bit No.	Bit Function	Bit Description
			48h - MCA_SVID_IMON_REQUEST_FAILED 4Ch - MCA_SVID_ALERT_REQUEST_FAILED 4Dh - MCA_SVID_MCP_VR_RAMP_ERROR 56h - MCA_FIVR_PD_HARDERR 58h - MCA_WATCHDOG_TIMEOUT_PKGC_SECONDARY 59h - MCA_WATCHDOG_TIMEOUT_PKGC_MAIN 5Ah - MCA_WATCHDOG_TIMEOUT_PKGS_MAIN 5Bh - MCA_WATCHDOG_TIMEOUT_MSG_CH_FSM 5Ch - MCA_WATCHDOG_TIMEOUT_BULK_CR_FSM 5Dh - MCA_WATCHDOG_TIMEOUT_IOSFSB_FSM 60h - MCA_PKGS_SAFE_WP_TIMEOUT 61h - MCA_PKGS_CPD_UNCPD_TIMEOUT 62h - MCA_PKGS_INVALID_REQ_PCH 63h - MCA_PKGS_INVALID_REQ_INTERNAL 64h - MCA_PKGS_INVALID_RSP_INTERNAL 65h-7Ah - MCA_PKGS_RESET_PREP_TIMEOUT 7Bh - MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT 7Ch - MCA_PKGS_SMBUS_MCP_PAUSE_TIMEOUT 7Dh - MCA_PKGS_SMBUS_SPD_PAUSE_TIMEOUT 80h - MCA_PKGC_DISP_BUSY_TIMEOUT 81h - MCA_PKGC_INVALID_RSP_PCH 83h - MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 84h - MCA_PKGC_WATCHDOG_HANG_CBZ_UP 87h - MCA_PKGC_WATCHDOG_HANG_C2_BLKMASTER 88h - MCA_PKGC_WATCHDOG_HANG_C2_PSLIMIT 89h - MCA_PKGC_WATCHDOG_HANG_SETDISP 8Bh - MCA_PKGC_ALLOW_L1_ERROR 90h - MCA_RECOVERABLE_DIE_THERMAL_TOO_HOT A0h - MCA_ADR_SIGNAL_TIMEOUT A1h - MCA_BCLK_FREQ_OC_ABOVE_THRESHOLD B0h - MCA_DISPATCHER_RUN_BUSY_TIMEOUT
	37:32	ENH_MCA_AVAIL0	Available when Enhanced MCA is in use.
	52:38	CORR_ERR_COUNT	Correctable error count.
	54:53	CORRERRORSTATUSIND	These bits are used to indicate when the number of corrected errors has exceeded the safe threshold to the point where an uncorrected error has become more likely to happen. Table 3 shows the encoding of these bits.
	56:55	ENH_MCA_AVAIL1	Available when Enhanced MCA is in use.
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.



### 16.11.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the MSRs IA32\_MC5\_STATUS, IA32\_MC7\_STATUS, IA32\_MC8\_STATUS. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL (see Chapter 15, "Machine-Check Architecture").

#### NOTE

The interconnect machine check errors in this section apply only to future Intel Xeon processors with a CPUID DisplayFamily\_DisplaySignature of 06\_6AH. These do not apply to future Intel Xeon processors with a CPUID DisplayFamily\_DisplaySignature of 06\_6CH.

Table 16-35 lists model-specific fields to interpret error codes applicable to IA32\_MCi\_STATUS, i= 5, 7, 8.

**Table 16-35. Interconnect MC Error Codes for IA32\_MCi\_STATUS, i = 5, 7, 8**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: - 0x0COF - Unsupported/Undefined Packet. - 0x0EOF - For all other corrected and uncorrected errors.
Model specific errors	21:16	MSCOD	The encoding of Uncorrectable (UC) errors are: 00h - Phy Initialization Failure (NumInit). 01h - Phy Detected Drift Buffer Alarm. 02h - Phy Detected Latency Buffer Rollover. 10h - LL Rx detected CRC error: unsuccessful LLR (entered Abort state). 11h - LL Rx Unsupported/Undefined packet. 12h - LL or Phy Control Error. 13h - LL Rx Parameter Exception. 1Fh - LL Detected Control Error. The encoding of correctable (COR) errors are: 20h - Phy Initialization Abort. 21h - Phy Inband Reset. 22h - Phy Lane failure, recovery in x8 width. 23h - Phy LOc error corrected without Phy reset. 24h - Phy LOc error triggering Phy reset. 25h - Phy LOp exit error corrected with reset. 30h - LL Rx detected CRC error: successful LLR without Phy Reinit. 31h - LL Rx detected CRC error: successful LLR with Phy Reinit. 32h - Tx received LLR. All other values are reserved.

Type	Bit No.	Bit Function	Bit Description
	31:22	MSCOD_SPARE	The definition below applies to MSCOD 12h (UC LL or Phy Control Errors). [Bit 22] : Phy Control Error. [Bit 23] : Unexpected Retry.Ack flit. [Bit 24] : Unexpected Retry.Req flit. [Bit 25] : RF parity error. [Bit 26] : Routeback Table error. [Bit 27] : Unexpected Tx Protocol flit (EOP, Header or Data). [Bit 28] : Rx Header-or-Credit BGF credit overflow/underflow. [Bit 29] : Link Layer Reset still in progress when Phy enters L0 (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL.LinkLayerReset going back to 0). [Bit 30] : Link Layer reset initiated while protocol traffic not idle. [Bit 31] : Link Layer Tx Parity Error.
	37:32	OTHER_INFO	Other Info.
	56:38	Corrected Error Cnt	See Chapter 15, "Machine-Check Architecture".
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

### 16.11.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers for the 3rd generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture are defined in Table 16-37.

The MSRs reporting MC error codes differ depending on the CPUID DisplayFamily\_DisplaySignature of the processor. See Table 16-36 for details.

**Table 16-36. MSRs Reporting MC Error Codes by CPUID DisplayFamily\_DisplaySignature**

Processor	CPUID DisplayFamily_DisplaySignature	MSRs Reporting MC Error Codes
3rd generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture	06_6AH	IA32_MC13_STATUS - IA32_MC15_STATUS IA32_MC17_STATUS - IA32_MC19_STATUS IA32_MC21_STATUS - IA32_MC23_STATUS IA32_MC25_STATUS - IA32_MC27_STATUS
3rd generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture	06_6CH	IA32_MC13_STATUS - IA32_MC15_STATUS IA32_MC17_STATUS - IA32_MC19_STATUS

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

**Table 16-37. Intel IMC MC Error Codes for IA32\_MCI\_STATUS (i= 13-15, 17-19, 21-23, 25-27)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - Address parity error. 0002H - Data parity error. 0003H - Data ECC error. 0004H - Data byte enable parity error. 0007H - Transaction ID parity error. 0008H - Corrected patrol scrub error. 0010H - Uncorrected patrol scrub error. 0020H - Corrected spare error. 0040H - Uncorrected spare error. 0080H - Corrected read error. 00A0H - Uncorrected read error. 00C0H - Uncorrected metadata. 0100H - WDB read parity error. 0103H - RPA parity error. 0104H - RPA parity error. 0105H - WPA parity error. 0106H - DDR_T_DPPP data BE error. 0107H - DDR_T_DPPP data error. 0108H - DDR link failure. 0111H - PCLS CAM error. 0112H - PCLS data error. 0200H - DDR4 command / address parity error. 0220H - HBM command / address parity error. 0221H - HBM data parity error. 0400H - RPQ parity (primary) error. 0401H - RPQ parity (buddy) error. 0404H - WPQ parity (primary) error. 0405H - WPQ parity (buddy) error. 0408H - RPB parity (primary) error. 0409H - RPB parity (buddy) error. 0800H - DDR-T bad request. 0801H - DDR Data response to an invalid entry. 0802H - DDR data response to an entry not expecting data. 0803H - DDR4 completion to an invalid entry. 0804H - DDR-T completion to an invalid entry. 0805H - DDR data/completion FIFO overflow. 0806H - DDR-T ERID correctable parity error. 0807H - DDR-T ERID uncorrectable error. 0808H - DDR-T interrupt received while outstanding interrupt was not ACKed.

INTERPRETING MACHINE-CHECK ERROR CODES

Type	Bit No.	Bit Function	Bit Description
			0809H - ERID FIFO overflow. 080AH - DDR-T error on FNV write credits. 080BH - DDR-T error on FNV read credits. 080CH - DDR-T scheduler error. 080DH - DDR-T FNV error event. 080EH - DDR-T FNV thermal event. 080FH - CMI packet while idle. 0810H - DDR_T_RPQ_REQ_PARITY_ERR. 0811H - DDR_T_WPQ_REQ_PARITY_ERR. 0812H - 2LM_NMFILLWR_CAM_ERR. 0813H - CMI_CREDIT_OVERSUB_ERR. 0814H - CMI_CREDIT_TOTAL_ERR. 0815H - CMI_CREDIT_RSVD_POOL_ERR. 0816H - DDR_T_RD_ERROR. 0817H - WDB_FIFO_ERR. 0818H - CMI_REQ_FIFO_OVERFLOW. 0819H - CMI_REQ_FIFO_UNDERFLOW. 081AH - CMI_RSP_FIFO_OVERFLOW. 081BH - CMI_RSP_FIFO_UNDERFLOW. 081CH - CMI_MISC_MC_CRDT_ERRORS. 081DH - CMI_MISC_MC_ARB_ERRORS. 081EH - DDR_T_WR_CMPL_FIFO_OVERFLOW. 081FH - DDR_T_WR_CMPL_FIFO_UNDERFLOW. 0820H - CMI_RD_CPL_FIFO_OVERFLOW. 0821H - CMI_RD_CPL_FIFO_UNDERFLOW. 0822H - TME_KEY_PAR_ERR. 0823H - TME_CMI_MISC_ERR. 0824H - TME_CMI_OVFL_ERR. 0825H - TME_CMI_UFL_ERR. 0826H - TME_TEM_SECURE_ERR. 0827H - TME_UFILL_PAR_ERR.
	37:32	Other info	Other Info.
	56:38		See Chapter 15, "Machine-Check Architecture".
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

Additional information is reported in the MSRs IA32\_MC13\_MISC – IA32\_MC15\_MISC, IA32\_MC17\_MISC – IA32\_MC19\_MISC, IA32\_MC21\_MISC – IA32\_MC23\_MISC, and IA32\_MC25\_MISC – IA32\_MC27\_MISC. Table 16-38 lists the information reported in IA32\_MCi\_MISC, i = 13-15, 17-19, 21-23, and 25-27.

**Table 16-38. Additional Information Reported in IA32\_MCi\_MISC (i= 13-15, 17-19, 21-23, 25-27)**

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 15-8.
8:6	Address Mode	See Table 15-3.
18:9	Column	Component of sub-DIMM address. Bits 18-17: Reserved Bit 16: Column 9 Bit 15: Column 8 Bit 14: Column 7 Bit 13: Column 6 Bit 12: Column 5 Bit 11: Column 4 Bit 10: Column 3 Bit 9: Reserved
39:19	Row	Component of sub-DIMM address.
45:40	Bank	Component of sub-DIMM address. Bit 45: Reserved Bit 44: Bank group 2 Bit 43: Bank address 1 Bit 42: Bank address 0 Bit 41: Bank group 1 Bit 40: Bank address 0
51:46	Failed Device	Failing device for correctable error (not valid for uncorrectable or transient errors).
55:52	SubRank	Logical Rank (3D Stacked SDRAM)
58:56	Rank	Rank
62:59	ECC Mode	0000b: SDDC memory mode 0001b: SDDC 0100b: ADDDC memory mode 0101b: ADDDC 1000b: Read from DDRT Other values: Reserved
63	Transient	0b: 1b: Error was transient

### 16.11.4 M2M Machine Check Errors

MC error codes associated with M2M for future Intel Xeon processors with a CPUID DisplayFamily\_DisplaySignature of 06\_6AH are reported in the MSRs IA32\_MC12\_STATUS, IA32\_MC16\_STATUS, IA32\_MC20\_STATUS, and IA32\_MC24\_STATUS.

MC error codes associated with M2M for future Intel Xeon processors with a CPUID DisplayFamily\_DisplaySignature of 06\_6AH are reported in the MSRs IA32\_MC12\_STATUS and IA32\_MC16\_STATUS.

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

**Table 16-39. M2M MC Error Codes for IA32\_MCi\_STATUS (i= 12, 16, 20, 24)**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model specific errors	16	MscodDataRdErr	Logged an MC read data error.
	17	Reserved	Reserved
	18	MscodPtlWrErr	Logged an MC partial write data error.
	19	MscodFullWrErr	Logged a full write data error.
	20	MscodBgfErr	Logged an M2M clock-domain-crossing buffer (BGF) error.
	21	MscodTimeOut	Logged an M2M time out.
	22	MscodParErr	Logged an M2M tracker parity error.
	23	MscodBucket1Err	Logged a fatal Bucket1 error.
	25:24	MscodDDRTYPE	Logged a DDR/DDR-T specific error.
	31:26	MscodMiscErrs	Logged a miscellaneous error.
37:32	Other info	Other Info.	
56:38		See Chapter 15, “Machine-Check Architecture”.	
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

MC error codes associated with mirrored memory corrections are reported in the MSRs IA32\_MC12\_MISC, IA32\_MC16\_MISC, IA32\_MC20\_MISC, and IA32\_MC24\_MISC. The model-specific error codes listed in Table 16-32 also apply to IA32\_MCi\_MISC, i = 12, 16, 20, 24.

## 16.12 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY\_DISPLAYMODEL SIGNATURE 06\_86H, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel® Atom® processors based on Tremont microarchitecture with CPUID DisplayFamily\_DisplaySignature 06\_86H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32\_MC4. Table 16-34 in Section 16.11.1 lists model-specific fields to interpret error codes applicable to IA32\_MC4\_STATUS.

### 16.12.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32\_MC13\_STATUS - IA32\_MC15\_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

The IA32\_MCi\_STATUS MSR (where i = 13, 14, 15) contains information related to a machine check error if its VAL(valid) flag is set. Bit definitions are the same as those found in Table 16-37 “Intel IMC MC Error Codes for IA32\_MCi\_STATUS (i= 13-15, 17-19, 21-23, 25-27)”.

The IA32\_MCi\_MISC MSR (where i = 13, 14, 15) contains information related memory corrections. Bit definitions are the same as those found in Table 16-38 “Additional Information Reported in IA32\_MCi\_MISC (i= 13-15, 17-19, 21-23, 25-27)”.

### 16.12.2 M2M Machine Check Errors

MC error codes associated with M2M are reported in the IA32\_MC12\_STATUS MSR. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

Bit definitions are the same as those found in Table 16-39 “M2M MC Error Codes for IA32\_MCi\_STATUS (i= 12, 16, 20, 24)”.

## 16.13 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 0FH MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-40 provides information for interpreting additional family 0FH model-specific fields for external bus errors. These errors are reported in the IA32\_MCi\_STATUS MSRs. They are reported architecturally as compound errors with a general form of 0000 1PPT RRRR IILL in the MCA error code field. See Chapter 15 for information on the interpretation of compound error codes.

**Table 16-40. Incremental Decoding Information: Processor Family 0FH Machine Error Codes For Machine Check**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0		
Model-specific error codes	16	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	17	Response hard fail	Hardware failure detected on response
	18	Response parity	Parity error detected on response
	19	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	20	Processor Signature = 00000F04H: Invalid PIC request  All other processors: Reserved	Processor Signature = 00000F04H. Indicates error due to an invalid PIC request access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No Invalid PIC request error Reserved
	21	Pad state machine	The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.
	22	Pad strobe glitch	Data strobe glitch
	23	Pad address glitch	Address strobe glitch
Other Information	56:24	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, “Machine-Check Architecture,” for more information.

Table 16-10 provides information on interpreting additional family 0FH, model specific fields for cache hierarchy errors. These errors are reported in one of the IA32\_MCi\_STATUS MSRs. These errors are reported, architecturally, as compound errors with a general form of 0000 0001 RRRR TTLL in the MCA error code field. See Chapter 15 for how to interpret the compound error code.

**16.13.1 Model-Specific Machine Check Error Codes for Intel Xeon Processor MP 7100 Series**

Intel Xeon processor MP 7100 series has 5 register banks which contains information related to Machine Check Errors. MCi\_STATUS[63:0] refers to all 5 register banks. MC0\_STATUS[63:0] through MC3\_STATUS[63:0] is the same as on previous generation of Intel Xeon processors within Family 0FH. MC4\_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

**Table 16-41. MCI\_STATUS Register Bit Definition**

Bit Field Name	Bits	Description
MCA_Error_Code	15:0	Specifies the machine check architecture defined error code for the machine check error condition detected. The machine check architecture defined error codes are guaranteed to be the same for all Intel Architecture processors that implement the machine check architecture. See tables below
Model_Specific_Error_Code	31:16	Specifies the model specific error code that uniquely identifies the machine check error condition detected. The model specific error codes may differ among Intel Architecture processors for the same Machine Check Error condition. See tables below
Other_Info	56:32	The functions of the bits in this field are implementation specific and are not part of the machine check architecture. Software that is intended to be portable among Intel Architecture processors should not rely on the values in this field.
PCC	57	Processor Context Corrupt flag indicates that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor’s state. This bit will always be set for MC errors which are not corrected.
ADDRV	58	MC_ADDR register valid flag indicates that the MC_ADDR register contains the address where the error occurred. When clear, this flag indicates that the MC_ADDR register does not contain the address where the error occurred. The MC_ADDR register should not be read if the ADDRv bit is clear.
MISCV	59	MC_MISC register valid flag indicates that the MC_MISC register contains additional information regarding the error. When clear, this flag indicates that the MC_MISC register does not contain additional information regarding the error. MC_MISC should not be read if the MISCV bit is not set.
EN	60	Error enabled flag indicates that reporting of the machine check exception for this error was enabled by the associated flag bit of the MC_CTL register. Note that correctable errors do not have associated enable bits in the MC_CTL register so the EN bit should be clear when a correctable error is logged.



**Table 16-41. MCI\_STATUS Register Bit Definition (Contd.)**

Bit Field Name	Bits	Description
UC	61	Error uncorrected flag indicates that the processor did not correct the error condition. When clear, this flag indicates that the processor was able to correct the event condition.
OVER	62	Machine check overflow flag indicates that a machine check error occurred while the results of a previous error were still in the register bank (i.e., the VAL bit was already set in the MC_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. Enabled errors are written over disabled errors, and uncorrected errors are written over corrected events. Uncorrected errors are not written over previous valid uncorrected errors.
VAL	63	MC_STATUS register valid flag indicates that the information within the MC_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the MC_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

### 16.13.1.1 Processor Machine Check Status Register MCA Error Code Definition

Intel Xeon processor MP 7100 series use compound MCA Error Codes for logging its CBC internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines additional Machine Check error types (IA32\_MC4\_STATUS[15:0]) beyond those defined in Chapter 15. Table 16-42 lists these model-specific MCA error codes. Error code details are specified in MC4\_STATUS [31:16] (see Section 16.13.3), the "Model Specific Error Code" field. The information in the "Other\_Info" field (MC4\_STATUS[56:32]) is common to the three processor error types and contains a correctable event count and specifies the MC4\_MISC register format.

**Table 16-42. Incremental MCA Error Code for Intel Xeon Processor MP 7100**

Processor MCA_Error_Code (MC4_STATUS[15:0])			
Type	Error Code	Binary Encoding	Meaning
C	Internal Error	0000 0100 0000 0000	Internal Error Type Code
A	L3 Tag Error	0000 0001 0000 1011	L3 Tag Error Type Code
B	Bus and Interconnect Error	0000 100x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 101x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 110x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 1110 0000 1111	Bus and Interconnection Error Type Code
		0000 1111 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations

The **Bold faced** binary encodings are the only encodings used by the processor for MC4\_STATUS[15:0].

### 16.13.2 Other\_Info Field (all MCA Error Types)

The MC4\_STATUS[56:32] field is common to the processor's three MCA error types (A, B & C).

**Table 16-43. Other Information Field Bit Definition**

Bit Field Name	Bits	Description
39:32	8-bit Correctable Event Count	Holds a count of the number of correctable events since cold reset. This is a saturating counter; the counter begins at 1 (with the first error) and saturates at a count of 255.
41:40	MC4_MISC format type	The value in this field specifies the format of information in the MC4_MISC register. Currently, only two values are defined. Valid only when MISCV is asserted.
43:42	-	Reserved
51:44	ECC syndrome	ECC syndrome value for a correctable ECC event when the "Valid ECC syndrome" bit is asserted
52	Valid ECC syndrome	Set when correctable ECC event supplies the ECC syndrome
54:53	Threshold-Based Error Status	00: No tracking - No hardware status tracking is provided for the structure reporting this event. 01: Green - Status tracking is provided for the structure posting the event; the current status is green (below threshold). 10: Yellow - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). 11: Reserved for future use  Valid only if Valid bit (bit 63) is set Undefined if the UC bit (bit 61) is set
56:55	-	Reserved

### 16.13.3 Processor Model Specific Error Code Field

#### 16.13.3.1 MCA Error Type A: L3 Error

Note: The Model Specific Error Code field in MC4\_STATUS (bits 31:16).

**Table 16-44. Type A: L3 Error Codes**

Bit Num	Sub-Field Name	Description	Legal Value(s)
18:16	L3 Error Code	Describes the L3 error encountered	000 - No error 001 - More than one way reporting a correctable event 010 - More than one way reporting an uncorrectable error 011 - More than one way reporting a tag hit 100 - No error 101 - One way reporting a correctable event 110 - One way reporting an uncorrectable error 111 - One or more ways reporting a correctable event while one or more ways are reporting an uncorrectable error
20:19	-	Reserved	00
31:21	-	Fixed pattern	0010_0000_000

### 16.13.3.2 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error

Note: The Model Specific Error Code field in MC4\_STATUS (bits 31:16).

**Table 16-45. Type B Bus and Interconnect Error Codes**

Bit Num	Sub-Field Name	Description
16	FSB Request Parity	Parity error detected during FSB request phase
17	Core0 Addr Parity	Parity error detected on Core 0 request's address field
18	Core1 Addr Parity	Parity error detected on Core 1 request's address field
19		Reserved
20	FSB Response Parity	Parity error on FSB response field detected
21	FSB Data Parity	FSB data parity error on inbound data detected
22	Core0 Data Parity	Data parity error on data received from Core 0 detected
23	Core1 Data Parity	Data parity error on data received from Core 1 detected
24	IDS Parity	Detected an Enhanced Defer parity error (phase A or phase B)
25	FSB Inbound Data ECC	Data ECC event to error on inbound data (correctable or uncorrectable)
26	FSB Data Glitch	Pad logic detected a data strobe 'glitch' (or sequencing error)
27	FSB Address Glitch	Pad logic detected a request strobe 'glitch' (or sequencing error)
31:28	---	Reserved

Exactly one of the bits defined in the preceding table will be set for a Bus and Interconnect Error. The Data ECC can be correctable or uncorrectable (the MC4\_STATUS.UC bit, of course, distinguishes between correctable and uncorrectable cases with the Other\_Info field possibly providing the ECC Syndrome for correctable errors). All other errors for this processor MCA Error Type are uncorrectable.

### 16.13.3.3 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error

**Table 16-46. Type C Cache Bus Controller Error Codes**

MC4_STATUS[31:16] (MSCE) Value	Error Description
0000_0000_0000_0001 0001H	Inclusion Error from Core 0
0000_0000_0000_0010 0002H	Inclusion Error from Core 1
0000_0000_0000_0011 0003H	Write Exclusive Error from Core 0
0000_0000_0000_0100 0004H	Write Exclusive Error from Core 1
0000_0000_0000_0101 0005H	Inclusion Error from FSB
0000_0000_0000_0110 0006H	SNP Stall Error from FSB
0000_0000_0000_0111 0007H	Write Stall Error from FSB
0000_0000_0000_1000 0008H	FSB Arb Timeout Error
0000_0000_0000_1001 0009H	CBC OOD Queue Underflow/overflow
0000_0001_0000_0000 0100H	Enhanced Intel SpeedStep Technology TM1-TM2 Error
0000_0010_0000_0000 0200H	Internal Timeout error
0000_0011_0000_0000 0300H	Internal Timeout Error
0000_0100_0000_0000 0400H	Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow

**Table 16-46. Type C Cache Bus Controller Error Codes (Contd.)**

MC4_STATUS[31:16] (MSCE) Value	Error Description
1100_0000_0000_0001 C001H	Correctable ECC event on outgoing FSB data
1100_0000_0000_0010 C002H	Correctable ECC event on outgoing Core 0 data
1100_0000_0000_0100 C004H	Correctable ECC event on outgoing Core 1 data
1110_0000_0000_0001 E001H	Uncorrectable ECC error on outgoing FSB data
1110_0000_0000_0010 E002H	Uncorrectable ECC error on outgoing Core 0 data
1110_0000_0000_0100 E004H	Uncorrectable ECC error on outgoing Core 1 data
— all other encodings —	Reserved

All errors - except for the correctable ECC types - in this table are uncorrectable. The correctable ECC events may supply the ECC syndrome in the Other\_Info field of the MC4\_STATUS MSR.

**Table 16-47. Decoding Family 0FH Machine Check Codes for Cache Hierarchy Errors**

Type	Bit No.	Bit Function	Bit Description
MCA error codes <sup>1</sup>	15:0		
Model specific error codes	17:16	Tag Error Code	Contains the tag error code for this machine check error: 00 = No error detected 01 = Parity error on tag miss with a clean line 10 = Parity error/multiple tag match on tag hit 11 = Parity error/multiple tag match on tag miss
	19:18	Data Error Code	Contains the data error code for this machine check error: 00 = No error detected 01 = Single bit error 10 = Double bit error on a clean line 11 = Double bit error on a modified line
	20	L3 Error	This bit is set if the machine check error originated in the L3 it can be ignored for invalid PIC request errors): 1 = L3 error 0 = L2 error
	21	Invalid PIC Request	Indicates error due to invalid PIC request access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No invalid PIC request error
	31:22	Reserved	Reserved
Other Information	39:32	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 255.
	56:40	Reserved	Reserved
Status register validity indicators <sup>1</sup>	63:57		

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

## 20. Updates to Chapter 17, Volume 3B

Change bars and green text show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes to this chapter: Updated figure 17-2 to show the DR0 register (previously this portion was cut off in the figure). Clarification added to Section 17.18.7, "Monitoring Resource Selection and Reporting Infrastructure".

# CHAPTER 17

## DEBUG, BRANCH PROFILE, TSC, AND INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FEATURES

---

Intel 64 and IA-32 architectures provide debug facilities for use in debugging code and monitoring performance. These facilities are valuable for debugging application software, system software, and multitasking operating systems. Debug support is accessed using debug registers (DR0 through DR7) and model-specific registers (MSRs):

- Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.
- MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.
- Time stamp counter is described in Section 17.17, "Time-Stamp Counter".
- Features which allow monitoring of shared platform resources such as the L3 cache are described in Section 17.18, "Intel® Resource Director Technology (Intel® RDT) Monitoring Features".
- Features which enable control over shared platform resources are described in Section 17.19, "Intel® Resource Director Technology (Intel® RDT) Allocation Features".

### 17.1 OVERVIEW OF DEBUG SUPPORT FACILITIES

The following processor facilities support debugging and performance monitoring:

- **Debug exception (#DB)** — Transfers program control to a debug procedure or task when a debug event occurs.
- **Breakpoint exception (#BP)** — See breakpoint instruction (INT3) below.
- **Breakpoint-address registers (DR0 through DR3)** — Specifies the addresses of up to 4 breakpoints.
- **Debug status register (DR6)** — Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- **Debug control register (DR7)** — Specifies the forms of memory or I/O access that cause breakpoints to be generated.
- **T (trap) flag, TSS** — Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.
- **RF (resume) flag, EFLAGS register** — Suppresses multiple exceptions to the same instruction.
- **TF (trap) flag, EFLAGS register** — Generates a debug exception (#DB) after every execution of an instruction.
- **Breakpoint instruction (INT3)** — Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set instruction breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.

- Execution of the breakpoint instruction.
- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write to a specified memory address/range.
- Write to a specified memory address/range.
- Input from a specified I/O address/range.
- Output to a specified I/O address/range.
- Attempt to change the contents of a debug register.

## 17.2 DEBUG REGISTERS

Eight debug registers (see Figure 17-1 for 32-bit operation and Figure 17-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.

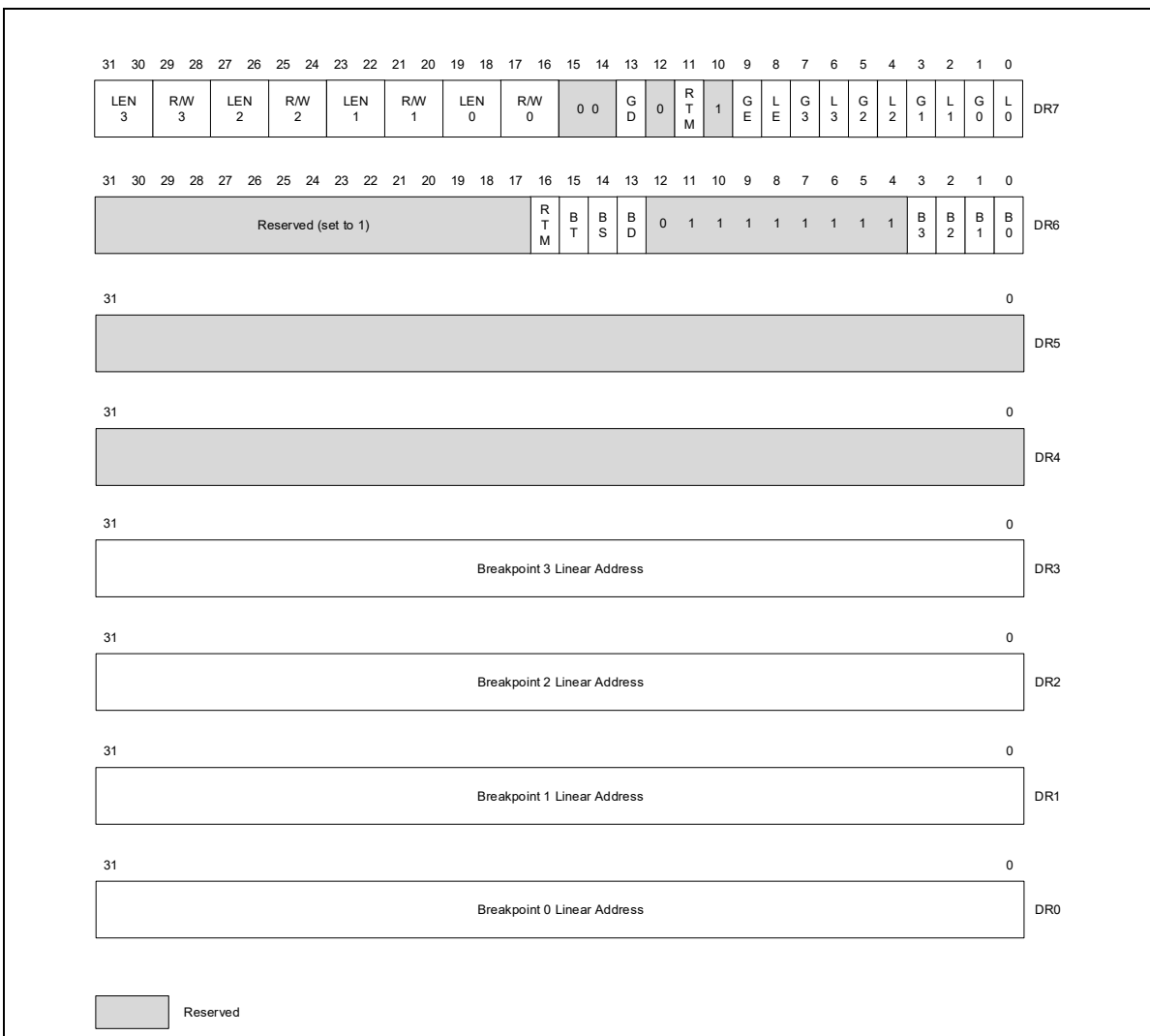


Figure 17-1. Debug Registers

Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 through 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 17.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

### 17.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 17-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

### 17.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

### 17.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 17-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3 (breakpoint condition detected) flags (bits 0 through 3)** — Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LEN<sub>n</sub>, and R/W<sub>n</sub> flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.
- **BD (debug register access detected) flag (bit 13)** — Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 17.2.4, “Debug Control Register (DR7),” for further explanation of the purpose of this flag.
- **BS (single step) flag (bit 14)** — Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.
- **BT (task switch) flag (bit 15)** — Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 7.2.1, “Task-State Segment (TSS),” for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.
- **RTM (restricted transactional memory) flag (bit 16)** — Indicates (when **clear**) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.



Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

## 17.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 17-1). The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6)** — Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated  $L_n$  flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7)** — Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated  $G_n$  flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.
- **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9)** — This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.
- **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 17.3.3). This advanced debugging is enabled only if IA32\_DEBUGCTL.RTM is also set.
- **GD (general detect enable) flag (bit 13)** — Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.

- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29)** — Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the  $R/W_n$  fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

- 00 — Break on instruction execution only.
- 01 — Break on data writes only.
- 10 — Break on I/O reads or writes.
- 11 — Break on data reads or writes but not instruction fetches.

When the DE flag is clear, the processor interprets the  $R/W_n$  bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

- 00 — Break on instruction execution only.
- 01 — Break on data writes only.
- 10 — Undefined.
- 11 — Break on data reads or writes but not instruction fetches.

- **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31)** — Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

- 00 — 1-byte length.
- 01 — 2-byte length.
- 10 — Undefined (or 8 byte length, see note below).
- 11 — 4-byte length.

If the corresponding  $RWn$  field in register DR7 is 00 (instruction execution), then the  $LENn$  field should also be 00. The effect of using other lengths is undefined. See Section 17.2.5, “Breakpoint Field Recognition,” below.

## NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), breakpoint conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the  $LENn$  field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). The Encoding 10B is supported in processors based on Intel® Atom™ microarchitecture, with CPUID signature of family 6, DisplayModel value 1CH. The encoding 10B is undefined for other processors.

### 17.2.5 Breakpoint Field Recognition

Breakpoint address registers (debug registers DR0 through DR3) and the  $LENn$  fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The  $LENn$  fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register ( $DRn$ ). Two-byte ranges must be aligned on word boundaries; 4-byte ranges must be aligned on doubleword boundaries, 8-byte ranges must be aligned on quadword boundaries. I/O addresses are zero-extended (from 16 to 32 bits, for comparison with the breakpoint address in the selected debug register). These requirements are enforced by the processor; it uses  $LENn$  field bits to mask the lower address bits in the debug registers. Unaligned data or I/O breakpoint addresses do not yield valid results.

A data breakpoint for reading or writing data is triggered if any of the bytes participating in an access is within the range defined by a breakpoint address register and its  $LENn$  field. Table 17-1 provides an example setup of debug registers and data accesses that would subsequently trap or not trap on the breakpoints.

A data breakpoint for an unaligned operand can be constructed using two breakpoints, where each breakpoint is byte-aligned and the two breakpoints together cover the operand. The breakpoints generate exceptions only for the operand, not for neighboring bytes.

Instruction breakpoint addresses must have a length specification of 1 byte (the  $LENn$  field is set to 00). Instruction breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

**Table 17-1. Breakpoint Examples**

Debug Register Setup			
Debug Register	R/Wn	Breakpoint Address	LENn
DR0	R/W0 = 11 (Read/Write)	A0001H	LEN0 = 00 (1 byte)
DR1	R/W1 = 01 (Write)	A0002H	LEN1 = 00 (1 byte)
DR2	R/W2 = 11 (Read/Write)	B0002H	LEN2 = 01) (2 bytes)
DR3	R/W3 = 01 (Write)	C0000H	LEN3 = 11 (4 bytes)
Data Accesses			
Operation		Address	Access Length (In Bytes)
Data operations that trap			
- Read or write		A0001H	1
- Read or write		A0001H	2
- Write		A0002H	1
- Write		A0002H	2
- Read or write		B0001H	4
- Read or write		B0002H	1
- Read or write		B0002H	2
- Write		C0000H	4
- Write		C0001H	2
- Write		C0003H	1
Data operations that do not trap			
- Read or write		A0000H	1
- Read		A0002H	1
- Read or write		A0003H	4
- Read or write		B0000H	2
- Read		C0000H	2
- Read or write		C0004H	4

### 17.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 17-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Break point conditions for 8-byte memory read/writes are supported in all modes.

## 17.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

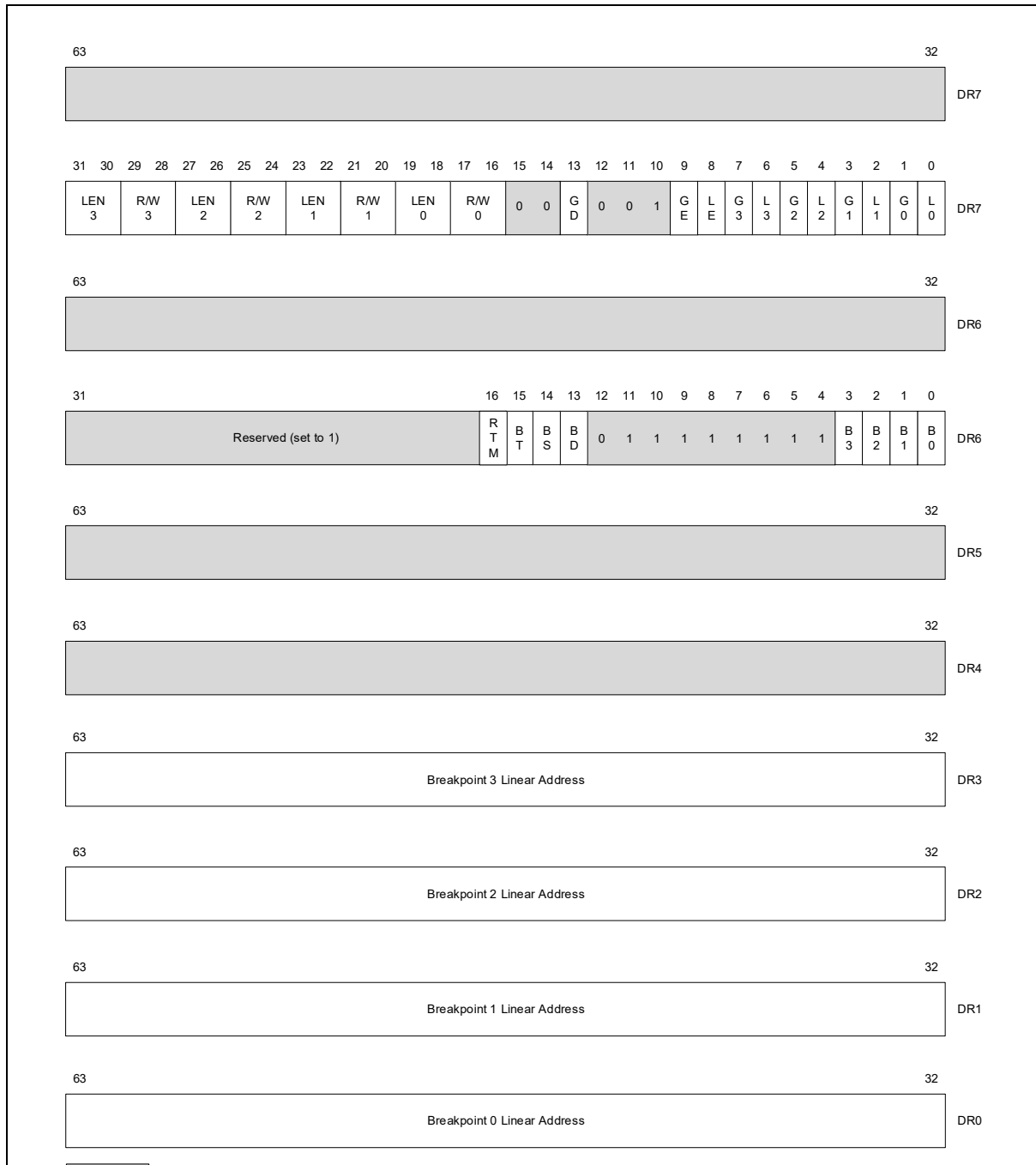


Figure 17-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

### 17.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 17-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 17.3.1.3, “General-Detect Exception Condition”) result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

The INT1 instruction generates a debug exception as a trap. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

See also: Chapter 6, “Interrupt 1—Debug Exception (#DB),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Table 17-2. Debug Exception Conditions**

Debug or Breakpoint Condition	DR6 Flags Tested	DR7 Flags Tested	Exception Class
Single-step trap	BS = 1		Trap
Instruction breakpoint, at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 0	Fault
Data write breakpoint, at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 1	Trap
I/O read or write breakpoint, at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 2	Trap
Data read or write (but not instruction fetches), at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 3	Trap
General detect fault, resulting from an attempt to modify debug registers (usually in conjunction with in-circuit emulation)	BD = 1	None	Fault
Task switch	BT = 1	None	Trap
INT1 instruction	None	None	Trap

### 17.3.1.1 Instruction-Breakpoint Exception Condition

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if an instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint will be suppressed as if EFLAGS.RF were 1 (see the next paragraph and Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, “System Flags and Fields in the EFLAGS Register,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for instruction breakpoints, CS limit violations, and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.

- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  - Debug exceptions generated in response to instruction breakpoints
  - Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)
  - Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)
  - Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

### 17.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

Data memory and I/O breakpoints are reported when the processor attempts to access a memory or I/O address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect data or I/O accesses (R/W flag is set to 1, 2, or 3). The processor generates the exception after it executes the instruction that made the access, so these breakpoint condition causes a trap-class exception to be generated.

Because data breakpoints are traps, an instruction that writes memory overwrites the original data before the debug exception generated by a data breakpoint is generated. If a debugger needs to save the contents of a write breakpoint location, it should save the original contents before setting the breakpoint. The handler can report the saved value after the breakpoint is triggered. The address in the debug registers can be used to locate the new value stored by the instruction that triggered the breakpoint.

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

Intel486 and later processors ignore the GE and LE flags in DR7. In Intel386 processors, exact data breakpoint matching does not occur unless it is enabled by setting the LE and/or the GE flags.

For repeated INS and OUTS instructions that generate an I/O-breakpoint debug exception, the processor generates the exception after the completion of the first iteration. Repeated INS and OUTS instructions generate a data-breakpoint debug exception after the iteration in which the memory address breakpoint location is accessed.

If an execution of the MOV or POP instruction loads the SS register and encounters a data breakpoint, the resulting debug exception is delivered after completion of the next instruction (the one after the MOV or POP).

Any pending data or I/O breakpoints are lost upon delivery of an exception. For example, if a machine-check exception (#MC) occurs following an instruction that encounters a data breakpoint (but before the resulting debug exception is delivered), the data breakpoint is lost. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, the data breakpoint is lost if the next instruction causes a fault.

Delivery of events due to INT *n*, INT3, or INTO does not cause a loss of data breakpoints. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, and the next instruction is software interrupt (INT *n*, INT3, or INTO), a debug exception (#DB) resulting from a data breakpoint will be delivered after the transition to the software-interrupt handler. The #DB handler should account for the fact that the #DB may have been delivered

after a invocation of a software-interrupt handler, and in particular that the CPL may have changed between recognition of the data breakpoint and delivery of the #DB.

### 17.3.1.3 General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required. The debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

### 17.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT *n*, INT3, and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT *n* instruction that calls the interrupt handler.

If an occurrence of the MOV or POP instruction loads the SS register executes with EFLAGS.TF = 1, no single-step debug exception occurs following the MOV or POP instruction.

### 17.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task's TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

## 17.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT3 instruction. See Chapter 6, "Interrupt 3—Breakpoint Exception (#BP)." Debuggers use breakpoint exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers, because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.



### 17.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify **transactional regions** (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a **transactional abort**. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable **advanced debugging of RTM transactional regions** by setting DR7.RTM[bit 11] and IA32\_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does **not** resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered. (A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

## 17.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™ 2 Duo, Intel® Core™ i7 and Intel® Atom™ processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt and exception recording:

- Section 17.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)”
- Section 17.6, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture”
- Section 17.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem”
- Section 17.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge”
- Section 17.11, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture”
- Section 17.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture”
- Section 17.14, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors)”
- Section 17.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)”
- Section 17.16, “Last Branch, Interrupt, and Exception Recording (P6 Family Processors)”

The following subsections of Section 17.4 describe common features of profiling branches. These features are generally enabled using the IA32\_DEBUGCTL MSR (older processor may have implemented a subset or model-specific features, see definitions of MSR\_DEBUGCTLA, MSR\_DEBUGCTLB, MSR\_DEBUGCTL).

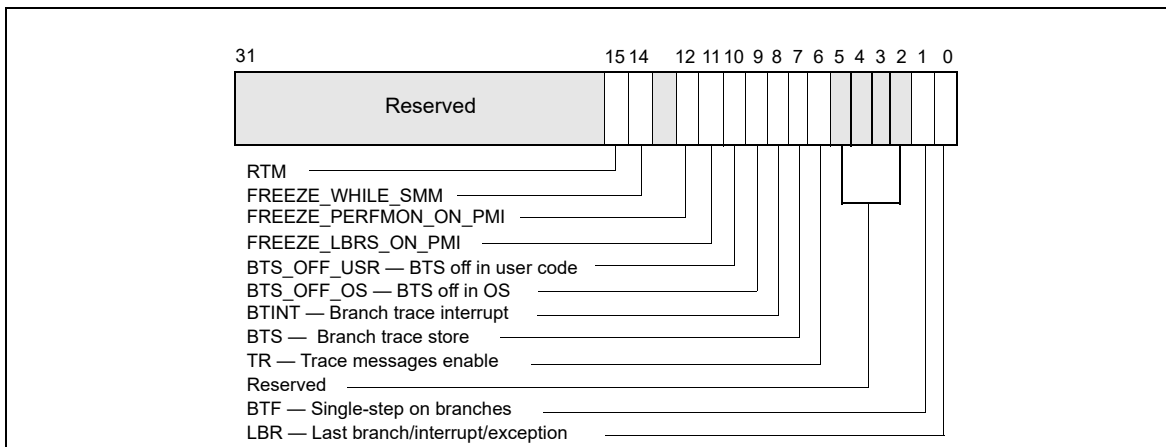


### 17.4.1 IA32\_DEBUGCTL MSR

The **IA32\_DEBUGCTL** MSR provides bit field controls to enable debug trace interrupts, debug trace stores, trace messages enable, single stepping on branches, last branch record recording, and to control freezing of LBR stack or performance counters on a PMI request. IA32\_DEBUGCTL MSR is located at register address 01D9H.

See Figure 17-3 for the MSR layout and the bullets below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the Section 17.5.1, “LBR Stack” (Intel® Core™2 Duo and Intel® Atom™ Processor Family) and Section 17.9.1, “LBR Stack” (processors based on Intel® Microarchitecture code name Nehalem).
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.
- **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages,” for more information about the TR flag.
- **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bit 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.



**Figure 17-3. IA32\_DEBUGCTL MSR for Processors based on Intel Core microarchitecture**

- **BTS\_OFF\_OS (branch trace off in privileged code) flag (bit 9)** — When set, BTS or BTM is skipped if CPL is 0. See Section 17.13.2.
- **BTS\_OFF\_USR (branch trace off in user code) flag (bit 10)** — When set, BTS or BTM is skipped if CPL is greater than 0. See Section 17.13.2.
- **FREEZE\_LBRS\_ON\_PMI flag (bit 11)** — When set, the LBR stack is frozen on a hardware PMI request (e.g. when a counter overflows and is configured to trigger PMI). See Section 17.4.7 for details.
- **FREEZE\_PERFMON\_ON\_PMI flag (bit 12)** — When set, the performance counters (IA32\_PMCx and IA32\_FIXED\_CTRx) are frozen on a PMI request. See Section 17.4.7 for details.
- **FREEZE\_WHILE\_SMM (bit 14)** — If this bit is set, upon the delivery of an SMI, the processor will clear all the enable bits of IA32\_PERF\_GLOBAL\_CTRL, save a copy of the content of IA32\_DEBUGCTL and disable LBR, BTF,

TR, and BTS fields of IA32\_DEBUGCTL before transferring control to the SMI handler. Subsequently, the enable bits of IA32\_PERF\_GLOBAL\_CTRL will be set to 1, the saved copy of IA32\_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its service. Note that system software must check if the processor supports the IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM control bit.

IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM is supported if IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is reporting 1. See Section 18.8 for details of detecting the presence of IA32\_PERF\_CAPABILITIES MSR.

- **RTM (bit 15)** — If this bit is set, advanced debugging of RTM transactional regions is enabled if DR7.RTM is also set. See Section 17.3.3.

## 17.4.2 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag (bit 0) in the IA32\_DEBUGCTL MSR is set, the processor automatically begins recording branch records for taken branches, interrupts, and exceptions (except for debug exceptions) in the LBR stack MSRs.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler. This action does not clear previously stored LBR stack MSRs.

A debugger can use the linear addresses in the LBR stack to re-set breakpoints in the breakpoint address registers (DR0 through DR3). This allows a backward trace from the manifestation of a particular bug toward its source.

On some processors, if the LBR flag is cleared and TR flag in the IA32\_DEBUGCTL MSR remains set, the processor will continue to update LBR stack MSRs. This is because those processors use the entries in the LBR stack in the process of generating BTM/BTS records. A #DB does not automatically clear the TR flag.

## 17.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32\_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.<sup>1</sup> This mechanism allows a debugger to single-step on control transfers caused by branches. This “branch single stepping” helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

## 17.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32\_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 17.4.2, “Monitoring Branches, Exceptions, and Interrupts.”

For P6 processor family, Pentium M processor family, processors based on Intel Core microarchitecture, TR and LBR bits can not be set at the same time due to hardware limitation. The content of LBR stack is undefined when TR is set.

For processors with Intel NetBurst microarchitecture, Intel Atom processors, and Intel Core and related Intel Xeon processors both starting with the Nehalem microarchitecture, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32\_DEBUGCTL MSR (or the equivalent MSR\_DEBUGCTLA, MSR\_DEBUGCTLB).

The following exception applies:

- BTM may not be observable on Intel Atom processor families that do not provide an externally visible system bus (i.e., processors based on the Silvermont microarchitecture or later).

1. Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 7.2.1, “Task-State Segment (TSS).”

### 17.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

### 17.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32\_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 17.4.9.2, "Setting Up the DS Save Area." for additional details.

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

### 17.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if `CPUID.01H:ECX[bit 4] = 1`.

The CPL-qualified branch trace mechanism is described in Section 17.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, `BTS_OFF_USR` (bit 10) and `BTS_OFF_OS` (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

### 17.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or greater (i.e. `CPUID.0AH:EAX[7:0] > 1`). These capabilities provides the following interface in IA32\_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

- **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by `CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0]`, two flavors are supported:
  - Legacy `Freeze_LBR_on_PMI` is supported for `ArchPerfMonVerID <= 3` and `ArchPerfMonVerID > 1`. If `IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1`, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32\_DEBUGCTL. Software must then re-enable `IA32_DEBUGCTL.LBR` to resume recording branches. When using this feature, software should be careful about writes to IA32\_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  - Streamlined `Freeze_LBR_on_PMI` is supported for `ArchPerfMonVerID >= 4`. If `IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1`, the processor behaves as follows:
    - sets `IA32_PERF_GLOBAL_STATUS.LBR_Frz = 1` to disable recording, but does not change the LBR bit (bit 0) in IA32\_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.
- **Freezing PMCs on PMI (bit 12)** — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by `CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0]`, two flavors are supported:

- Legacy Freeze\_Perfmon\_on\_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32\_DEBUGCTL.Freeze\_Perfmon\_On\_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32\_PERF\_GLOBAL\_CTRL MSR (see Figure 18-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 18.6.2.1, “Fixed-function Performance Counters”). Software must re-enable counts by writing 1s to the corresponding enable bits in IA32\_PERF\_GLOBAL\_CTRL before leaving a PMI service routine to continue counter operation.
- Streamlined Freeze\_Perfmon\_on\_PMI is supported for ArchPerfMonVerID >= 4. The processor behaves as follows:
  - sets IA32\_PERF\_GLOBAL\_STATUS.CTR\_Frz =1 to disable counting on a counter overflow condition, but does not change the IA32\_PERF\_GLOBAL\_CTRL MSR.

Freezing LBRs and PMCs on PMIs (both legacy and streamlined operation) occur when one of the following applies:

- A performance counter had an overflow and was programmed to signal a PMI in case of an overflow.
  - For the general-purpose counters; enabling PMI is done by setting bit 20 of the IA32\_PERFEVTSELx register.
  - For the fixed-function counters; enabling PMI is done by setting the 3rd bit in the corresponding 4-bit control field of the MSR\_PERF\_FIXED\_CTR\_CTRL register (see Figure 18-1) or IA32\_FIXED\_CTR\_CTRL MSR (see Figure 18-2).
- The PEBS buffer is almost full and reaches the interrupt threshold.
- The BTS buffer is almost full and reaches the interrupt threshold.

Table 17-3 compares the interaction of the processor with the PMI handler using the legacy versus streamlined Freeza\_Perfmon\_On\_PMI interface.

**Table 17-3. Legacy and Streamlined Operation with Freeze\_Perfmon\_On\_PMI = 1, Counter Overflowed**

Legacy Freeze_Perfmon_On_PMI	Streamlined Freeze_Perfmon_On_PMI	Comment
Processor freezes the counters on overflow	Processor freezes the counters on overflow	Unchanged
Processor clears IA32_PERF_GLOBAL_CTRL	Processor set IA32_PERF_GLOBAL_STATUS.CTR_FTZ	
Handler reads IA32_PERF_GLOBAL_STATUS (0x38E) to examine which counter(s) overflowed	<b>mask</b> = RDMSR(0x38E)	Similar
Handler services the PMI	Handler services the PMI	Unchanged
Handler writes 1s to IA32_PERF_GLOBAL_OVF_CTL (0x390)	Handler writes <b>mask</b> into IA32_PERF_GLOBAL_OVF_RESET (0x390)	
Processor clears IA32_PERF_GLOBAL_STATUS	Processor clears IA32_PERF_GLOBAL_STATUS	Unchanged
Handler re-enables IA32_PERF_GLOBAL_CTRL	None	Reduced software overhead

## 17.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 17-4 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily\_DisplayModel encoding (see CPUID instruction in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

**Table 17-4. LBR Stack Size and TOS Pointer Range**

DisplayFamily_DisplayModel	Size of LBR Stack	Component of an LBR Entry	Range of TOS Pointer
06_5CH, 06_5FH	32	FROM_IP, TO_IP	0 to 31
06_4EH, 06_5EH, 06_8EH, 06_9EH, 06_55H, 06_66H, 06_7AH, 06_67H, 06_6AH, 06_6CH, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH	32	FROM_IP, TO_IP, LBR_INFO <sup>1</sup>	0 to 31
06_3DH, 06_47H, 06_4FH, 06_56H, 06_3CH, 06_45H, 06_46H, 06_3FH, 06_2AH, 06_2DH, 06_3AH, 06_3EH, 06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	16	FROM_IP, TO_IP	0 to 15
06_17H, 06_1DH, 06_0FH	4	FROM_IP, TO_IP	0 to 3
06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH, 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	8	FROM_IP, TO_IP	0 to 7

**NOTES:**

1. See Section 17.12.

The last branch recording mechanism tracks not only branch instructions (like JMP, Jcc, LOOP and CALL instructions), but also other operations that cause a change in the instruction pointer (like external interrupts, traps and faults). The branch recording mechanisms generally employs a set of MSRs, referred to as last branch record (LBR) stack. The size and exact locations of the LBR stack are generally model-specific (see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for model-specific MSR addresses).

- **Last Branch Record (LBR) Stack** — The LBR consists of N pairs of MSRs (N is listed in the LBR stack size column of Table 17-4) that store source and destination address of recent branches (see Figure 17-3):
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address is model specific) through the next consecutive (N-1) MSR address store source addresses.
  - MSR\_LASTBRANCH\_0\_TO\_IP (address is model specific ) through the next consecutive (N-1) MSR address store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant M bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address is model specific) contains an M-bit pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. The valid range of the M-bit POS pointer is given in Table 17-4.

### 17.4.8.1 LBR Stack and Intel® 64 Processors

LBR MSRs are 64-bits. In 64-bit mode, last branch records store the full address. Outside of 64-bit mode, the upper 32-bits of branch addresses will be stored as 0.

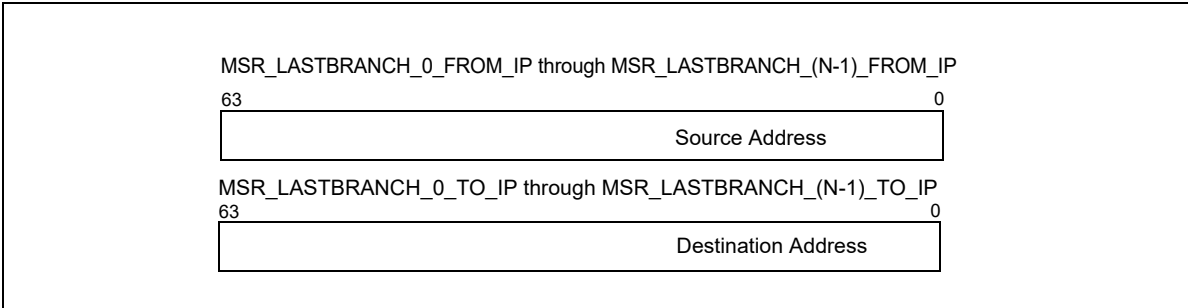


Figure 17-4. 64-bit Address Layout of LBR MSR

Software should query an architectural MSR IA32\_PERF\_CAPABILITIES[5:0] about the format of the address that is stored in the LBR stack. Four formats are defined by the following encoding:

- **000000B (32-bit record format)** — Stores 32-bit offset in current CS of respective source/destination,
- **000001B (64-bit LIP record format)** — Stores 64-bit linear address of respective source/destination,
- **000010B (64-bit EIP record format)** — Stores 64-bit offset (effective address) of respective source/destination.
- **000011B (64-bit EIP record format) and Flags** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction info is reported in the upper bit of 'FROM' registers in the LBR stack. See LBR stack details below for flag support and definition.
- **000100B (64-bit EIP record format), Flags and TSX** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction and TSX info are reported in the upper bits of 'FROM' registers in the LBR stack.
- **000101B (64-bit EIP record format), Flags, TSX, LBR\_INFO** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction, TSX, and elapsed cycles since the last LBR update are reported in the LBR\_INFO MSR stack.
- **000110B (64-bit LIP record format), Flags, Cycles** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction info is reported in the upper bits of 'FROM' registers in the LBR stack. Elapsed cycles since the last LBR update are reported in the upper 16 bits of the 'TO' registers in the LBR stack (see Section 17.6).
- **000111B (64-bit LIP record format), Flags, LBR\_INFO** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction, and elapsed cycles since the last LBR update are reported in the LBR\_INFO MSR stack.

Processor's support for the architectural MSR IA32\_PERF\_CAPABILITIES is provided by CPUID.01H:ECX[PERF\_CAPAB\_MSR] (bit 15).

### 17.4.8.2 LBR Stack and IA-32 Processors

The LBR MSRs in IA-32 processors introduced prior to Intel 64 architecture store the 32-bit "To Linear Address" and "From Linear Address" using the high and low half of each 64-bit MSR.

### 17.4.8.3 Last Exception Records and Intel 64 Architecture

Intel 64 and IA-32 processors also provide MSRs that store the branch record for the last branch taken prior to an exception or an interrupt. The location of the last exception record (LER) MSRs are model specific. The MSRs that store last exception records are 64-bits. If IA-32e mode is disabled, only the lower 32-bits of the address is recorded. If IA-32e mode is enabled, the processor writes 64-bit values into the MSR. In 64-bit mode, last exception records store 64-bit addresses; in compatibility mode, the upper 32-bits of last exception records are cleared.



## 17.4.9 BTS and DS Save Area

The **Debug store (DS)** feature flag (bit 21), returned by CPUID.1:EDX[21] indicates that the processor provides the debug store (DS) mechanism. The DS mechanism allows:

- BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, “Branch Trace Store (BTS).”
- Processor event-based sampling (PEBS) also uses the DS save area provided by debug store mechanism. The capability of PEBS varies across different microarchitectures. See Section 18.6.2.4, “Processor Event Based Sampling (PEBS),” and the relevant PEBS sub-sections across the core PMU sections in Chapter 18, “Performance Monitoring.”

When CPUID.1:EDX[21] is set:

- The BTS\_UNAVAILABLE and PEBS\_UNAVAILABLE flags in the IA32\_MISC\_ENABLE MSR indicate (when clear) the availability of the BTS and PEBS facilities, including the ability to set the BTS and BTINT bits in the appropriate DEBUGCTL MSR.
- The IA32\_DS\_AREA MSR exists and points to the DS save area.

The debug store (DS) save area is a software-designated area of memory that is used to collect the following two types of information:

- **Branch records** — When the BTS flag in the IA32\_DEBUGCTL MSR is set, a branch record is stored in the BTS buffer in the DS save area whenever a taken branch, interrupt, or exception is detected.
- **PEBS records** — When a performance counter is configured for PEBS, a PEBS record is stored in the PEBS buffer in the DS save area after the counter overflow occurs. This record contains the architectural state of the processor (state of the 8 general purpose registers, EIP register, and EFLAGS register) at the next occurrence of the PEBS event that caused the counter to overflow. When the state information has been logged, the counter is automatically reset to a specified value, and event counting begins again. The content layout of a PEBS record varies across different implementations that support PEBS. See Section 18.6.2.4.2 for details of enumerating PEBS record format.

### NOTES

Prior to processors based on the Goldmont microarchitecture, PEBS facility only supports a subset of implementation-specific precise events. See Section 18.5.3.1 for a PEBS enhancement that can generate records for both precise and non-precise events.

The DS save area and recording mechanism are disabled on INIT, processor Reset or transition to system-management mode (SMM) or IA-32e mode. It is similarly disabled on the generation of a machine-check exception on 45nm and 32nm Intel Atom processors and on processors with Netburst or Intel Core microarchitecture.

The BTS and PEBS facilities may not be available on all processors. The availability of these facilities is indicated by the BTS\_UNAVAILABLE and PEBS\_UNAVAILABLE flags, respectively, in the IA32\_MISC\_ENABLE MSR (see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*).

The DS save area is divided into three parts: buffer management area, branch trace store (BTS) buffer, and PEBS buffer (see Figure 17-5). The buffer management area is used to define the location and size of the BTS and PEBS buffers. The processor then uses the buffer management area to keep track of the branch and/or PEBS records in their respective buffers and to record the performance counter reset value. The linear address of the first byte of the DS buffer management area is specified with the IA32\_DS\_AREA MSR.

The fields in the buffer management area are as follows:

- **BTS buffer base** — Linear address of the first byte of the BTS buffer. This address should point to a natural doubleword boundary.
- **BTS index** — Linear address of the first byte of the next BTS record to be written to. Initially, this address should be the same as the address in the BTS buffer base field.
- **BTS absolute maximum** — Linear address of the next byte past the end of the BTS buffer. This address should be a multiple of the BTS record size (12 bytes) plus 1.

- **BTS interrupt threshold** — Linear address of the BTS record on which an interrupt is to be generated. This address must point to an offset from the BTS buffer base that is a multiple of the BTS record size. Also, it must be several records short of the BTS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the BTS absolute maximum record.
- **PEBS buffer base** — Linear address of the first byte of the PEBS buffer. This address should point to a natural doubleword boundary.
- **PEBS index** — Linear address of the first byte of the next PEBS record to be written to. Initially, this address should be the same as the address in the PEBS buffer base field.
- **PEBS absolute maximum** — Linear address of the next byte past the end of the PEBS buffer. This address should be a multiple of the PEBS record size (40 bytes) plus 1.
- **PEBS interrupt threshold** — Linear address of the PEBS record on which an interrupt is to be generated. This address must point to an offset from the PEBS buffer base that is a multiple of the PEBS record size. Also, it must be several records short of the PEBS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the PEBS absolute maximum record.
- **PEBS counter reset value** — A 64-bit value that the counter is to be set to when a PEBS record is written. Bits beyond the size of the counter are ignored. This value allows state information to be collected regularly every time the specified number of events occur.

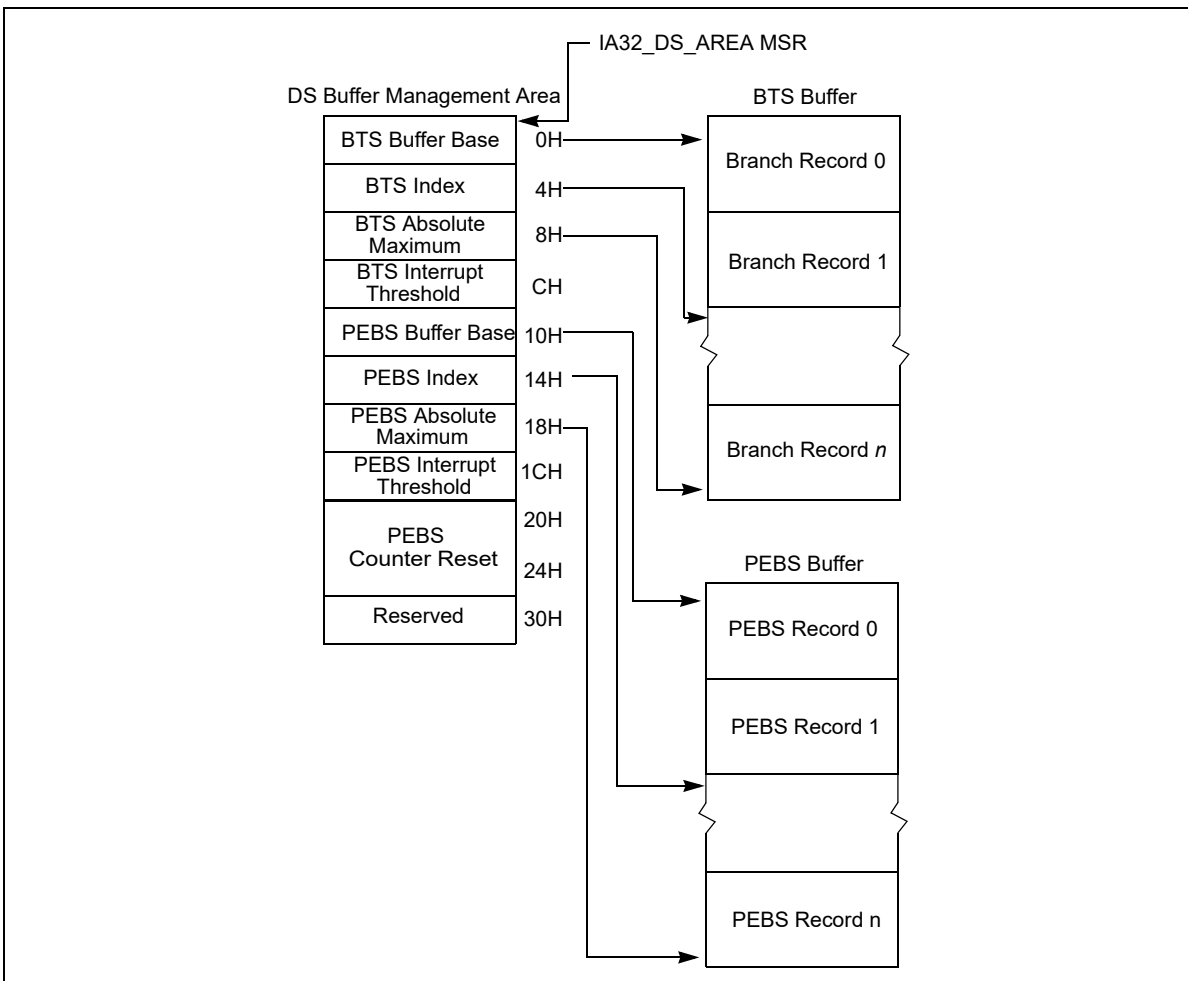


Figure 17-5. DS Save Area Example<sup>1</sup>

**NOTES:**

1. This example represents the format for a system that supports PEBS on only one counter.



Figure 17-6 shows the structure of a 12-byte branch record in the BTS buffer. The fields in each record are as follows:

- **Last branch from** — Linear address of the instruction from which the branch, interrupt, or exception was taken.
- **Last branch to** — Linear address of the branch target or the first instruction in the interrupt or exception service routine.
- **Branch predicted** — Bit 4 of field indicates whether the branch that was taken was predicted (set) or not predicted (clear).

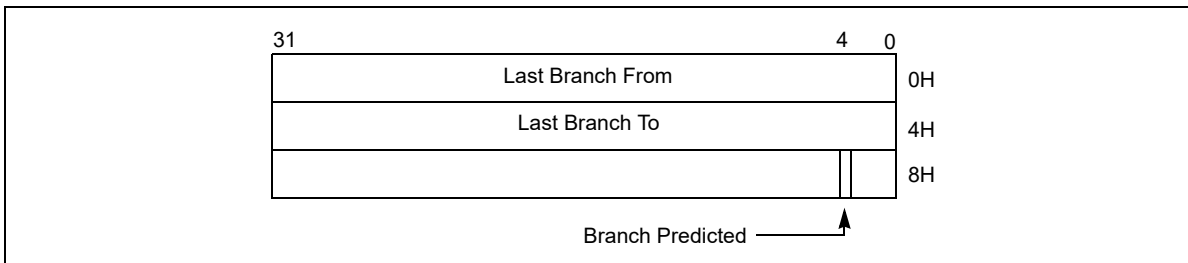


Figure 17-6. 32-bit Branch Trace Record Format

Figure 17-7 shows the structure of the 40-byte PEBS records. Nominally the register values are those at the beginning of the instruction that caused the event. However, there are cases where the registers may be logged in a partially modified state. The linear IP field shows the value in the EIP register translated from an offset into the current code segment to a linear address.

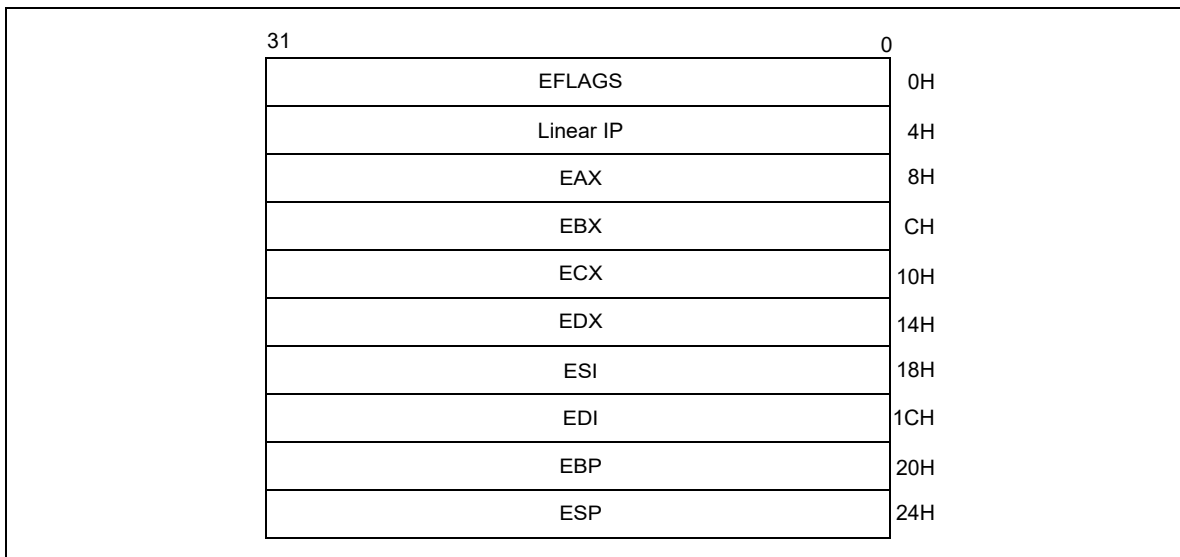


Figure 17-7. PEBS Record Format

### 17.4.9.1 64 Bit Format of the DS Save Area

When DTES64 = 1 (CPUID.1.ECX[2] = 1), the structure of the DS save area is shown in Figure 17-8.

When DTES64 = 0 (CPUID.1.ECX[2] = 0) and IA-32e mode is active, the structure of the DS save area is shown in Figure 17-8. If IA-32e mode is not active the structure of the DS save area is as shown in Figure 17-5.

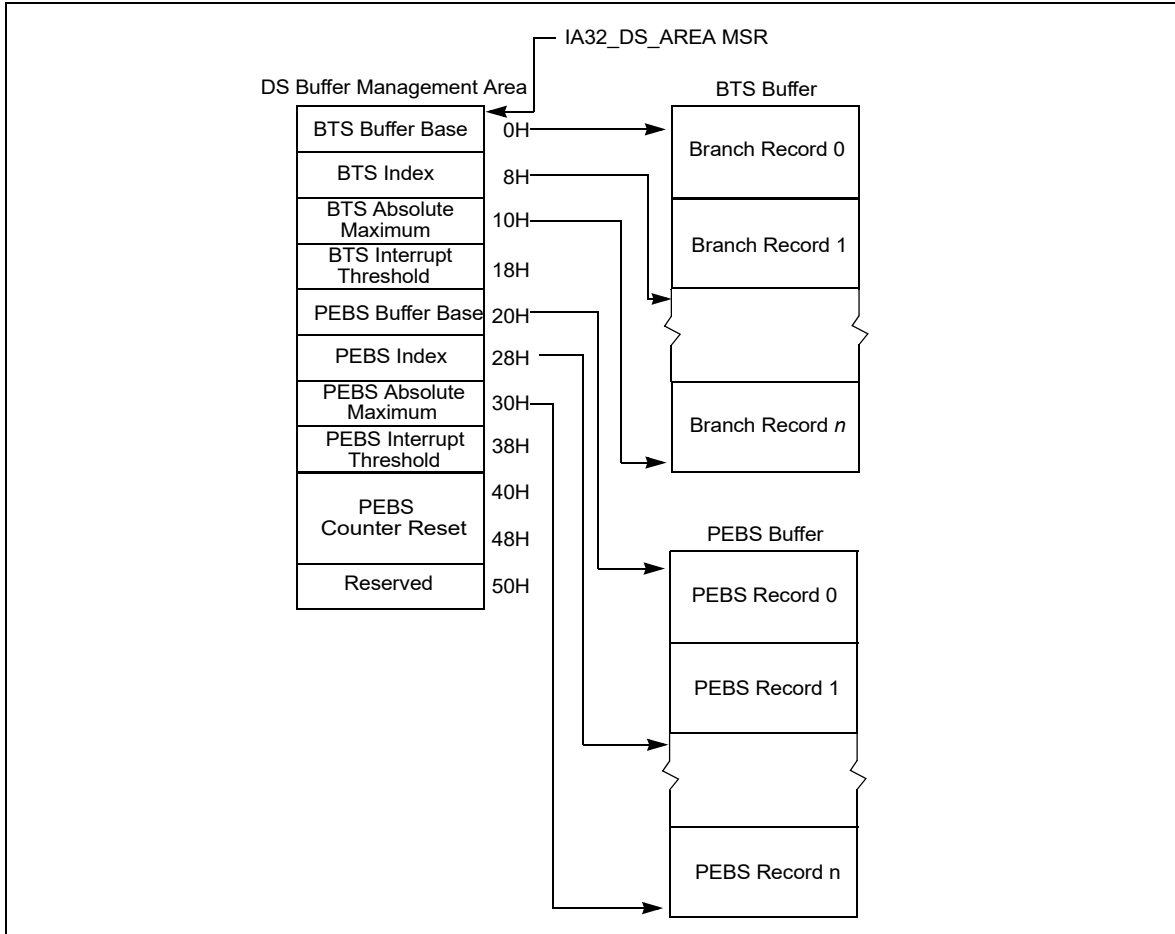


Figure 17-8. IA-32e Mode DS Save Area Example<sup>1</sup>

**NOTES:**

1. This example represents the format for a system that supports PEBS on only one counter.

The IA32\_DS\_AREA MSR holds the 64-bit linear address of the first byte of the DS buffer management area. The structure of a branch trace record is similar to that shown in Figure 17-6, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 17-9). The structure of a PEBS record is similar to that shown in Figure 17-7, but each field is 8 bytes in length and architectural states include register R8 through R15. This makes the size of a PEBS record in 64-bit mode 144 bytes (see Figure 17-10).

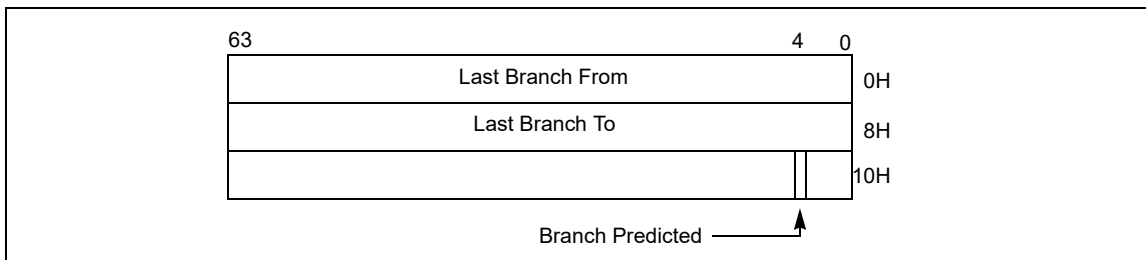
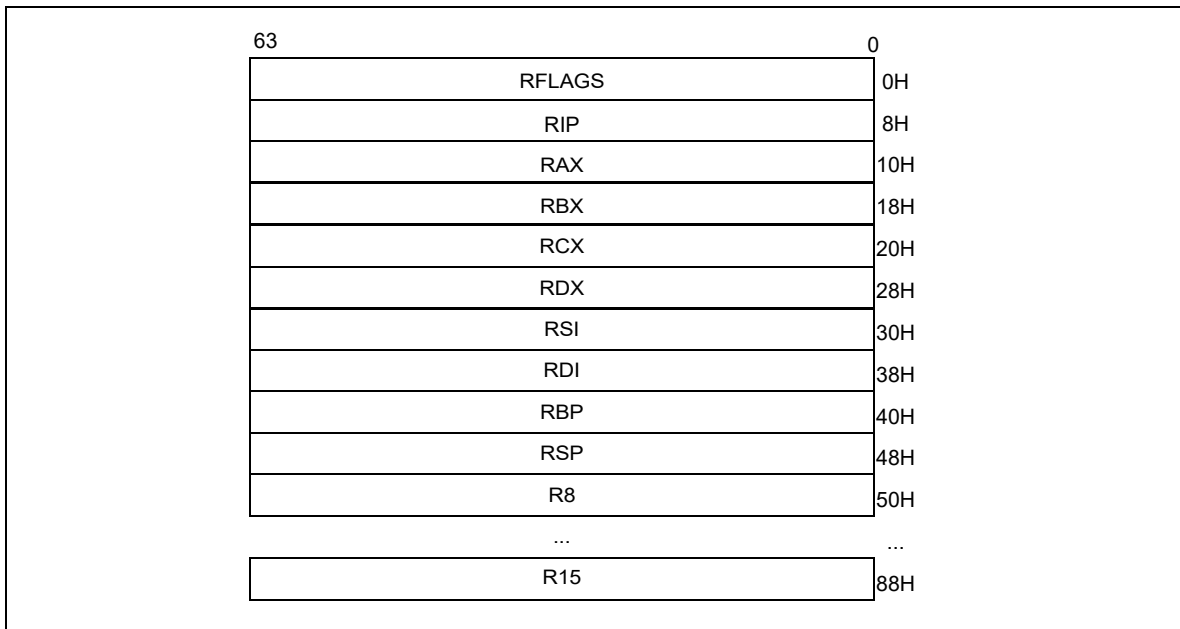


Figure 17-9. 64-bit Branch Trace Record Format



**Figure 17-10. 64-bit PEBS Record Format**

Fields in the buffer management area of a DS save area are described in Section 17.4.9.

The format of a branch trace record and a PEBS record are the same as the 64-bit record formats shown in Figures 17-9 and Figures 17-10, with the exception that the branch predicted bit is not supported by Intel Core microarchitecture or Intel Atom microarchitecture. The 64-bit record formats for BTS and PEBS apply to DS save area for all operating modes.

The procedures used to program IA32\_DEBUGCTL MSR to set up a BTS buffer or a CPL-qualified BTS are described in Section 17.4.9.3 and Section 17.4.9.4.

Required elements for writing a DS interrupt service routine are largely the same on processors that support using DS Save area for BTS or PEBS records. However, on processors based on Intel NetBurst® microarchitecture, re-enabling counting requires writing to CCCRs. But a DS interrupt service routine on processors supporting architectural performance monitoring should:

- Re-enable the enable bits in IA32\_PERF\_GLOBAL\_CTRL MSR if it is servicing an overflow PMI due to PEBS.
- Clear overflow indications by writing to IA32\_PERF\_GLOBAL\_OVF\_CTRL when a counting configuration is changed. This includes bit 62 (ClrOvfBuffer) and the overflow indication of counters used in either PEBS or general-purpose counting (specifically: bits 0 or 1; see Figures 18-3).

### 17.4.9.2 Setting Up the DS Save Area

To save branch records with the BTS buffer, the DS save area must first be set up in memory as described in the following procedure (See Section 18.6.2.4.1, “Setting up the PEBS Buffer,” for instructions for setting up a PEBS buffer, respectively, in the DS save area):

1. Create the DS buffer management information area in memory (see Section 17.4.9, “BTS and DS Save Area,” and Section 17.4.9.1, “64 Bit Format of the DS Save Area”). Also see the additional notes in this section.
2. Write the base linear address of the DS buffer management area into the IA32\_DS\_AREA MSR.
3. Set up the performance counter entry in the xAPIC LVT for fixed delivery and edge sensitive. See Section 10.5.1, “Local Vector Table.”
4. Establish an interrupt handler in the IDT for the vector associated with the performance counter entry in the xAPIC LVT.

5. Write an interrupt service routine to handle the interrupt. See Section 17.4.9.5, “Writing the DS Interrupt Service Routine.”

The following restrictions should be applied to the DS save area.

- The recording of branch records in the BTS buffer (or PEBS records in the PEBS buffer) may not operate properly if accesses to the linear addresses in any of the three DS save area sections cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do “lazy” page-table entry propagation for these pages. Some newer processor generations support “lazy” EPT page-table entry propagation for PEBS; see Section 18.3.10.1 and Section 18.9.5 for more information. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.
- The DS save area can be larger than a page, but the pages must be mapped to contiguous linear addresses. The buffer may share a page, so it need not be aligned on a 4-KByte boundary. For performance reasons, the base of the buffer must be aligned on a doubleword boundary and should be aligned on a cache line boundary.
- It is recommended that the buffer size for the BTS buffer and the PEBS buffer be an integer multiple of the corresponding record sizes.
- The precise event records buffer should be large enough to hold the number of precise event records that can occur while waiting for the interrupt to be serviced.
- The DS save area should be in kernel space. It must not be on the same page as code, to avoid triggering self-modifying code actions.
- There are no memory type restrictions on the buffers, although it is recommended that the buffers be designated as WB memory type for performance considerations.
- Either the system must be prevented from entering A20M mode while DS save area is active, or bit 20 of all addresses within buffer bounds must be 0.
- Pages that contain buffers must be mapped to the same physical addresses for all processes, such that any change to control register CR3 will not change the DS addresses.
- The DS save area is expected to be used only on systems with an enabled APIC. The LVT Performance Counter entry in the APCI must be initialized to use an interrupt gate instead of the trap gate.

### 17.4.9.3 Setting Up the BTS Buffer

Three flags in the MSR\_DEBUGCTLA MSR (see Table 17-5), IA32\_DEBUGCTL (see Figure 17-3), or MSR\_DEBUGCTLB (see Figure 17-16) control the generation of branch records and storing of them in the BTS buffer; these are TR, BTS, and BTINT. The TR flag enables the generation of BTMs. The BTS flag determines whether the BTMs are sent out on the system bus (clear) or stored in the BTS buffer (set). BTMs cannot be simultaneously sent to the system bus and logged in the BTS buffer. The BTINT flag enables the generation of an interrupt when the BTS buffer is full. When this flag is clear, the BTS buffer is a circular buffer.

**Table 17-5. IA32\_DEBUGCTL Flag Encodings**

TR	BTS	BTINT	Description
0	X	X	Branch trace messages (BTMs) off
1	0	X	Generate BTMs
1	1	0	Store BTMs in the BTS buffer, used here as a circular buffer
1	1	1	Store BTMs in the BTS buffer, and generate an interrupt when the buffer is nearly full

The following procedure describes how to set up a DS Save area to collect branch records in the BTS buffer:

1. Place values in the BTS buffer base, BTS index, BTS absolute maximum, and BTS interrupt threshold fields of the DS buffer management area to set up the BTS buffer in memory.
2. Set the TR and BTS flags in the IA32\_DEBUGCTL for Intel Core Solo and Intel Core Duo processors or later processors (or MSR\_DEBUGCTLA MSR for processors based on Intel NetBurst Microarchitecture; or MSR\_DEBUGCTLB for Pentium M processors).

- Clear the BTINT flag in the corresponding IA32\_DEBUGCTL (or MSR\_DEBUGCTLA MSR; or MSR\_DEBUGCTLB) if a circular BTS buffer is desired.

### NOTES

If the buffer size is set to less than the minimum allowable value (i.e. BTS absolute maximum < 1 + size of BTS record), the results of BTS is undefined.

In order to prevent generating an interrupt, when working with circular BTS buffer, SW need to set BTS interrupt threshold to a value greater than BTS absolute maximum (fields of the DS buffer management area). It's not enough to clear the BTINT flag itself only.

#### 17.4.9.4 Setting Up CPL-Qualified BTS

If the processor supports CPL-qualified last branch recording mechanism, the generation of branch records and storing of them in the BTS buffer are determined by: TR, BTS, BTS\_OFF\_OS, BTS\_OFF\_USR, and BTINT. The encoding of these five bits are shown in Table 17-6.

**Table 17-6. CPL-Qualified Branch Trace Store Encodings**

TR	BTS	BTS_OFF_OS	BTS_OFF_USR	BTINT	Description
0	X	X	X	X	Branch trace messages (BTMs) off
1	0	X	X	X	Generates BTMs but do not store BTMs
1	1	0	0	0	Store all BTMs in the BTS buffer, used here as a circular buffer
1	1	1	0	0	Store BTMs with CPL > 0 in the BTS buffer
1	1	0	1	0	Store BTMs with CPL = 0 in the BTS buffer
1	1	1	1	X	Generate BTMs but do not store BTMs
1	1	0	0	1	Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	1	0	1	Store BTMs with CPL > 0 in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	0	1	1	Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full

#### 17.4.9.5 Writing the DS Interrupt Service Routine

The BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector and interrupt service routine (called the debug store interrupt service routine or DS ISR). To handle BTS, non-precise event-based sampling, and PEBS interrupts: separate handler routines must be included in the DS ISR. Use the following guidelines when writing a DS ISR to handle BTS, non-precise event-based sampling, and/or PEBS interrupts.

- The DS interrupt service routine (ISR) must be part of a kernel driver and operate at a current privilege level of 0 to secure the buffer storage area.
- Because the BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector, the DS ISR must check for all the possible causes of interrupts from these facilities and pass control on to the appropriate handler.

BTS and PEBS buffer overflow would be the sources of the interrupt if the buffer index matches/exceeds the interrupt threshold specified. Detection of non-precise event-based sampling as the source of the interrupt is accomplished by checking for counter overflow.

- There must be separate save areas, buffers, and state for each processor in an MP system.
- Upon entering the ISR, branch trace messages and PEBS should be disabled to prevent race conditions during access to the DS save area. This is done by clearing TR flag in the IA32\_DEBUGCTL (or MSR\_DEBUGCTLA MSR) and by clearing the precise event enable flag in the MSR\_PEBS\_ENABLE MSR. These settings should be restored to their original values when exiting the ISR.

- The processor will not disable the DS save area when the buffer is full and the circular mode has not been selected. The current DS setting must be retained and restored by the ISR on exit.
- After reading the data in the appropriate buffer, up to but not including the current index into the buffer, the ISR must reset the buffer index to the beginning of the buffer. Otherwise, everything up to the index will look like new entries upon the next invocation of the ISR.
- The ISR must clear the mask bit in the performance counter LVT entry.
- The ISR must re-enable the counters to count via IA32\_PERF\_GLOBAL\_CTRL/IA32\_PERF\_GLOBAL\_OVF\_CTRL if it is servicing an overflow PMI due to PEBS (or via CCCR's ENABLE bit on processor based on Intel NetBurst microarchitecture).
- The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

## 17.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL® ATOM™ PROCESSORS)

The Intel Core 2 Duo processor family and Intel Xeon processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture provide last branch interrupt and exception recording. The facilities described in this section also apply to 45 nm and 32 nm Intel Atom processors. These capabilities are similar to those found in Pentium 4 processors, including support for the following facilities:

- **Debug Trace and Branch Recording Control** — The IA32\_DEBUGCTL MSR provide bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-3 for the MSR layout.
- **Last branch record (LBR) stack** — There are a collection of MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.5.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts**
  - See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
  - 45 nm and 32 nm Intel Atom processors clear the TR flag when the FREEZE\_LBRS\_ON\_PMI flag is set.
- **Branch trace messages** — See Section 17.4.4.
- **Last exception records** — See Section 17.13.3.
- **Branch trace store and CPL-qualified BTS** — See Section 17.4.5.
- **FREEZE\_LBRS\_ON\_PMI flag (bit 11)** — see Section 17.4.7 for legacy Freeze\_LBRs\_On\_PMI operation.
- **FREEZE\_PERFMON\_ON\_PMI flag (bit 12)** — see Section 17.4.7 for legacy Freeze\_Perfmon\_On\_PMI operation.
- **FREEZE\_WHILE\_SMM (bit 14)** — FREEZE\_WHILE\_SMM is supported if IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is reporting 1. See Section 17.4.1.

### 17.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Atom processor families, and Intel processors based on Intel NetBurst microarchitecture.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 processors families and Intel processors based on Intel NetBurst microarchitecture:

- **Last Branch Record (LBR) Stack**
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address 40H) through MSR\_LASTBRANCH\_3\_FROM\_IP (address 43H) store source addresses
  - MSR\_LASTBRANCH\_0\_TO\_IP (address 60H) through MSR\_LASTBRANCH\_3\_TO\_IP (address 63H) store destination addresses

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for 45 nm and 32 nm Intel Atom processors:

- **Last Branch Record (LBR) Stack**
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address 40H) through MSR\_LASTBRANCH\_7\_FROM\_IP (address 47H) store source addresses
  - MSR\_LASTBRANCH\_0\_TO\_IP (address 60H) through MSR\_LASTBRANCH\_7\_TO\_IP (address 67H) store destination addresses
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

The address format written in the FROM\_IP/TO\_IP MSRS may differ between processors. Software should query IA32\_PERF\_CAPABILITIES[5:0] and consult Section 17.4.8.1. The behavior of the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

### 17.5.2 LBR Stack in Intel Atom Processors based on the Silvermont Microarchitecture

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in Intel Atom processors based on the Silvermont and Airmont microarchitectures. Eight pairs of MSRs are supported in the LBR stack.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR\_LBR\_SELECT. The layout of MSR\_LBR\_SELECT is described in Table 17-11.

## 17.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Processors based on the Goldmont microarchitecture extend the capabilities described in Section 17.5.2 with the following enhancements:

- Supports new LBR format encoding 00110b in IA32\_PERF\_CAPABILITIES[5:0].
- Size of LBR stack increased to 32. Each entry includes MSR\_LASTBRANCH\_x\_FROM\_IP (address 0x680..0x69f) and MSR\_LASTBRANCH\_x\_TO\_IP (address 0x6c0..0x6df).
- LBR call stack filtering supported. The layout of MSR\_LBR\_SELECT is described in Table 17-13.
- Elapsed cycle information is added to MSR\_LASTBRANCH\_x\_TO\_IP. Format is shown in Table 17-7.
- Misprediction info is reported in the upper bits of MSR\_LASTBRANCH\_x\_FROM\_IP. MISRPRED bit format is shown in Table 17-8.
- Streamlined Freeze\_LBRs\_On\_PMI operation; see Section 17.12.2.
- LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 17.12.3.

**Table 17-7. MSR\_LASTBRANCH\_x\_TO\_IP for the Goldmont Microarchitecture**

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch to” address. See Section 17.4.8.1 for address format.
Cycle Count (Saturating)	63:48	R/W	Elapsed core clocks since last update to the LBR stack.

## 17.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont Plus microarchitecture. Processors based on the Goldmont Plus microarchitecture extend the capabilities described in Section 17.6 with the following changes:

- Enumeration of new LBR format: encoding 00111b in IA32\_PERF\_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of three MSRs:
  - MSR\_LASTBRANCH\_x\_FROM\_IP, the layout is simplified, see Table 17-9.
  - MSR\_LASTBRANCH\_x\_TO\_IP, the layout is the same as Table 17-9.
  - MSR\_LBR\_INFO\_x, stores branch prediction flag, TSX info, and elapsed cycle data. Layout is the same as Table 17-16.

## 17.8 LAST BRANCH, INTERRUPT AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in the Intel® Xeon Phi™ processor 7200/5200/3200 series based on the Knights Landing microarchitecture. Eight pairs of MSRs are supported in the LBR stack, per thread:

- **Last Branch Record (LBR) Stack**
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address 680H) through MSR\_LASTBRANCH\_7\_FROM\_IP (address 687H) store source addresses.
  - MSR\_LASTBRANCH\_0\_TO\_IP (address 6C0H) through MSR\_LASTBRANCH\_7\_TO\_IP (address 6C7H) store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR\_LBR\_SELECT. The layout of MSR\_LBR\_SELECT is described in Table 17-11.

The address format written in the FROM\_IP/TO\_IP MSRS may differ between processors. Software should query IA32\_PERF\_CAPABILITIES[5:0] and consult Section 17.4.8.1. The behavior of the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

## 17.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

The processors based on Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and adds additional capabilities:

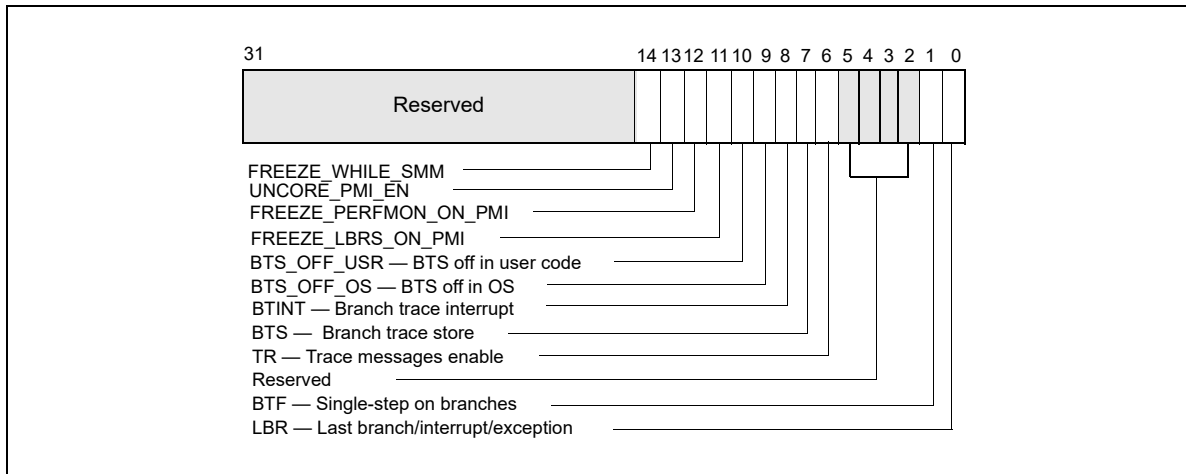
- **Debug Trace and Branch Recording Control** — The IA32\_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-11 for the MSR layout.
- **Last branch record (LBR) stack** — There are 16 MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.9.1.



- **Monitoring and single-stepping of branches, exceptions, and interrupts** — See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
- **Branch trace messages** — The IA32\_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 17.4.4. However, not all BTM messages are observable using the Intel® QPI link.
- **Last exception records** — See Section 17.13.3.
- **Branch trace store and CPL-qualified BTS** — See Section 17.4.6 and Section 17.4.5.
- **FREEZE\_LBRS\_ON\_PMI flag (bit 11)** — see Section 17.4.7 for legacy Freeze\_LBRS\_On\_PMI operation.
- **FREEZE\_PERFMON\_ON\_PMI flag (bit 12)** — see Section 17.4.7 for legacy Freeze\_Perfmon\_On\_PMI operation.
- **UNCORE\_PMI\_EN (bit 13)** — When set, this logical processor is enabled to receive an uncore counter overflow interrupt from the uncore.
- **FREEZE\_WHILE\_SMM (bit 14)** — FREEZE\_WHILE\_SMM is supported if IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is reporting 1. See Section 17.4.1.

Processors based on Intel microarchitecture code name Nehalem provide additional capabilities:

- **Independent control of uncore PMI** — The IA32\_DEBUGCTL MSR provides a bit field (see Figure 17-11) for software to enable each logical processor to receive an uncore counter overflow interrupt.
- **LBR filtering** — Processors based on Intel microarchitecture code name Nehalem support filtering of LBR based on combination of CPL and branch type conditions. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR\_LBR\_SELECT.



**Figure 17-11. IA32\_DEBUGCTL MSR for Processors based on Intel microarchitecture code name Nehalem**

## 17.9.1 LBR Stack

Processors based on Intel microarchitecture code name Nehalem provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 17-8 and Table 17-9.

**Table 17-8. MSR\_LASTBRANCH\_x\_FROM\_IP**

Bit Field	Bit Offset	Access	Description
<b>Data</b>	47:0	R/W	This is the “branch from” address. See Section 17.4.8.1 for address format.
<b>SIGN_EXT</b>	62:48	R/W	Signed extension of bit 47 of this register.
<b>MISPRED</b>	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

**Table 17-9. MSR\_LASTBRANCH\_x\_TO\_IP**

Bit Field	Bit Offset	Access	Description
<b>Data</b>	47:0	R/W	This is the “branch to” address. See Section 17.4.8.1 for address format
<b>SIGN_EXT</b>	63:48	R/W	Signed extension of bit 47 of this register.

Processors based on Intel microarchitecture code name Nehalem have an LBR MSR Stack as shown in Table 17-10.

**Table 17-10. LBR Stack Size and TOS Pointer Range**

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
<b>06_1AH</b>	16	0 to 15

## 17.9.2 Filtering of Last Branch Records

MSR\_LBR\_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e. all branches will be captured. MSR\_LBR\_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR\_LBR\_SELECT is shown in Table 17-11.

**Table 17-11. MSR\_LBR\_SELECT for Intel microarchitecture code name Nehalem**

Bit Field	Bit Offset	Access	Description
<b>CPL_EQ_0</b>	0	R/W	When set, do not capture branches ending in ring 0
<b>CPL_NEQ_0</b>	1	R/W	When set, do not capture branches ending in ring >0
<b>JCC</b>	2	R/W	When set, do not capture conditional branches
<b>NEAR_REL_CALL</b>	3	R/W	When set, do not capture near relative calls
<b>NEAR_IND_CALL</b>	4	R/W	When set, do not capture near indirect calls
<b>NEAR_RET</b>	5	R/W	When set, do not capture near returns
<b>NEAR_IND_JMP</b>	6	R/W	When set, do not capture near indirect jumps
<b>NEAR_REL_JMP</b>	7	R/W	When set, do not capture near relative jumps
<b>FAR_BRANCH</b>	8	R/W	When set, do not capture far branches
<b>Reserved</b>	63:9		Must be zero

## 17.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem”, apply to processors based on Intel microarchitecture code name Sandy Bridge. For processors based on Intel microarchitecture code name Ivy Bridge, the same holds true.

One difference of note is that MSR\_LBR\_SELECT is shared between two logical processors in the same core. In Intel microarchitecture code name Sandy Bridge, each logical processor has its own MSR\_LBR\_SELECT. The filtering semantics for “Near\_ind\_jmp” and “Near\_rel\_jmp” has been enhanced, see Table 17-12.

**Table 17-12. MSR\_LBR\_SELECT for Intel® microarchitecture code name Sandy Bridge**

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

## 17.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge”, apply to next generation processors based on Intel microarchitecture code name Haswell.

The LBR facility also supports an alternate capability to profile call stack profiles. Configuring the LBR facility to conduct call stack profiling is by writing 1 to the MSR\_LBR\_SELECT.EN\_CALLSTACK[bit 9]; see Table 17-13. If MSR\_LBR\_SELECT.EN\_CALLSTACK is clear, the LBR facility will capture branches normally as described in Section 17.10.

**Table 17-13. MSR\_LBR\_SELECT for Intel® microarchitecture code name Haswell**

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.

**Table 17-13. MSR\_LBR\_SELECT for Intel® microarchitecture code name Haswell**

Bit Field	Bit Offset	Access	Description
FAR_BRANCH	8	R/W	When set, do not capture far branches
EN_CALLSTACK <sup>1</sup>	9		Enable LBR stack to use LIFO filtering to capture Call stack profile
Reserved	63:10		Must be zero

**NOTES:**

1. Must set valid combination of bits 0-8 in conjunction with bit 9 (as described below), otherwise the contents of the LBR MSRs are undefined.

The call stack profiling capability is an enhancement of the LBR facility. The LBR stack is a ring buffer typically used to profile control flow transitions resulting from branches. However, the finite depth of the LBR stack often become less effective when profiling certain high-level languages (e.g. C++), where a transition of the execution flow is accompanied by a large number of leaf function calls, each of which returns an individual parameter to form the list of parameters for the main execution function call. A long list of such parameters returned by the leaf functions would serve to flush the data captured in the LBR stack, often losing the main execution context.

When the call stack feature is enabled, the LBR stack will capture unfiltered call data normally, but as return instructions are executed the last captured branch record is flushed from the on-chip registers in a last-in first-out (LIFO) manner. Thus, branch information relative to leaf functions will not be captured, while preserving the call stack information of the main line execution path.

The configuration of the call stack facility is summarized below:

- Set IA32\_DEBUGCTL.LBR (bit 0) to enable the LBR stack to capture branch records. The source and target addresses of the call branches will be captured in the 16 pairs of From/To LBR MSRs that form the LBR stack.
- Program the Top of Stack (TOS) MSR that points to the last valid from/to pair. This register is incremented by 1, modulo 16, before recording the next pair of addresses.
- Program the branch filtering bits of MSR\_LBR\_SELECT (bits 0:8) as desired.
- Program the MSR\_LBR\_SELECT to enable LIFO filtering of return instructions with:
  - The following bits in MSR\_LBR\_SELECT must be set to '1': JCC, NEAR\_IND\_JMP, NEAR\_REL\_JMP, FAR\_BRANCH, EN\_CALLSTACK;
  - The following bits in MSR\_LBR\_SELECT must be cleared: NEAR\_REL\_CALL, NEAR-IND\_CALL, NEAR\_RET;
  - At most one of CPL\_EQ\_0, CPL\_NEQ\_0 is set.

Note that when call stack profiling is enabled, “zero length calls” are excluded from writing into the LBRs. (A “zero length call” uses the attribute of the call instruction to push the immediate instruction pointer on to the stack and then pops off that address into a register. This is accomplished without any matching return on the call.)

### 17.11.1 LBR Stack Enhancement

Processors based on Intel microarchitecture code name Haswell provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is enumerated by IA32\_PERF\_CAPABILITIES[5:0] = 04H, and is shown in Table 17-14 and Table 17-9.

**Table 17-14. MSR\_LASTBRANCH\_x\_FROM\_IP with TSX Information**

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch from” address. See Section 17.4.8.1 for address format.
SIGN_EXT	60:48	R/W	Signed extension of bit 47 of this register.
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region, or EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region

**Table 17-14. MSR\_LASTBRANCH\_x\_FROM\_IP with TSX Information (Contd.)**

Bit Field	Bit Offset	Access	Description
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

## 17.12 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SKYLAKE MICROARCHITECTURE

Processors based on the Skylake microarchitecture provide a number of enhancement with storing last branch records:

- enumeration of new LBR format: encoding 00101b in IA32\_PERF\_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of a triplets of MSRs:
  - MSR\_LASTBRANCH\_x\_FROM\_IP, the layout is simplified, see Table 17-9.
  - MSR\_LASTBRANCH\_x\_TO\_IP, the layout is the same as Table 17-9.
  - MSR\_LBR\_INFO\_x, stores branch prediction flag, TSX info, and elapsed cycle data.
- Size of LBR stack increased to 32.

Processors based on the Skylake microarchitecture supports the same LBR filtering capabilities as described in Table 17-13.

**Table 17-15. LBR Stack Size and TOS Pointer Range**

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_4EH, 06_5EH	32	0 to 31

### 17.12.1 MSR\_LBR\_INFO\_x MSR

The layout of each MSR\_LBR\_INFO\_x MSR is shown in Table 17-16.

**Table 17-16. MSR\_LBR\_INFO\_x**

Bit Field	Bit Offset	Access	Description
Cycle Count (saturating)	15:0	R/W	Elapsed core clocks since last update to the LBR stack.
Reserved	60:16	R/W	Reserved
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region OR EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

### 17.12.2 Streamlined Freeze\_LBRs\_On\_PMI Operation

The FREEZE\_LBRS\_ON\_PMI feature causes the LBRs to be frozen on a hardware request for a PMI. This prevents the LBRs from being overwritten by new branches, allowing the PMI handler to examine the control flow that preceded the PMI generation. Architectural performance monitoring version 4 and above supports a streamlined FREEZE\_LBRS\_ON\_PMI operation for PMI service routine that replaces the legacy FREEZE\_LBRS\_ON\_PMI operation (see Section 17.4.7).

While the legacy FREEZE\_LBRS\_ON\_PMI clear the LBR bit in the IA32\_DEBUGCTL MSR on a PMI request, the streamlined FREEZE\_LBRS\_ON\_PMI will set the LBR\_FRZ bit in IA32\_PERF\_GLOBAL\_STATUS. Branches will not cause the LBRs to be updated when LBR\_FRZ is set. Software can clear LBR\_FRZ at the same time as it clears overflow bits by setting the LBR\_FRZ bit as well as the needed overflow bit when writing to IA32\_PERF\_GLOBAL\_STATUS\_RESET MSR.

This streamlined behavior avoids race conditions between software and processor writes to IA32\_DEBUGCTL that are possible with FREEZE\_LBRS\_ON\_PMI clearing of the LBR enable.

### 17.12.3 LBR Behavior and Deep C-State

When MWAIT is used to request a C-state that is numerically higher than C1, then LBR state may be initialized to zero depending on optimized “waiting” state that is selected by the processor. The affected LBR states include the FROM, TO, INFO, LAST\_BRANCH, LER and LBR\_TOS registers. The LBR enable bit and LBR\_FROZEN bit are not affected. The LBR-time of the first LBR record inserted after an exit from such a C-state request will be zero.

## 17.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

Pentium 4 and Intel Xeon processors based on Intel NetBurst microarchitecture provide the following methods for recording taken branches, interrupts and exceptions:

- Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address.
- Send the branch records out on the system bus as branch trace messages (BTMs).
- Log BTMs in a memory-resident branch trace store (BTS) buffer.

To support these functions, the processor provides the following MSRs and related facilities:

- **MSR\_DEBUGCTLA MSR** — Enables last branch, interrupt, and exception recording; single-stepping on taken branches; branch trace messages (BTMs); and branch trace store (BTS). This register is named DebugCtlMSR in the P6 family processors.
- **Debug store (DS) feature flag (CPUID.1:EDX.DS[bit 21])** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer.
- **CPL-qualified debug store (DS) feature flag (CPUID.1:ECX.DS-CPL[bit 4])** — Indicates that the processor provides a CPL-qualified debug store (DS) mechanism, which allows software to selectively skip sending and storing BTMs, according to specified current privilege level settings, into a memory-resident BTS buffer.
- **IA32\_MISC\_ENABLE MSR** — Indicates that the processor provides the BTS facilities.
- **Last branch record (LBR) stack** — The LBR stack is a circular stack that consists of four MSRs (MSR\_LASTBRANCH\_0 through MSR\_LASTBRANCH\_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR\_LASTBRANCH\_0\_FROM\_IP through MSR\_LASTBRANCH\_15\_FROM\_IP and MSR\_LASTBRANCH\_0\_TO\_IP through MSR\_LASTBRANCH\_15\_TO\_IP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- **Last branch record top-of-stack (TOS) pointer** — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the

Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H]. See also: Table 17-17, Figure 17-12, and Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

- **Last exception record** — See Section 17.13.3, “Last Exception Records.”

### 17.13.1 MSR\_DEBUGCTLA MSR

The MSR\_DEBUGCTLA MSR enables and disables the various last branch recording mechanisms described in the previous section. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-12 shows the flags in the MSR\_DEBUGCTLA MSR. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. Each branch, interrupt, or exception is recorded as a 64-bit branch record. The processor clears this flag whenever a debug exception is generated (for example, when an instruction or data breakpoint or a single-step trap occurs). See Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches.”
- **TR (trace message enable) flag (bit 2)** — When set, branch trace messages are enabled. Thereafter, when the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages.”

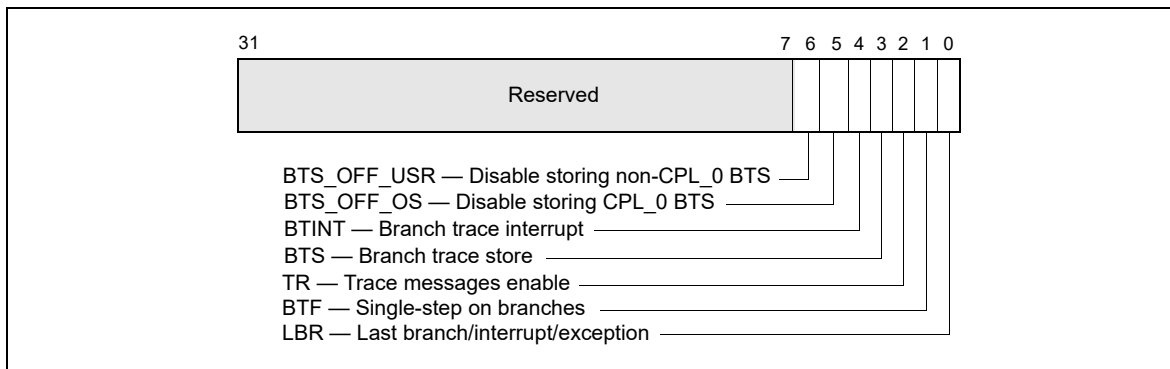


Figure 17-12. MSR\_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **BTS (branch trace store) flag (bit 3)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bits 4)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS).”
- **BTS\_OFF\_OS (disable ring 0 branch trace store) flag (bit 5)** — When set, enables the BTS facilities to skip sending/logging CPL\_0 BTMs to the memory-resident BTS buffer. See Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **BTS\_OFF\_USR (disable ring 0 branch trace store) flag (bit 6)** — When set, enables the BTS facilities to skip sending/logging non-CPL\_0 BTMs to the memory-resident BTS buffer. See Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”



**NOTE**

The initial implementation of BTS\_OFF\_USR and BTS\_OFF\_OS in MSR\_DEBUGCTLA is shown in Figure 17-12. The BTS\_OFF\_USR and BTS\_OFF\_OS fields may be implemented on other model-specific debug control register at different locations.

See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

**17.13.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture**

The LBR stack is made up of LBR MSRs that are treated by the processor as a circular stack. The TOS pointer (MSR\_LASTBRANCH\_TOS MSR) points to the LBR MSR (or LBR MSR pair) that contains the most recent (last) branch record placed on the stack. Prior to placing a new branch record on the stack, the TOS is incremented by 1. When the TOS pointer reaches its maximum value, it wraps around to 0. See Table 17-17 and Figure 17-12.

**Table 17-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family**

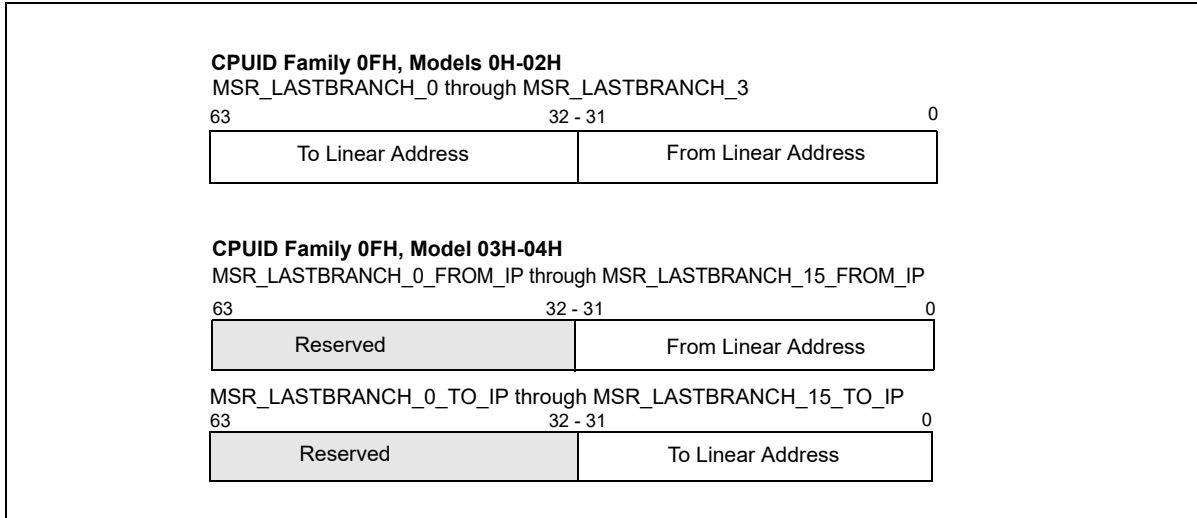
DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	4	0 to 3
Family 0FH, Models; MSRs at locations 680H-68FH.	16	0 to 15
Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	16	0 to 15

The registers in the LBR MSR stack and the MSR\_LASTBRANCH\_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 17-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the “from” and “to” instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the “from” address is the address of the branch instruction and the “to” address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the “from” address is the return instruction pointer (RIP) saved for the interrupt and the “to” address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the “from” address is the linear address of the instruction that caused the exception to be generated and the “to” address is the address of the first instruction in the exception handler routine.





**Figure 17-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family**

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

### 17.13.3 Last Exception Records

The Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors provide two MSRs (the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs) that duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors. The MSR\_LER\_TO\_LIP and MSR\_LER\_FROM\_LIP MSRs contain a branch record for the last branch that the processor took prior to an exception or interrupt being generated.

## 17.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

Intel Core Solo and Intel Core Duo processors provide last branch interrupt and exception recording. This capability is almost identical to that found in Pentium 4 and Intel Xeon processors. There are differences in the stack and in some MSR names and locations.

Note the following:

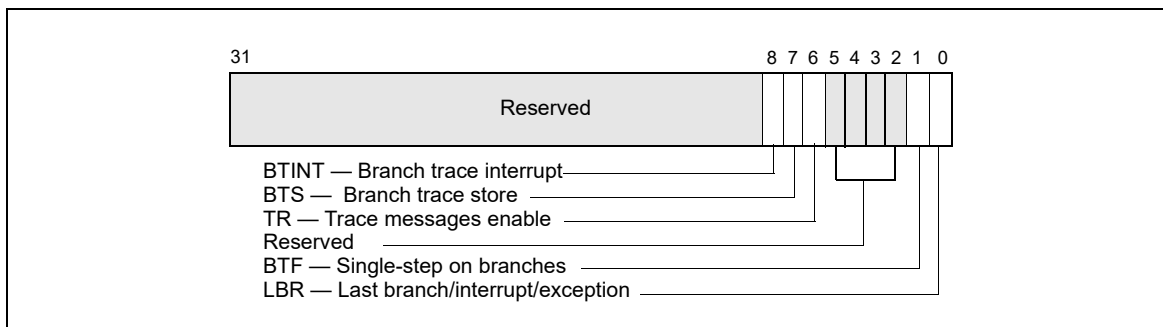
- **IA32\_DEBUGCTL MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. IA32\_DEBUGCTL MSR is located at register address 01D9H.

See Figure 17-14 for the layout and the entries below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the “Last Branch Record (LBR) Stack” below.
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism

allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.

- **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages,” for more information about the TR flag.
- **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.

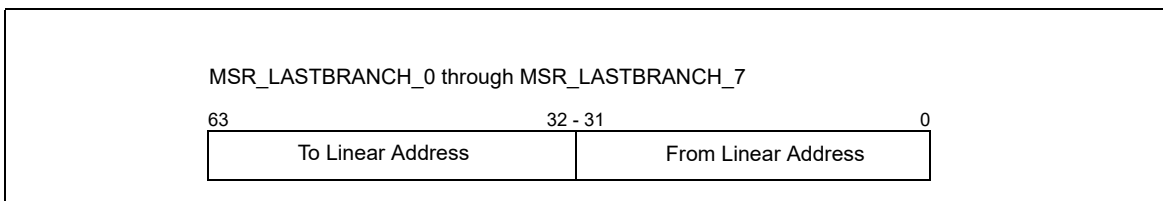


**Figure 17-14. IA32\_DEBUGCTL MSR for Intel Core Solo and Intel Core Duo Processors**

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, “Branch Trace Store (BTS).”
- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR\_LASTBRANCH\_0 through MSR\_LASTBRANCH\_7); bits 31-0 hold the ‘from’ address, bits 63-32 hold the ‘to’ address (MSR addresses start at 40H). See Figure 17-15.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Intel Core Solo and Intel Core Duo processors, this MSR is located at register address 01C9H.

For compatibility, the Intel Core Solo and Intel Core Duo processors provide two 32-bit MSRs (the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs) that duplicate functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

For details, see Section 17.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture,” and Section 2.20, “MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.



**Figure 17-15. LBR Branch Record Layout for the Intel Core Solo and Intel Core Duo Processor**

## 17.15 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS)

Like the Pentium 4 and Intel Xeon processor family, Pentium M processors provide last branch interrupt and exception recording. The capability operates almost identically to that found in Pentium 4 and Intel Xeon processors. There are differences in the shape of the stack and in some MSR names and locations. Note the following:

- **MSR\_DEBUGCTLB MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. For Pentium M processors, this MSR is located at register address 01D9H. See Figure 17-16 and the entries below for a description of the flags.
  - **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the “Last Branch Record (LBR) Stack” bullet below.
  - **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.
  - **PBi (performance monitoring/breakpoint pins) flags (bits 5-2)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BPi# pin when a breakpoint match occurs. When a PBi flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages,” for more information about the TR flag.
  - **BTS (branch trace store) flag (bit 7)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
  - **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.

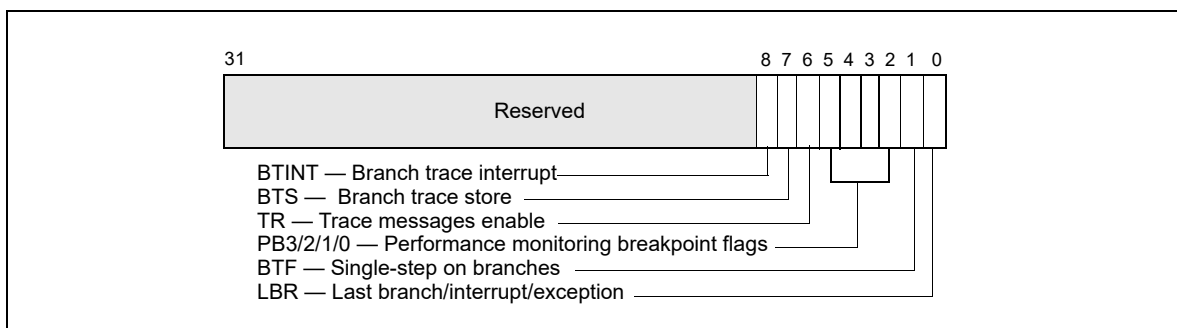


Figure 17-16. MSR\_DEBUGCTLB MSR for Pentium M Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, “Branch Trace Store (BTS).”

- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSR (MSR\_LASTBRANCH\_0 through MSR\_LASTBRANCH\_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 17-17.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.

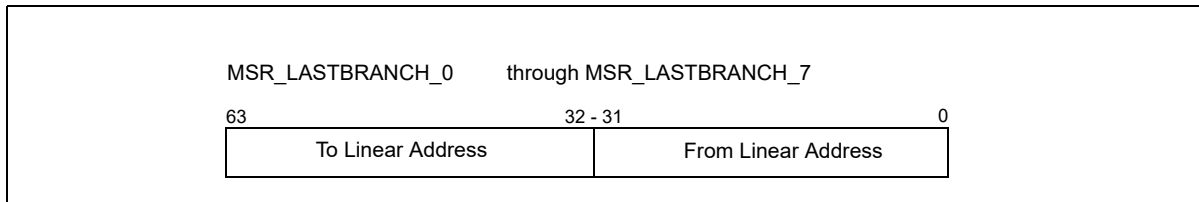


Figure 17-17. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 17.13.3, “Last Exception Records,” and Section 2.21, “MSRs In the Pentium M Processor” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

## 17.16 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (P6 FAMILY PROCESSORS)

The P6 family processors provide five MSRs for recording the last branch, interrupt, or exception taken by the processor: DEBUGCTLMSR, LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP. These registers can be used to collect last branch records, to set breakpoints on branches, interrupts, and exceptions, and to single-step from one branch to the next.

See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

### 17.16.1 DEBUGCTLMSR Register

The version of the DEBUGCTLMSR register found in the P6 family processors enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-18 shows the flags in the DEBUGCTLMSR register for the P6 family processors. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records the source and target addresses (in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs) for the last branch and the last exception or interrupt taken by the processor prior to a debug exception being generated. The processor clears this flag whenever a debug exception, such as an instruction or data breakpoint or single-step trap occurs.

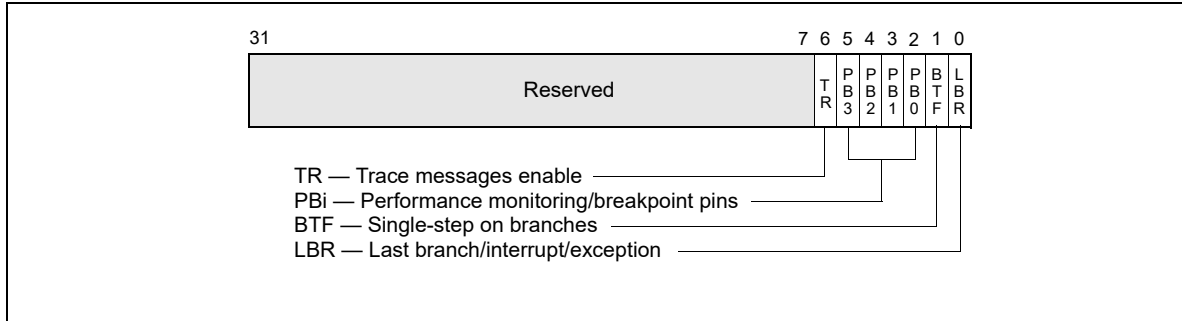


Figure 17-18. DEBUGCTLMR Register (P6 Family Processors)

- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag. See Section 17.4.3, “Single-Stepping on Branches.”
- **PBi (performance monitoring/breakpoint pins) flags (bits 2 through 5)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BPi# pin when a breakpoint match occurs. When a PBi flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
- **TR (trace message enable) flag (bit 6)** — When set, trace messages are enabled as described in Section 17.4.4, “Branch Trace Messages.” Setting this flag greatly reduces the performance of the processor. When trace messages are enabled, the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are undefined.

### 17.16.2 Last Branch and Last Exception MSRs

The LastBranchToIP and LastBranchFromIP MSRs are 32-bit registers for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. When a branch occurs, the processor loads the address of the branch instruction into the LastBranchFromIP MSR and loads the target address for the branch into the LastBranchToIP MSR.

When an interrupt or exception occurs (other than a debug exception), the address of the instruction that was interrupted by the exception or interrupt is loaded into the LastBranchFromIP MSR and the address of the exception or interrupt handler that is called is loaded into the LastBranchToIP MSR.

The LastExceptionToIP and LastExceptionFromIP MSRs (also 32-bit registers) record the instruction pointers for the last branch that the processor took prior to an exception or interrupt being generated. When an exception or interrupt occurs, the contents of the LastBranchToIP and LastBranchFromIP MSRs are copied into these registers before the to and from addresses of the exception or interrupt are recorded in the LastBranchToIP and LastBranchFromIP MSRs.

These registers can be read using the RDMSR instruction.

Note that the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into the current code segment, as opposed to linear addresses, which are saved in last branch records for the Pentium 4 and Intel Xeon processors.

### 17.16.3 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag in the DEBUGCTLMR register is set, the processor automatically begins recording branches that it takes, exceptions that are generated (except for debug exceptions), and interrupts that are serviced. Each time a branch, exception, or interrupt occurs, the processor records the to and from instruction pointers in the LastBranchToIP and LastBranchFromIP MSRs. In addition, for interrupts and exceptions, the processor copies the contents of the LastBranchToIP and LastBranchFromIP MSRs into the LastExceptionToIP and LastExceptionFromIP MSRs prior to recording the to and from addresses of the interrupt or exception.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler, but does not touch the last branch and last exception MSRs. The addresses for the last branch, interrupt, or exception taken are thus retained in the LastBranchToIP and LastBranchFromIP MSRs and the addresses of the last branch prior to an interrupt or exception are retained in the LastExceptionToIP, and LastExceptionFromIP MSRs.

The debugger can use the last branch, interrupt, and/or exception addresses in combination with code-segment selectors retrieved from the stack to reset breakpoints in the breakpoint-address registers (DR0 through DR3), allowing a backward trace from the manifestation of a particular bug toward its source. Because the instruction pointers recorded in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into a code segment, software must determine the segment base address of the code segment associated with the control transfer to calculate the linear address to be placed in the breakpoint-address registers. The segment base address can be determined by reading the segment selector for the code segment from the stack and using it to locate the segment descriptor for the segment in the GDT or LDT. The segment base address can then be read from the segment descriptor.

Before resuming program execution from a debug-exception handler, the handler must set the LBR flag again to re-enable last branch and last exception/interrupt recording.

## 17.17 TIME-STAMP COUNTER

The Intel 64 and IA-32 architectures (beginning with the Pentium processor) define a time-stamp counter mechanism that can be used to monitor and identify the relative time occurrence of processor events. The counter's architecture includes the following components:

- **TSC flag** — A feature bit that indicates the availability of the time-stamp counter. The counter is available in an if the function `CPUID.1:EDX.TSC[bit 4] = 1`.
- **IA32\_TIME\_STAMP\_COUNTER MSR** (called TSC MSR in P6 family and Pentium processors) — The MSR used as the counter.
- **RDTSC instruction** — An instruction used to read the time-stamp counter.
- **TSD flag** — A control register flag is used to enable or disable the time-stamp counter (enabled if `CR4.TSD[bit 2] = 1`).

The time-stamp counter (as implemented in the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter increments even when the processor is halted by the HLT instruction or the external STPCLK# pin. Note that the assertion of the external DPSLP# pin may cause the time-stamp counter to stop.

Processor families increment the time-stamp counter differently:

- For Pentium M processors (family [06H], models [09H, 0DH]); for Pentium 4 processors, Intel Xeon processors (family [0FH], models [00H, 01H, or 02H]); and for P6 family processors: the time-stamp counter increments with every internal processor clock cycle.

The internal processor clock cycle is determined by the current core-clock to bus-clock ratio. Intel® SpeedStep® technology transitions may also impact the processor clock.

- For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the processor or may be set by the maximum resolved frequency at which the processor is booted. The maximum resolved frequency may differ from the processor base frequency, see Section 18.7.2 for more detail. On certain processors, the TSC frequency may not be the same as the frequency in the brand string.

The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. This is the architectural behavior moving forward.

## NOTE

To determine average processor clock frequency, Intel recommends the use of performance monitoring logic to count processor core clocks over the period of time for which the average is required. See Section 18.6.4.5, “Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture,” and <https://perfmon-events.intel.com/> for more information.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound. Intel guarantees that the time-stamp counter will not wraparound within 10 years after being reset. The period for counter wrap is longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag allows use of this instruction to be restricted to programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions read and write the time-stamp counter, treating the time-stamp counter as an ordinary MSR (address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter are read using RDMSR (just as with RDTSC). When WRMSR is used to write the time-stamp counter on processors before family [0FH], models [03H, 04H]: only the low-order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family [0FH], models [03H, 04H, 06H]; for family [06H]], model [0EH, 0FH]; for family [06H]], DisplayModel [17H, 1AH, 1CH, 1DH]: all 64 bits are writable.

### 17.17.1 Invariant TSC

The time stamp counter in newer processors may support an enhancement, referred to as invariant TSC. Processor’s support for invariant TSC is indicated by CPUID.80000007H:EDX[8].

The invariant TSC will run at a constant rate in all ACPI P-, C-, and T-states. This is the architectural behavior moving forward. On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource.

### 17.17.2 IA32\_TSC\_AUX Register and RDTSCP Support

Processors based on Intel microarchitecture code name Nehalem provide an auxiliary TSC register, IA32\_TSC\_AUX that is designed to be used in conjunction with IA32\_TSC. IA32\_TSC\_AUX provides a 32-bit field that is initialized by privileged software with a signature value (for example, a logical processor ID).

The primary usage of IA32\_TSC\_AUX in conjunction with IA32\_TSC is to allow software to read the 64-bit time stamp in IA32\_TSC and signature value in IA32\_TSC\_AUX with the instruction RDTSCP in an atomic operation. RDTSCP returns the 64-bit time stamp in EDX:EAX and the 32-bit TSC\_AUX signature value in ECX. The atomicity of RDTSCP ensures that no context switch can occur between the reads of the TSC and TSC\_AUX values.

Support for RDTSCP is indicated by CPUID.80000001H:EDX[27]. As with RDTSC instruction, non-ring 0 access is controlled by CR4.TSD (Time Stamp Disable flag).

User mode software can use RDTSCP to detect if CPU migration has occurred between successive reads of the TSC. It can also be used to adjust for per-CPU differences in TSC values in a NUMA system.



### 17.17.3 Time-Stamp Counter Adjustment

Software can modify the value of the time-stamp counter (TSC) of a logical processor by using the WRMSR instruction to write to the IA32\_TIME\_STAMP\_COUNTER MSR (address 10H). Because such a write applies only to that logical processor, software seeking to synchronize the TSC values of multiple logical processors must perform these writes on each logical processor. It may be difficult for software to do this in a way that ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32\_TSC\_ADJUST MSR (address 3BH). Like the IA32\_TIME\_STAMP\_COUNTER MSR, the IA32\_TSC\_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32\_TSC\_ADJUST MSR as follows:

- On RESET, the value of the IA32\_TSC\_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32\_TIME\_STAMP\_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32\_TSC\_ADJUST MSR.
- If an execution of WRMSR to the IA32\_TSC\_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32\_TSC\_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32\_TIME\_STAMP\_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32\_TSC\_ADJUST MSR on each logical processor.

Processor support for the IA32\_TSC\_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC\_ADJUST (bit 1).

### 17.17.4 Invariant Time-Keeping

The invariant TSC is based on the invariant timekeeping hardware (called Always Running Timer or ART), that runs at the core crystal clock frequency. The ratio defined by CPUID leaf 15H expresses the frequency relationship between the ART hardware and TSC.

If CPUID.15H:EBX[31:0] != 0 and CPUID.80000007H:EDX[InvariantTSC] = 1, the following linearity relationship holds between TSC and the ART hardware:

$$\text{TSC\_Value} = (\text{ART\_Value} * \text{CPUID.15H:EBX}[31:0]) / \text{CPUID.15H:EAX}[31:0] + K$$

Where 'K' is an offset that can be adjusted by a privileged agent<sup>2</sup>.

When ART hardware is reset, both invariant TSC and K are also reset.

## 17.18 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) MONITORING FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of monitoring capabilities including Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM). The Intel® Xeon® processor E5 v3 family introduced resource monitoring capability in each logical processor to measure specific platform shared resource metrics, for example, L3 cache occupancy. The programming interface for these monitoring features is described in this section. Two features within the monitoring feature set provided are described - Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

Cache Monitoring Technology (CMT) allows an Operating System, Hypervisor or similar system management agent to determine the usage of cache by applications running on the platform. The initial implementation is directed at L3 cache monitoring (currently the last level cache in most server platforms).

Memory Bandwidth Monitoring (MBM), introduced in the Intel® Xeon® processor E5 v4 family, builds on the CMT infrastructure to allow monitoring of bandwidth from one level of the cache hierarchy to the next - in this case

2. IA32\_TSC\_ADJUST MSR and the TSC-offset field in the VM execution controls of VMCS are some of the common interfaces that privileged software can use to manage the time stamp counter for keeping time



focusing on the L3 cache, which is typically backed directly by system memory. As a result of this implementation, memory bandwidth can be monitored.

The monitoring mechanisms described provide the following key shared infrastructure features:

- A mechanism to enumerate the presence of the monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for the OS or Hypervisor to indicate a software-defined ID for each of the software threads (applications, virtual machines, etc.) that are scheduled to run on a logical processor. These identifiers are known as Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for the OS or Hypervisor to read back the collected metrics such as L3 occupancy or Memory Bandwidth for a given software ID at any point during runtime.

### 17.18.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring

The shared resource monitoring features described in this chapter provide a layer of abstraction between applications and logical processors through the use of **Resource Monitoring IDs** (RMIDs). Each logical processor in the system can be assigned an RMID independently, or multiple logical processors can be assigned to the same RMID value (e.g., to track an application with multiple threads). For each logical processor, only one RMID value is active at a time. This is enforced by the IA32\_PQR\_ASSOC MSR, which specifies the active RMID of a logical processor. Writing to this MSR by software changes the active RMID of the logical processor from an old value to a new value.

The underlying platform shared resource monitoring hardware tracks cache metrics such as cache utilization and misses as a result of memory accesses according to the RMIDs and reports monitored data via a counter register (IA32\_QM\_CTR). The specific event types supported vary by generation and can be enumerated via CPUID. Before reading back monitored data software must configure an event selection MSR (IA32\_QM\_EVTSEL) to specify which metric is to be reported, and the specific RMID for which the data should be returned.

Processor support of the monitoring framework and sub-features such as CMT is reported via the CPUID instruction. The resource type available to the monitoring framework is enumerated via a new leaf function in CPUID. Reading and writing to the monitoring MSRs requires the RDMSR and WRMSR instructions.

The Cache Monitoring Technology feature set provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the CMT feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- CMT-specific event codes to read occupancy for a given level of the cache hierarchy.

The Memory Bandwidth Monitoring feature provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the MBM feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- MBM-specific event codes to read bandwidth out to the next level of the hierarchy and various sub-event codes to read more specific metrics as discussed later (e.g., total bandwidth vs. bandwidth only from local memory controllers on the same package).

### 17.18.2 Enabling Monitoring: Usage Flow

Figure 17-19 illustrates the key steps for OS/VMM to detect support of shared resource monitoring features such as CMT and enable resource monitoring for available resource types and monitoring events.

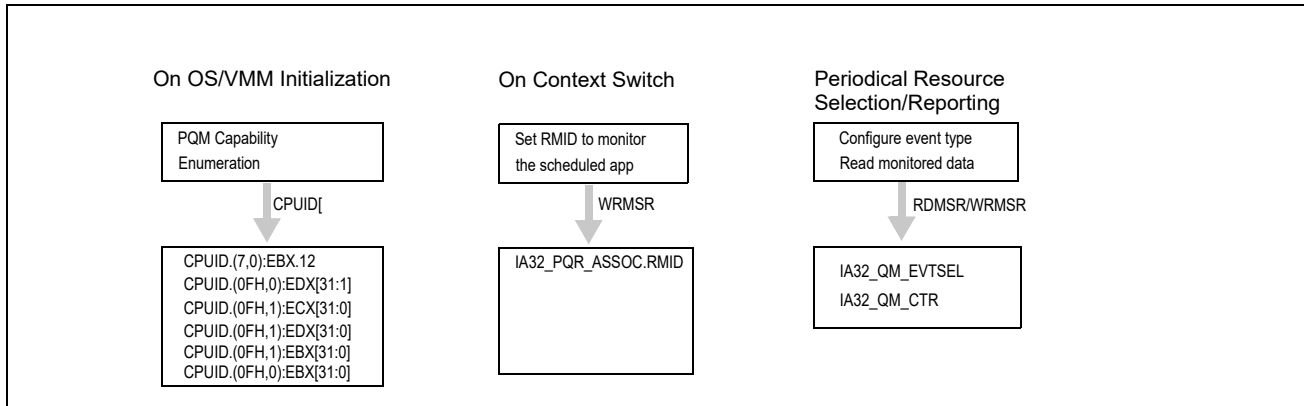


Figure 17-19. Platform Shared Resource Monitoring Usage Flow

### 17.18.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring

Software can query processor support of shared resource monitoring features capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] reports 1, the processor provides the following programming interfaces for shared resource monitoring, including Cache Monitoring Technology:

- CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides information on available resource types (see Section 17.18.4), and monitoring capabilities for each resource type (see Section 17.18.5). Note CMT and MBM capabilities are enumerated as separate event vectors using shared enumeration infrastructure under a given resource type.
- IA32\_PQR\_ASSOC.RMID: The per-logical-processor MSR, IA32\_PQR\_ASSOC, that OS/VMM can use to assign an RMID to each logical processor, see Section 17.18.6.
- IA32\_QM\_EVTSEL: This MSR specifies an Event ID (EvtID) and an RMID which the platform uses to look up and provide monitoring data in the monitoring counter, IA32\_QM\_CTR, see Section 17.18.7.
- IA32\_QM\_CTR: This MSR reports monitored resource data when available along with bits to allow software to check for error conditions and verify data validity.

Software must follow the following sequence of enumeration to discover Cache Monitoring Technology capabilities:

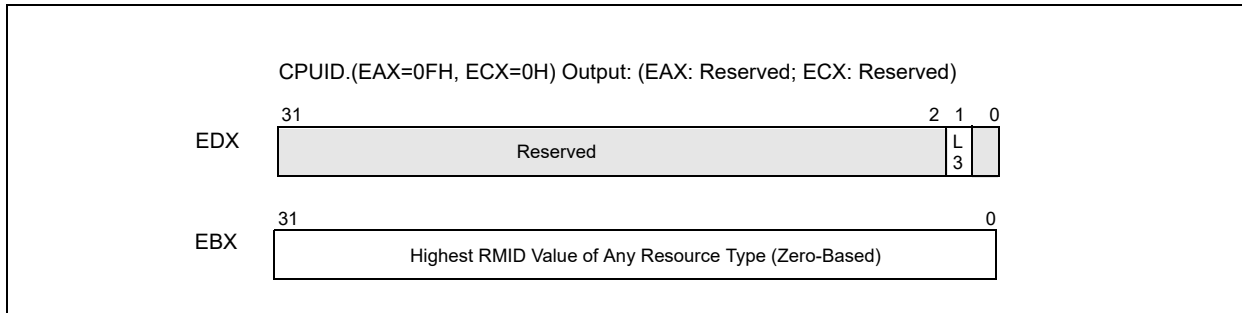
1. Execute CPUID with EAX=0 to discover the "cpuid\_maxLeaf" supported in the processor;
2. If cpuid\_maxLeaf >= 7, then execute CPUID with EAX=7, ECX= 0 to verify CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] is set;
3. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] = 1, then execute CPUID with EAX=0FH, ECX= 0 to query available resource types that support monitoring;
4. If CPUID.(EAX=0FH, ECX=0):EDX.L3[bit 1] = 1, then execute CPUID with EAX=0FH, ECX= 1 to query the specific capabilities of L3 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.
5. If CPUID.(EAX=0FH, ECX=0):EDX reports additional resource types supporting monitoring, then execute CPUID with EAX=0FH, ECX set to a corresponding resource type ID (ResID) as enumerated by the bit position of CPUID.(EAX=0FH, ECX=0):EDX.

### 17.18.4 Monitoring Resource Type and Capability Enumeration

CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides one sub-leaf (sub-function 0) that reports shared enumeration infrastructure, and one or more sub-functions that report feature-specific enumeration data:

- Monitoring leaf sub-function 0 enumerates available resources that support monitoring, i.e. executing CPUID with EAX=0FH and ECX=0H. In the initial implementation, L3 cache is the only resource type available. Each

supported resource type is represented by a bit in CPUID.(EAX=0FH, ECX=0):EDX[31:1]. The bit position corresponds to the sub-leaf index (ResID) that software must use to query details of the monitoring capability of that resource type (see Figure 17-21 and Figure 17-22). Reserved bits of CPUID.(EAX=0FH, ECX=0):EDX[31:2] correspond to unsupported sub-leaves of the CPUID.0FH leaf. Additionally, CPUID.(EAX=0FH, ECX=0H):EBX reports the highest RMID value of any resource type that supports monitoring in the processor.

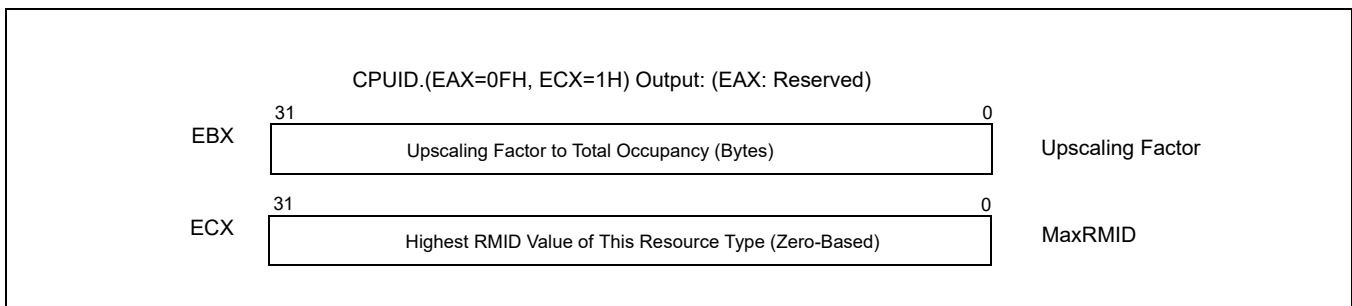


**Figure 17-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration**

### 17.18.5 Feature-Specific Enumeration

Each additional sub-leaf of CPUID.(EAX=0FH, ECX=ResID) enumerates the specific details for software to program Monitoring MSRs using the resource type associated with the given ResID.

Note that in future Monitoring implementations the meanings of the returned registers may vary in other sub-leaves that are not yet defined. The registers will be specified and defined on a per-ResID basis.



**Figure 17-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H))**

For each supported Cache Monitoring resource type, hardware supports only a finite number of RMIDs. CPUID.(EAX=0FH, ECX=1H).ECX enumerates the highest RMID value that can be monitored with this resource type, see Figure 17-21.

CPUID.(EAX=0FH, ECX=1H).EDX specifies a bit vector that is used to look up the EventID (See Figure 17-22 and Table 17-18) that software must program with IA32\_QM\_EVTSEL in order to retrieve event data. After software configures IA32\_QMEVTSEL with the desired RMID and EventID, it can read the resulting data from IA32\_QM\_CTR. The raw numerical value reported from IA32\_QM\_CTR can be converted to the final value (occupancy in bytes or bandwidth in bytes per sampled time period) by multiplying the counter value by the value from CPUID.(EAX=0FH, ECX=1H).EBX, see Figure 17-21.

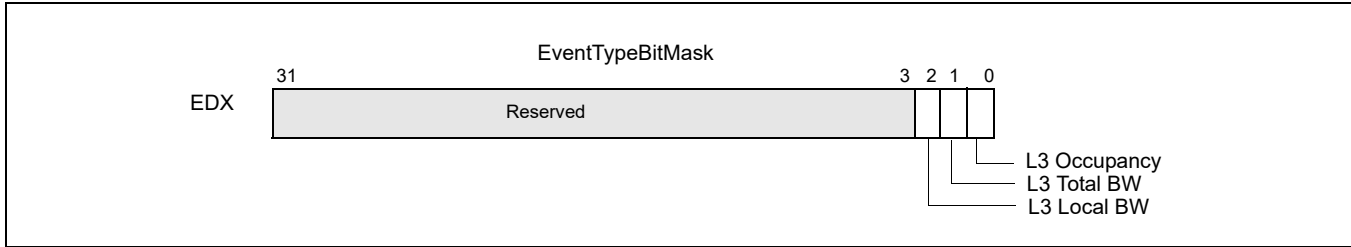


Figure 17-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H) )

### 17.18.5.1 Cache Monitoring Technology

On processors for which Cache Monitoring Technology supports the L3 cache occupancy event, CPUID.(EAX=0FH, ECX=1H).EDX would return with only bit 0 set. The corresponding event ID can be looked up from Table 17-18. The L3 occupancy data accumulated in IA32\_QM\_CTR can be converted to total occupancy (in bytes) by multiplying with CPUID.(EAX=0FH, ECX=1H).EBX.

Event codes for Cache Monitoring Technology are discussed in the next section.

### 17.18.5.2 Memory Bandwidth Monitoring

On processors that monitoring supports Memory Bandwidth Monitoring using ResID=1 (L3), two additional bits will be set in the vector at CPUID.(EAX=0FH, ECX=1H).EDX:

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 1]: indicates the L3 total external bandwidth monitoring event is supported if set. This event monitors the L3 total external bandwidth to the next level of the cache hierarchy, including all demand and prefetch misses from the L3 to the next hierarchy of the memory system. In most platforms, this represents memory bandwidth.
- CPUID.(EAX=0FH, ECX=1H).EDX[bit 2]: indicates L3 local memory bandwidth monitoring event is supported if set. This event monitors the L3 external bandwidth satisfied by the local memory. In most platforms that support this event, L3 requests are likely serviced by a memory system with non-uniform memory architecture. This allows bandwidth to off-package memory resources to be tracked by subtracting local from total bandwidth (for instance, bandwidth over QPI to a memory controller on another physical processor could be tracked by subtraction).

The corresponding Event ID can be looked up from Table 17-18. The L3 bandwidth data accumulated in IA32\_QM\_CTR can be converted to total bandwidth (in bytes) using CPUID.(EAX=0FH, ECX=1H).EBX.

Table 17-18. Monitoring Supported Event IDs

Event Type	Event ID	Context
L3 Cache Occupancy	01H	Cache Monitoring Technology
L3 Total External Bandwidth	02H	MBM
L3 Local External Bandwidth	03H	MBM
Reserved	All other event codes	N/A

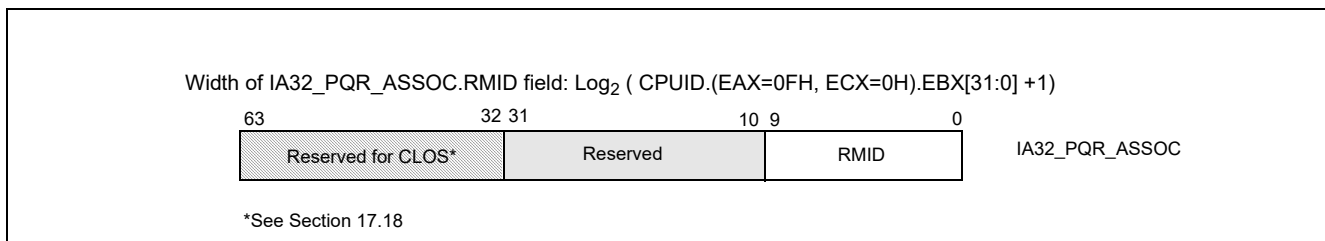
### 17.18.6 Monitoring Resource RMID Association

After Monitoring and sub-features has been enumerated, software can begin using the monitoring features. The first step is to associate a given software thread (or multiple threads as part of an application, VM, group of applications or other abstraction) with an RMID.

Note that the process of associating an RMID with a given software thread is the same for all shared resource monitoring features (CMT, MBM), and a given RMID number has the same meaning from the viewpoint of any logical processors in a package. Stated another way, a thread may be associated in a 1:1 mapping with an RMID, and that

RMID may allow cache occupancy, memory bandwidth information or other monitoring data to be read back later with monitoring event codes (retrieving data is discussed in a previous section).

The association of an application thread with an RMID requires an OS to program the per-logical-processor MSR IA32\_PQR\_ASSOC at context swap time (updates may also be made at any other arbitrary points during program execution such as application phase changes). The IA32\_PQR\_ASSOC MSR specifies the active RMID that monitoring hardware will use to tag internal operations, such as L3 cache requests. The layout of the MSR is shown in Figure 17-23. Software specifies the active RMID to monitor in the IA32\_PQR\_ASSOC.RMID field. The width of the RMID field can vary from one implementation to another, and is derived from  $\text{Ceil}(\log_2(1 + \text{CPUID}(\text{EAX}=0\text{FH}, \text{ECX}=0\text{H}).\text{EBX}[31:0]))$ . The value of IA32\_PQR\_ASSOC after power-on is 0.



**Figure 17-23. IA32\_PQR\_ASSOC MSR**

In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32\_PQR\_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g. RMID 31) may only be supported by one resource type but not by another resource type.

### 17.18.7 Monitoring Resource Selection and Reporting Infrastructure

The reporting mechanism for Cache Monitoring Technology and other related features is architecturally exposed as an MSR pair that can be programmed and read to measure various metrics such as the L3 cache occupancy (CMT) and bandwidths (MBM) depending on the level of Monitoring support provided by the platform. Data is reported back on a per-RMID basis. These events do not trigger based on event counts or trigger APIC interrupts (e.g. no Performance Monitoring Interrupt occurs based on counts). Rather, they are used to sample counts explicitly.

The MSR pair for the shared resource monitoring features (CMT, MBM) is separate from and not shared with architectural Perfmon counters, meaning software can use these monitoring features simultaneously with the Perfmon counters.

Access to the aggregated monitoring information is accomplished through the following programmable monitoring MSRs:

- IA32\_QM\_EVTSEL: This MSR provides a role similar to the event select MSRs for programmable performance monitoring described in Chapter 18. The simplified layout of the MSR is shown in Figure 17-24. Bits IA32\_QM\_EVTSEL.EvtID (bits 7:0) specify an event code of a supported resource type for hardware to report monitored data associated with IA32\_QM\_EVTSEL.RMID (bits 41:32). Software can configure IA32\_QM\_EVTSEL.RMID with any RMID that is active within the physical processor. The width of IA32\_QM\_EVTSEL.RMID matches that of IA32\_PQR\_ASSOC.RMID. Supported event codes for the IA32\_QM\_EVTSEL register are shown in Table 17-18. Note that valid event codes may not necessarily map directly to the bit position used to enumerate support for the resource via CPUID.

Software can program an RMID / Event ID pair into the IA32\_QM\_EVTSEL MSR bit field to select an RMID to read a particular counter for a given resource. The currently supported list of Monitoring Event IDs is discussed in Section 17.18.5, which covers feature-specific details.

Thread access to the IA32\_QM\_EVTSEL and IA32\_QM\_CTR MSR pair should be serialized (that is, treated as a critical section under lock) to avoid situations where one thread changes the RMID/EvtID just before another thread reads monitoring data from IA32\_QM\_CTR.

- IA32\_QM\_CTR: This MSR reports monitored data when available. It contains three bit fields. If software configures an unsupported RMID or event type in IA32\_QM\_EVTSEL, then IA32\_QM\_CTR.Error (bit 63) will be set, indicating there is no valid data to report. If IA32\_QM\_CTR.Unavailable (bit 62) is set, it indicates monitored data for the RMID is not available, and IA32\_QM\_CTR.data (bits 61:0) should be ignored. Therefore, IA32\_QM\_CTR.data (bits 61:0) is valid only if bit 63 and 62 are both clear. For Cache Monitoring Technology, software can convert IA32\_QM\_CTR.data into cache occupancy or bandwidth metrics expressed in bytes by multiplying with the conversion factor from CPUID.(EAX=0FH, ECX=1H).EBX.

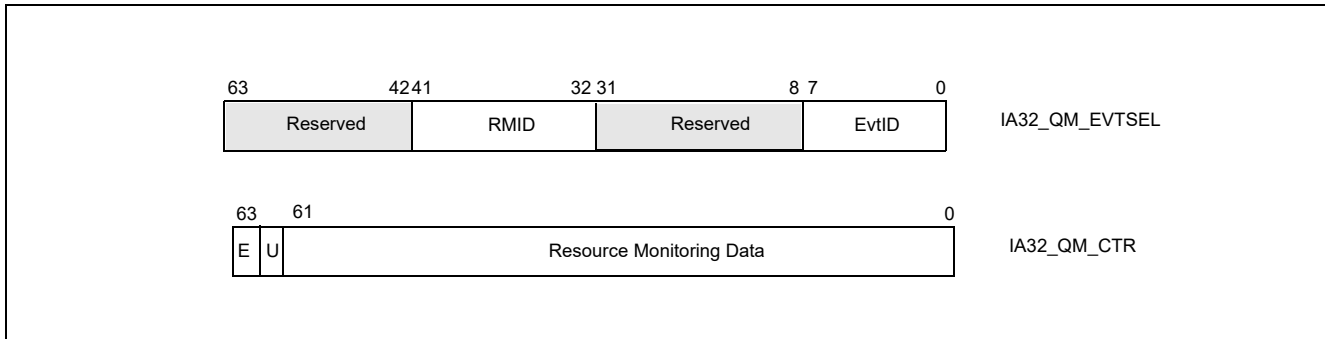


Figure 17-24. IA32\_QM\_EVTSEL and IA32\_QM\_CTR MSRs

### 17.18.8 Monitoring Programming Considerations

Figure 17-23 illustrates how system software can program IA32\_QOSEVTSEL and IA32\_QM\_CTR to perform resource monitoring.

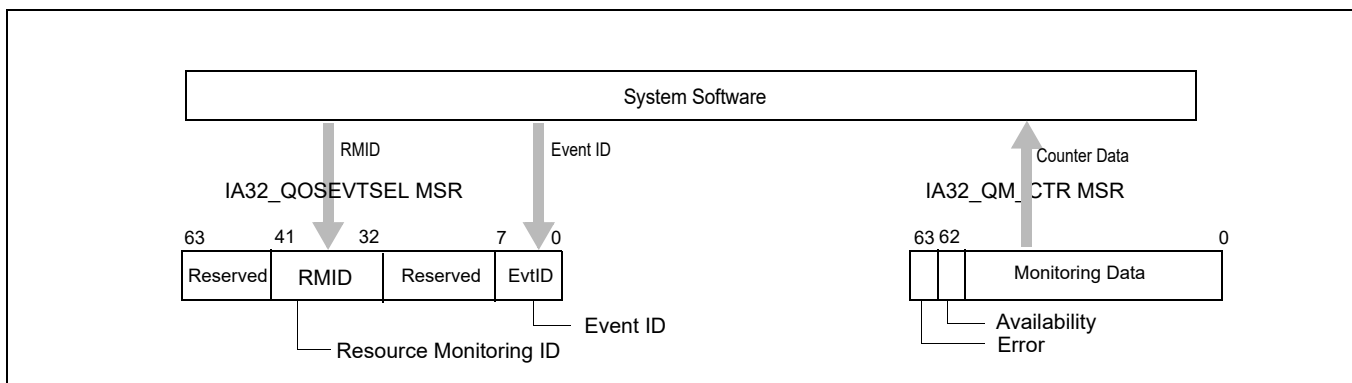


Figure 17-25. Software Usage of Cache Monitoring Resources

Though the field provided in IA32\_QM\_CTR allows for up to 62 bits of data to be returned, often a subset of bits are used. With Cache Monitoring Technology for instance, the number of bits used will be proportional to the base-two logarithm of the total cache size divided by the Upscaling Factor from CPUID.

In Memory Bandwidth Monitoring the initial counter size is 24 bits, and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period. Any future changes to counter width will be enumerated to software.

### 17.18.8.1 Monitoring Dynamic Configuration

Both the IA32\_QM\_EVTSEL and IA32\_PQR\_ASSOC registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- An RMID exceeding the maxRMID is used.

### 17.18.8.2 Monitoring Operation With Power Saving Features

Note that some advanced power management features such as deep package C-states may shrink the L3 cache and cause CMT occupancy count to be reduced. MBM bandwidth counts may increase due to flushing cached data out of L3.

### 17.18.8.3 Monitoring Operation with Other Operating Modes

The states in IA32\_PQR\_ASSOC and monitoring counter are unmodified across an SMI delivery. Thus, the execution of SMM handler code and SMM handler's data can manifest as spurious contribution in the monitored data.

It is possible for an SMM handler to minimize the impact on of spurious contribution in the QOS monitoring counters by reserving a dedicated RMID for monitoring the SMM handler. Such an SMM handler can save the previously configured QOS Monitoring state immediately upon entering SMM, and restoring the QOS monitoring state back to the prev-SMM RMID upon exit.

### 17.18.8.4 Monitoring Operation with RAS Features

In general the Reliability, Availability and Serviceability (RAS) features present in Intel Platforms are not expected to significantly affect shared resource monitoring counts. In cases where software RAS features cause memory copies or cache accesses these may be tracked and may influence the shared resource monitoring counter values.

## 17.19 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of allocation (resource control) capabilities including Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP). The Intel Xeon processor E5 v4 family (and a subset of communication-focused processors in the Intel Xeon E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Certain Intel Atom processors also provide support for control over the L2 cache, with capabilities as described below. The programming interface for Cache Allocation Technology and for the more general allocation capabilities are described in the rest of this chapter. The CAT and CDP capabilities, where architecturally supported, may be detected and enumerated in software using the *CPUID* instruction, as described in this chapter.

The Intel Xeon Processor Scalable Family introduces the Memory Bandwidth Allocation (MBA) feature which provides indirect control over the memory bandwidth available to CPU cores, and is discussed later in this chapter.

### 17.19.1 Introduction to Cache Allocation Technology (CAT)

Cache Allocation Technology enables an Operating System (OS), Hypervisor /Virtual Machine Manager (VMM) or similar system service management agent to specify the amount of cache space into which an application can fill (as a hint to hardware - certain features such as power management may override CAT settings). Specialized user-level implementations with minimal OS support are also possible, though not necessarily recommended (see notes below for OS/Hypervisor with respect to ring 3 software and virtual guests). Depending on the processor family, L2 or L3 cache allocation capability may be provided, and the technology is designed to scale across multiple cache levels and technology generations.



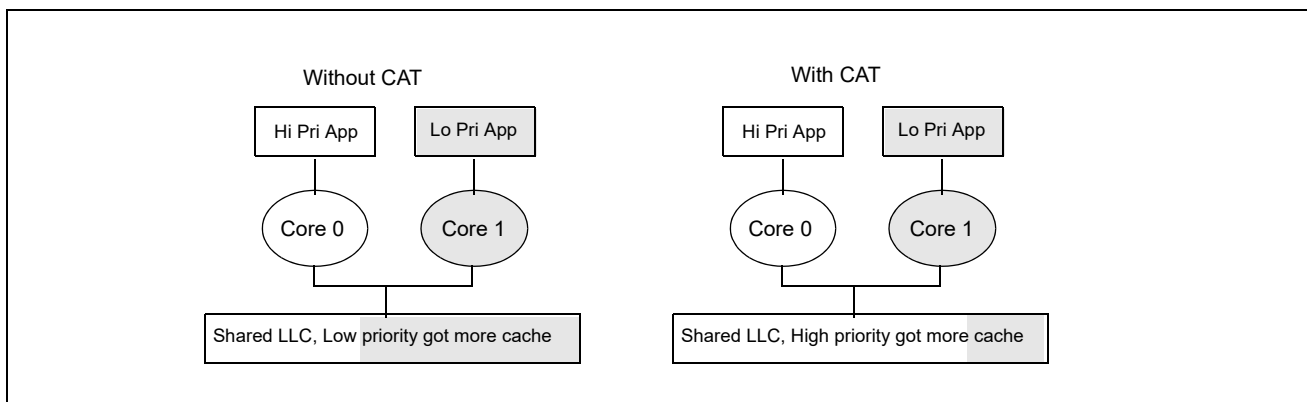
Software can determine which levels are supported in a given platform programmatically using CPUID as described in the following sections.

The CAT mechanisms defined in this document provide the following key features:

- A mechanism to enumerate platform Cache Allocation Technology capabilities and available resource types that provides CAT control capabilities. For implementations that support Cache Allocation Technology, CPUID provides enumeration support to query which levels of the cache hierarchy are supported and specific CAT capabilities, such as the max allocation bitmask size,
- A mechanism for the OS or Hypervisor to configure the amount of a resource available to a particular Class of Service via a list of allocation bitmasks,
- Mechanisms for the OS or Hypervisor to signal the Class of Service to which an application belongs, and
- Hardware mechanisms to guide the LLC fill policy when an application has been designated to belong to a specific Class of Service.

Note that for many usages, an OS or Hypervisor may not want to expose Cache Allocation Technology mechanisms to Ring3 software or virtualized guests.

The Cache Allocation Technology feature enables more cache resources (i.e. cache space) to be made available for high priority applications based on guidance from the execution environment as shown in Figure 17-26. The architecture also allows dynamic resource reassignment during runtime to further optimize the performance of the high priority application with minimal degradation to the low priority app. Additionally, resources can be rebalanced for system throughput benefit across uses cases of OSES, VMMs, containers and other scenarios by managing the CPUID and MSR interfaces. This section describes the hardware and software support required in the platform including what is required of the execution environment (i.e. OS/VMM) to support such resource control. Note that in Figure 17-26 the L3 Cache is shown as an example resource.



**Figure 17-26. Cache Allocation Technology Enables Allocation of More Resources to High Priority Applications**

### 17.19.2 Cache Allocation Technology Architecture

The fundamental goal of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (COS or CLOS). The processor exposes a set of Classes of Service into which applications (or individual threads) can be assigned. Cache allocation for the respective applications or threads is then restricted based on the class with which they are associated. Each Class of Service can be configured using capacity bitmasks (CBMs) which represent capacity and indicate the degree of overlap and isolation between classes. For each logical processor there is a register exposed (referred to here as the IA32\_PQR\_ASSOC MSR or PQR) to allow the OS/VMM to specify a COS when an application, thread or VM is scheduled.

The usage of Classes of Service (COS) are consistent across resources and a COS may have multiple resource control attributes attached, which reduces software overhead at context swap time. Rather than adding new types of COS tags per resource for instance, the COS management overhead is constant. Cache allocation for the indicated application/thread/container/VM is then controlled automatically by the hardware based on the class and the bitmask associated with that class. Bitmasks are configured via the IA32\_resourceType\_MASK\_n MSRs, where resourceType indicates a resource type (e.g. "L3" for the L3 cache) and "n" indicates a COS number.

The basic ingredients of Cache Allocation Technology are as follows:



- An architecturally exposed mechanism using CPUID to indicate whether CAT is supported, and what resource types are available which can be controlled,
- For each available resourceType, CPUID also enumerates the total number of Classes of Services and the length of the capacity bitmasks that can be used to enforce cache allocation to applications on the platform,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to configure the behavior of different classes of service using the bitmasks available,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to assign a COS to an executing software thread (i.e. associating the active CR3 of a logical processor with the COS in IA32\_PQR\_ASSOC),
- Implementation-dependent mechanisms to indicate which COS is associated with a memory access and to enforce the cache allocation on a per COS basis.

A capacity bitmask (CBM) provides a hint to the hardware indicating the cache space an application should be limited to as well as providing an indication of overlap and isolation in the CAT-capable cache from other applications contending for the cache. The bit length of the capacity mask available generally depends on the configuration of the cache and is specified in the enumeration process for CAT in CPUID (this may vary between models in a processor family as well). Similarly, other parameters such as the number of supported COS may vary for each resource type, and these details can be enumerated via CPUID.

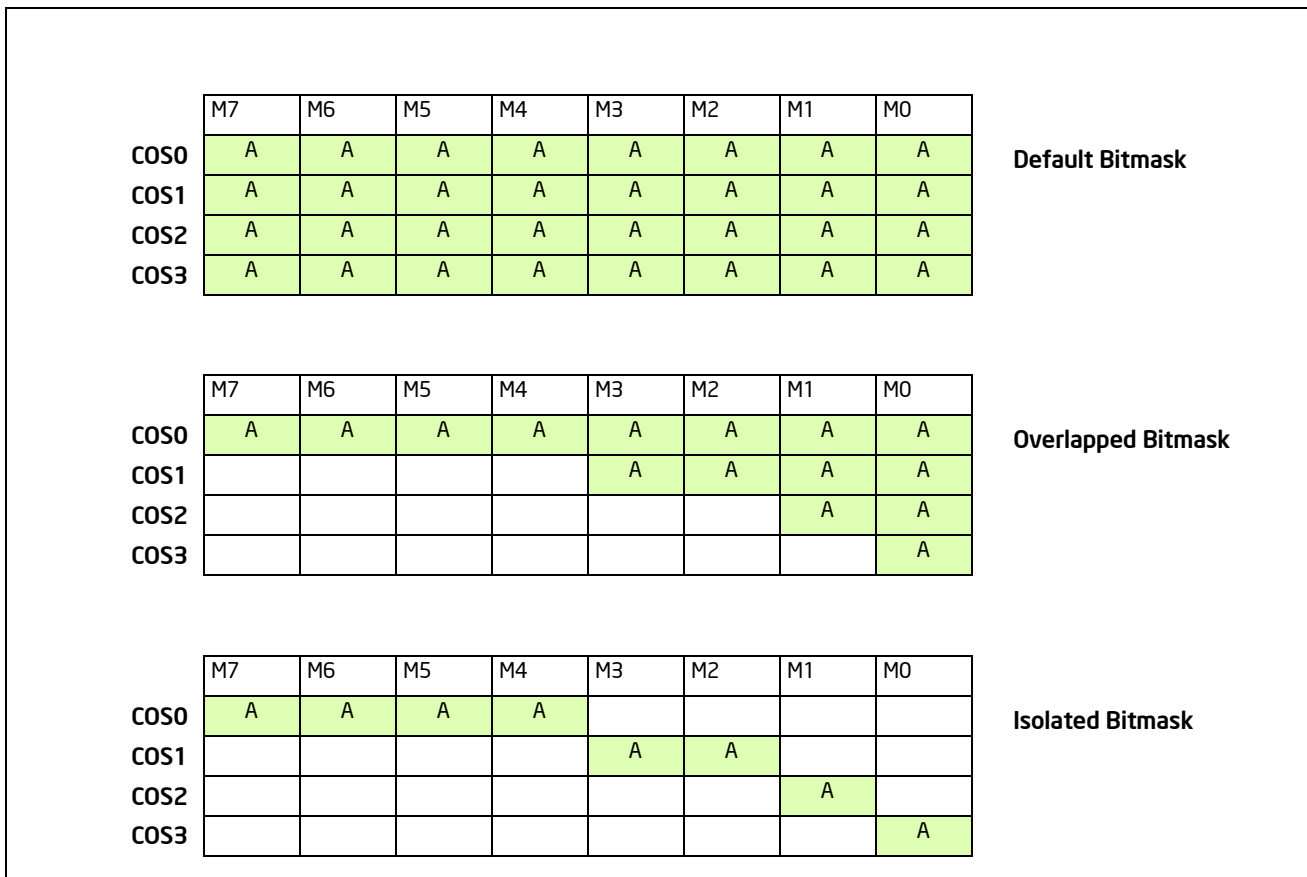


Figure 17-27. Examples of Cache Capacity Bitmasks

Sample cache capacity bitmasks for a bit length of 8 are shown in Figure 17-27. Please note that all (and only) contiguous '1' combinations are allowed (e.g. FFFFH, 0FF0H, 003CH, etc.). Attempts to program a value without contiguous '1's (including zero) will result in a general protection fault (#GP(0)). It is generally expected that in way-based implementations, one capacity mask bit corresponds to some number of ways in cache, but the specific mapping is implementation-dependent. In all cases, a mask bit set to '1' specifies that a particular Class of Service can allocate into the cache subset represented by that bit. A value of '0' in a mask bit specifies that a Class of

Service cannot allocate into the given cache subset. In general, allocating more cache to a given application is usually beneficial to its performance.

Figure 17-27 also shows three examples of sets of Cache Capacity Bitmasks. For simplicity these are represented as 8-bit vectors, though this may vary depending on the implementation and how the mask is mapped to the available cache capacity. The first example shows the default case where all 4 Classes of Service (the total number of COS are implementation-dependent) have full access to the cache. The second case shows an overlapped case, which would allow some lower-priority threads share cache space with the highest priority threads. The third case shows various non-overlapped partitioning schemes. As a matter of software policy for extensibility COS0 should typically be considered and configured as the highest priority COS, followed by COS1, and so on, though there is no hardware restriction enforcing this mapping. When the system boots all threads are initialized to COS0, which has full access to the cache by default.

Though the representation of the CBMs looks similar to a way-based mapping they are independent of any specific enforcement implementation (e.g. way partitioning.) Rather, this is a convenient manner to represent capacity, overlap and isolation of cache space. For example, executing a *POPCNT* instruction (population count of set bits) on the capacity bitmask can provide the fraction of cache space that a class of service can allocate into. In addition to the fraction, the exact location of the bits also shows whether the class of service overlaps with other classes of service or is entirely isolated in terms of cache space used.

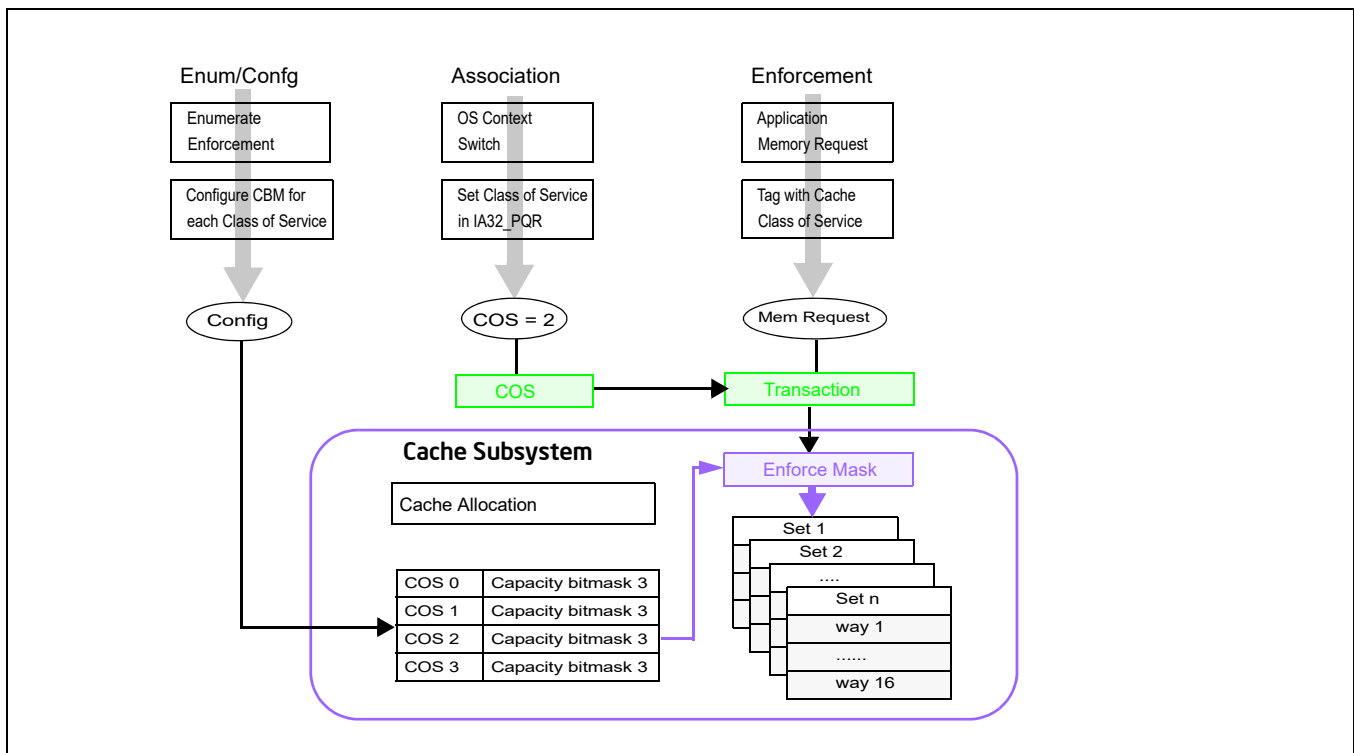


Figure 17-28. Class of Service and Cache Capacity Bitmasks

Figure 17-28 shows how the Cache Capacity Bitmasks and the per-logical-processor Class of Service are logically used to enable Cache Allocation Technology. All (and only) contiguous 1's in the CBM are permitted. The length of a CBM may vary from resource to resource or between processor generations and can be enumerated using CPUID. From the available mask set and based on the goals of the OS/VMM (shared or isolated cache, etc.) bitmasks are selected and associated with different classes of service. For the available Classes of Service the associated CBMs can be programmed via the global set of CAT configuration registers (in the case of L3 CAT, via the IA32\_L3\_MASK\_n MSRs, where "n" is the Class of Service, starting from zero). In all architectural implementations supporting CPUID it is possible to change the CBMs dynamically, during program execution, unless stated otherwise by Intel.

The currently running application's Class of Service is communicated to the hardware through the per-logical-processor PQR MSR (IA32\_PQR\_ASSOC MSR). When the OS schedules an application thread on a logical processor,

the application thread is associated with a specific COS (i.e. the corresponding COS in the PQR) and all requests to the CAT-capable resource from that logical processor are tagged with that COS (in other words, the application thread is configured to belong to a specific COS). The cache subsystem uses this tagged request information to enforce QoS. The capacity bitmask may be mapped into a way bitmask (or a similar enforcement entity based on the implementation) at the cache before it is applied to the allocation policy. For example, the capacity bitmask can be an 8-bit mask and the enforcement may be accomplished using a 16-way bitmask for a cache enforcement implementation based on way partitioning.

The following sections describe extensions of CAT such as Code and Data Prioritization (CDP), followed by details on specific features such as L3 CAT, L3 CDP, L2 CAT, and L2 CDP. Depending on the specific processor a mix of features may be supported, and CPUID provides enumeration capabilities to enable software to dynamically detect the set of supported features.

### 17.19.3 Code and Data Prioritization (CDP) Technology

Code and Data Prioritization Technology is an extension of CAT. CDP enables isolation and separate prioritization of code and data fetches to the L2 or L3 cache in a software configurable manner, depending on hardware support, which can enable workload prioritization and tuning of cache capacity to the characteristics of the workload. CDP extends Cache Allocation Technology (CAT) by providing separate code and data masks per Class of Service (COS). Support for the L2 CDP feature and the L3 CDP features are separately enumerated (via CPUID) and separately controlled (via remapping the L2 CAT MSRs or L3 CAT MSRs respectively). Section 17.19.6.3 and Section 17.19.7 provide details on enumerating, controlling and enabling L3 and L2 CDP respectively, while this section provides a general overview.

The L3 CDP feature was first introduced on the Intel Xeon E5 v4 family of server processors, as an extension to L3 CAT. The L2 CDP feature is first introduced on future Intel Atom family processors, as an extension to L2 CAT.

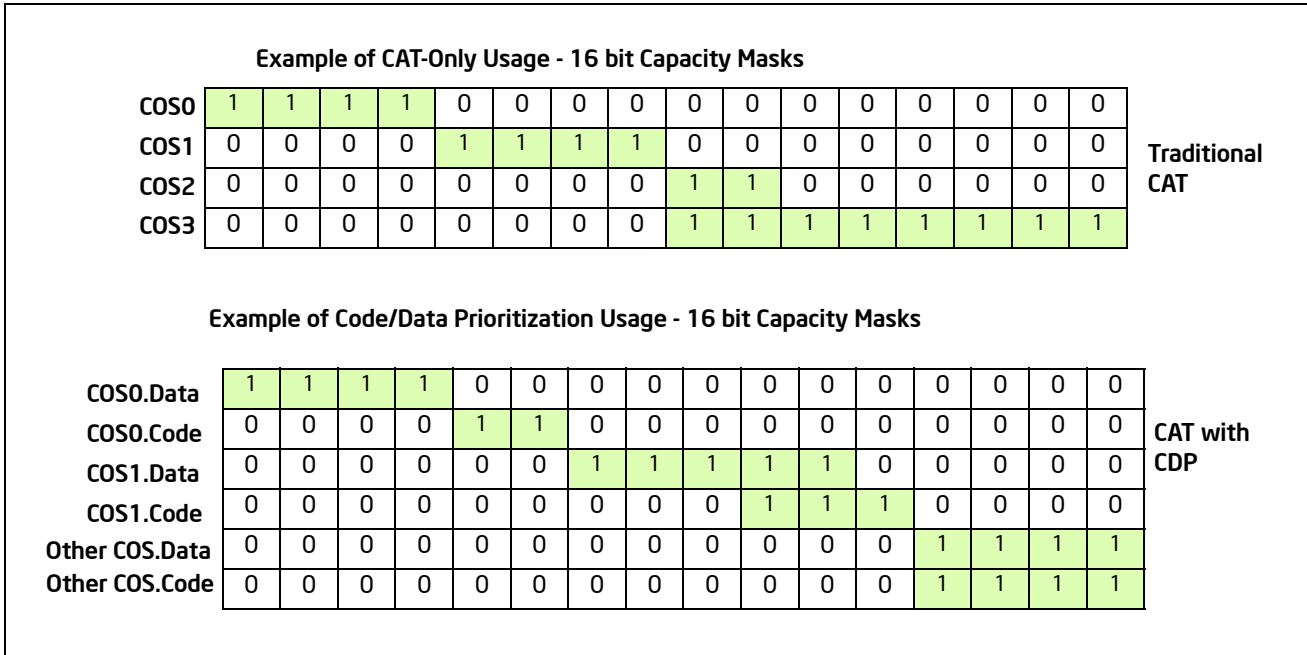
By default, CDP is disabled on the processor. If the CAT MSRs are used without enabling CDP, the processor operates in a traditional CAT-only mode. When CDP is enabled,

- the CAT mask MSRs are re-mapped into interleaved pairs of mask MSRs for data or code fetches (see Figure 17-29),
- the range of COS for CAT is re-indexed, with the lower-half of the COS range available for CDP.

Using the CDP feature, virtual isolation between code and data can be configured on the L2 or L3 cache if desired, similar to how some processor cache levels provide separate L1 data and L1 instruction caches.

Like the CAT feature, CDP may be dynamically configured by privileged software at any point during normal system operation, including dynamically enabling or disabling the feature provided that certain software configuration requirements are met (see Section 17.19.5).

An example of the operating mode of CDP is shown in Figure 17-29. Shown at the top are traditional CAT usage models where capacity masks map 1:1 with a COS number to enable control over the cache space which a given COS (and thus applications, threads or VMs) may occupy. Shown at the bottom are example mask configurations where CDP is enabled, and each COS number maps 1:2 to two masks, one for code and one for data. This enables code and data to be either overlapped or isolated to varying degrees either globally or on a per-COS basis, depending on application and system needs.



**Figure 17-29. Code and Data Capacity Bitmasks of CDP**

When CDP is enabled, the existing mask space for CAT-only operation is split. As an example if the system supports 16 CAT-only COS, when CDP is enabled the same MSR interfaces are used, however half of the masks correspond to code, half correspond to data, and the effective number of COS is reduced by half. Code/Data masks are defined per-COS and interleaved in the MSR space as described in subsequent sections.

In cases where CPUID exposes a non-even number of supported Classes of Service for the CAT or CDP features, software using CDP should use the lower matched pairs of code/data masks, and any upper unpaired masks should not be used. As an example, if CPUID exposes 5 CLOS, when CDP is enabled then two code/data pairs are available (masks 0/1 for CLOS[0] data/code and masks 2/3 for CLOS[1] data/code), however the upper un-paired mask should not be used (mask 4 in this case) or undefined behavior may result.

### 17.19.4 Enabling Cache Allocation Technology Usage Flow

Figure 17-30 illustrates the key steps for OS/VMM to detect support of Cache Allocation Technology and enable priority-based resource allocation for a CAT-capable resource.

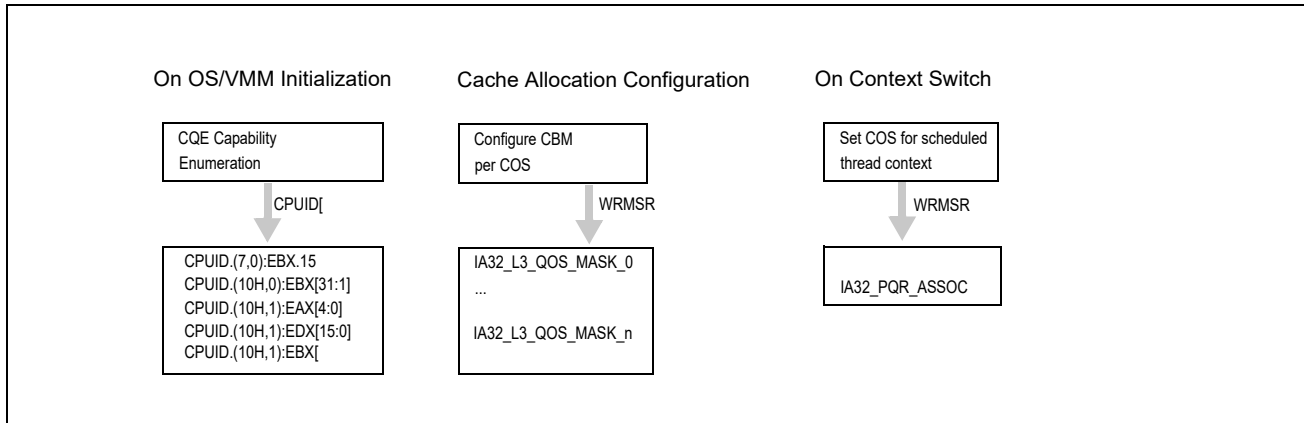


Figure 17-30. Cache Allocation Technology Usage Flow

Enumeration and configuration of L2 CAT is similar to L3 CAT, however CPUID details and MSR addresses differ. Common CLOS are used across the features.

### 17.19.4.1 Enumeration and Detection Support of Cache Allocation Technology

Software can query processor support of CAT capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software must use CPUID leaf 10H to enumerate additional details of available resource types, classes of services and capability bitmasks. The programming interfaces provided by Cache Allocation Technology include:

- CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) and its sub-functions provide information on available resource types, and CAT capability for each resource type (see Section 17.19.4.2).
- IA32\_L3\_MASK\_n: A range of MSRs is provided for each resource type, each MSR within that range specifying a software-configured capacity bitmask for each class of service. For L3 with Cache Allocation support, the CBM is specified using one of the IA32\_L3\_QOS\_MASK\_n MSR, where 'n' corresponds to a number within the supported range of COS, i.e. the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive. See Section 17.19.4.3 for details.
- IA32\_L2\_MASK\_n: A range of MSRs is provided for L2 Cache Allocation Technology, enabling software control over the amount of L2 cache available for each CLOS. Similar to L3 CAT, a CBM is specified for each CLOS using the set of registers, IA32\_L2\_QOS\_MASK\_n MSR, where 'n' ranges from zero to the maximum CLOS number reported for L2 CAT in CPUID. See Section 17.19.4.3 for details.

The L2 mask MSRs are scoped at the same level as the L2 cache (similarly, the L3 mask MSRs are scoped at the same level as the L3 cache). Software may determine which logical processors share an MSR (for instance local to a core, or shared across multiple cores) by performing a write to one of these MSRs and noting which logical threads observe the change. Example flows for a similar method to determine register scope are described in Section 15.5.2, "System Software Recommendation for Managing CMTI and Machine Check Resources". Software may also use CPUID leaf 4 to determine the maximum number of logical processor IDs that may share a given level of the cache.

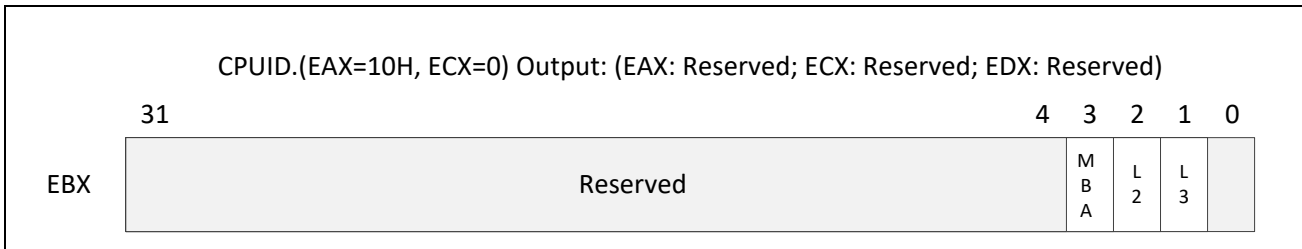
- IA32\_PQR\_ASSOC.CLOS: The IA32\_PQR\_ASSOC MSR provides a COS field that OS/VMM can use to assign a logical processor to an available COS. The set of COS are common across all allocation features, meaning that multiple features may be supported in the same processor without additional software COS management overhead at context swap time. See Section 17.19.4.4 for details.

### 17.19.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration

CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) provides two or more sub-functions:

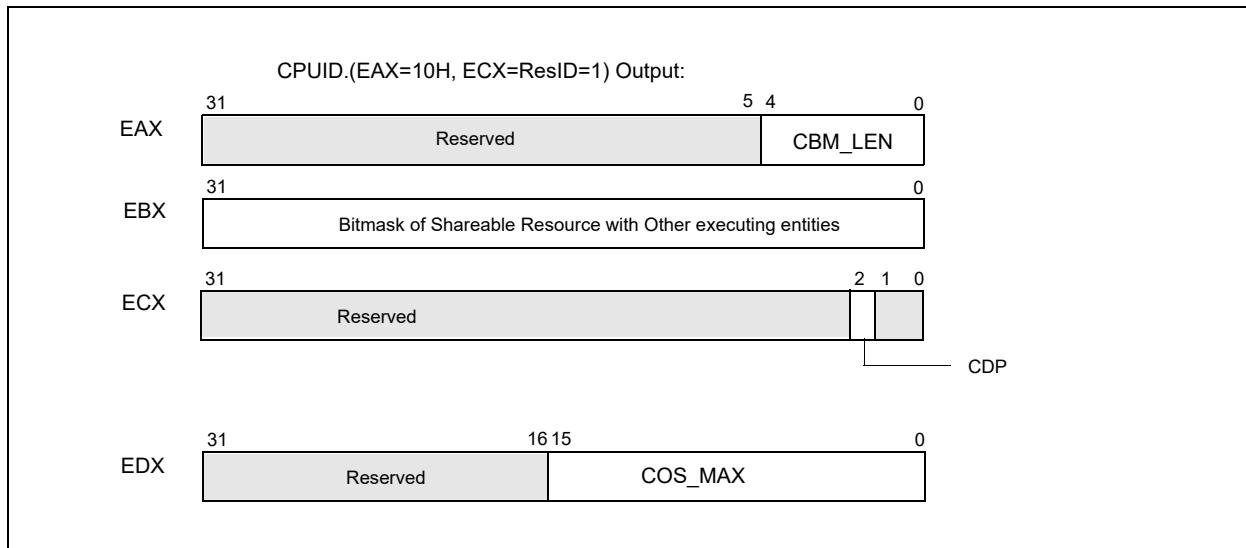
- CAT Enumeration leaf sub-function 0 enumerates available resource types that support allocation control, i.e. by executing CPUID with EAX=10H and ECX=0H. Each supported resource type is represented by a bit field in

CPUID.(EAX=10H, ECX=0):EBX[31:1]. The bit position of each set bit corresponds to a Resource ID (ResID), for instance ResID=1 is used to indicate L3 CAT support, and ResID=2 indicates L2 CAT support. The ResID is also the sub-leaf index that software must use to query details of the CAT capability of that resource type (see Figure 17-31).



**Figure 17-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification**

- For ECX>0, EAX[4:0] reports the length of the capacity bitmask length (ECX=1 or 2 for L2 CAT or L3 CAT respectively) using minus-one notation, e.g., a value of 15 corresponds to the capacity bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.
- Sub-functions of CPUID.EAX=10H with a non-zero ECX input matching a supported ResID enumerate the specific enforcement details of the corresponding ResID. The capabilities enumerated include the length of the capacity bitmasks and the number of Classes of Service for a given ResID. Software should query the capability of each available ResID that supports CAT from a sub-leaf of leaf 10H using the sub-leaf index reported by the corresponding non-zero bit in CPUID.(EAX=10H, ECX=0):EBX[31:1] in order to obtain additional feature details.
- CAT capability for L3 is enumerated by CPUID.(EAX=10H, ECX=1H), see Figure 17-32. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=1) are:

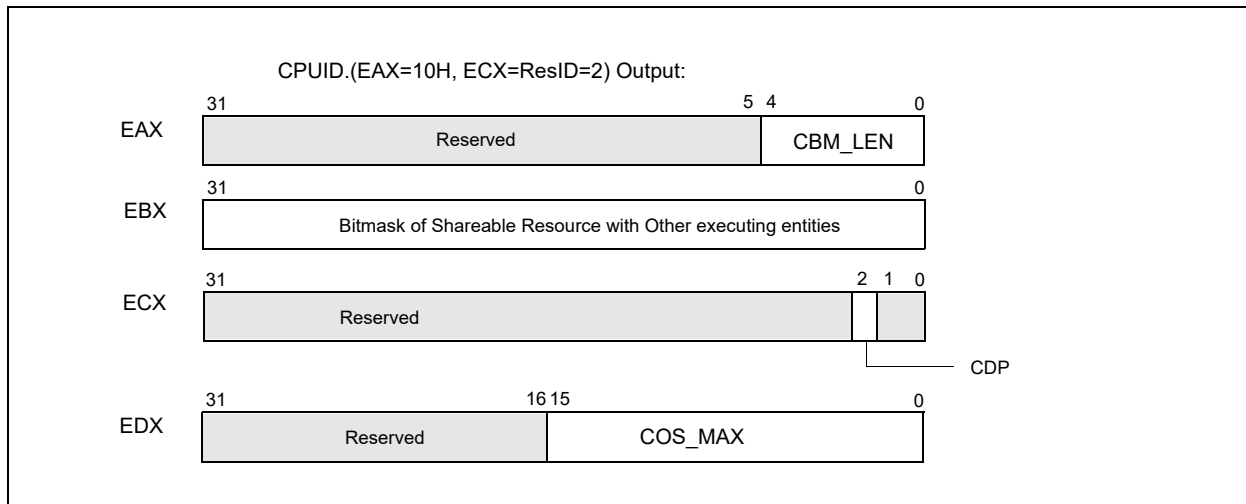


**Figure 17-32. L3 Cache Allocation Technology and CDP Enumeration**

- CPUID.(EAX=10H, ECX=ResID=1):EAX[4:0] reports the length of the capacity bitmask length using minus-one notation, i.e. a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.
- CPUID.(EAX=10H, ECX=1):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L3 allocation may be used by other entities in the platform (e.g. an

integrated graphics engine or hardware units outside the processor core and have direct access to L3). Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.

- CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2]: If 1, indicates L3 Code and Data Prioritization Technology is supported (see Section 17.19.5). Other bits of CPUID.(EAX=10H, ECX=1):ECX are reserved.
- CPUID.(EAX=10H, ECX=1):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.
- CAT capability for L2 is enumerated by CPUID.(EAX=10H, ECX=2H), see Figure 17-33. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=2) are:



**Figure 17-33. L2 Cache Allocation Technology**

- CPUID.(EAX=10H, ECX=ResID=2):EAX[4:0] reports the length of the capacity bitmask length using minus-one notation, i.e. a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.
- CPUID.(EAX=10H, ECX=2):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L2 allocation may be used by other entities in the platform. Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.
- CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2]: If 1, indicates L2 Code and Data Prioritization Technology is supported (see Section 17.19.6). Other bits of CPUID.(EAX=10H, ECX=2):ECX are reserved.
- CPUID.(EAX=10H, ECX=2):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.

A note on migration of Classes of Service (COS): Software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the performance of the Cache Allocation Technology feature may result if COS are migrated frequently. This is aligned with the industry-standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

### 17.19.4.3 Cache Allocation Technology: Cache Mask Configuration

After determining the length of the capacity bitmasks (CBM) and number of COS supported using CPUID (see Section 17.19.4.2), each COS needs to be programmed with a CBM to dictate its available cache via a write to the corresponding IA32\_resourceType\_MASK\_n register, where 'n' corresponds to a number within the supported range of COS, i.e. the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive, and 'resourceType' corresponds to a specific resource as enumerated by the set bits of CPUID.(EAX=10H, ECX=0):EAX[31:1], for instance, 'L2' or 'L3' cache.

A hierarchy of MSRs is reserved for Cache Allocation Technology registers of the form IA32\_resourceType\_MASK\_n:

- From 0C90H through 0D8FH (inclusive), providing support for multiple sub-ranges to support varying resource types. The first supported resourceType is 'L3', corresponding to the L3 cache in a platform. The MSRs range from 0C90H through 0D0FH (inclusive), enables support for up to 128 L3 CAT Classes of Service.

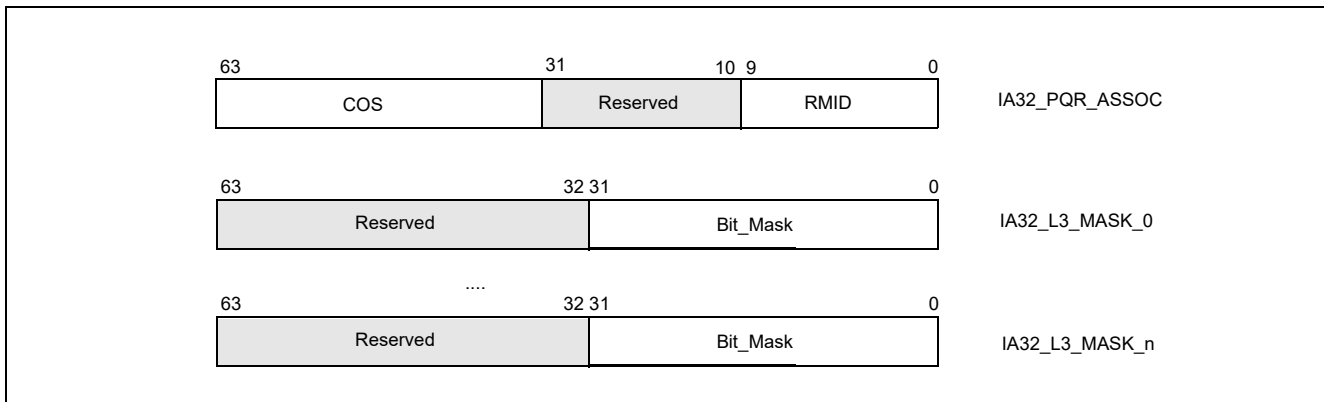


Figure 17-34. IA32\_PQR\_ASSOC, IA32\_L3\_MASK\_n MSRs

- Within the same CAT range hierarchy, another set of registers is defined for resourceType 'L2', corresponding to the L2 cache in a platform, and MSRs IA32\_L2\_MASK\_n are defined for n=[0,63] at addresses 0D10H through 0D4FH (inclusive).

Figure 17-34 and Figure 17-35 provide an overview of the relevant registers.

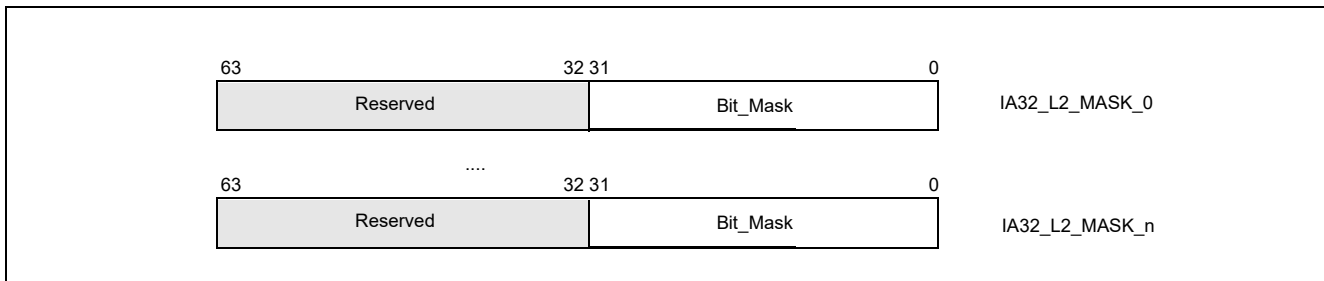


Figure 17-35. IA32\_L2\_MASK\_n MSRs

All CAT configuration registers can be accessed using the standard RDMSR / WRMSR instructions.

Note that once L3 or L2 CAT masks are configured, threads can be grouped into Classes of Service (COS) using the IA32\_PQR\_ASSOC MSR as described in Chapter 17, "Class of Service to Cache Mask Association: Common Across Allocation Features".

### 17.19.4.4 Class of Service to Cache Mask Association: Common Across Allocation Features

After configuring the available classes of service with the preferred set of capacity bitmasks, the OS/VMM can set the IA32\_PQR\_ASSOC.COS of a logical processor to the class of service with the desired CBM when a thread



context switch occurs. This allows the OS/VMM to indicate which class of service an executing thread/VM belongs within. Each logical processor contains an instance of the IA32\_PQR\_ASSOC register at MSR location 0C8FH, and Figure 17-34 shows the bit field layout for this register. Bits[63:32] contain the COS field for each logical processor.

Note that placing the RMID field within the same PQR register enables both RMID and CLOS to be swapped at context swap time for simultaneous use of monitoring and allocation features with a single register write for efficiency.

When CDP is enabled, Specifying a COS value in IA32\_PQR\_ASSOC.COS greater than MAX\_COS\_CDP = (CPUID.(EAX=10H, ECX=1):EDX[15:0] >> 1) will cause undefined performance impact to code and data fetches. In all cases, code and data masks for L2 and L3 CDP should be programmed with at least one bit set.

Note that if the IA32\_PQR\_ASSOC.COS is never written then the CAT capability defaults to using COS 0, which in turn is set to the default mask in IA32\_L3\_MASK\_0 - which is all "1"s (on reset). This essentially disables the enforcement feature by default or for legacy operating systems and software.

See Section 17.19.7, "Introduction to Memory Bandwidth Allocation" for important COS programming considerations including maximum values when using CAT and CDP.

### 17.19.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology

L3 CDP is an extension of L3 CAT. The presence of the L3 CDP feature is enumerated via CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] (see Figure 17-32). Most of the CPUID.(EAX=10H, ECX=1) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=1):EDX.COS\_MAX\_CAT specifies the maximum COS applicable to CAT-only operation. For CDP operations, COS\_MAX\_CDP is equal to (CPUID.(EAX=10H, ECX=1):EDX.COS\_MAX\_CAT >> 1).

If CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] = 1, the processor supports CDP and provides a new MSR IA32\_L3\_QOS\_CFG at address 0C81H. The layout of IA32\_L3\_QOS\_CFG is shown in Figure 17-36. The bit field definition of IA32\_L3\_QOS\_CFG are:

- Bit 0: L3 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32\_PQR\_ASSOC.COS is COS\_MAX\_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).

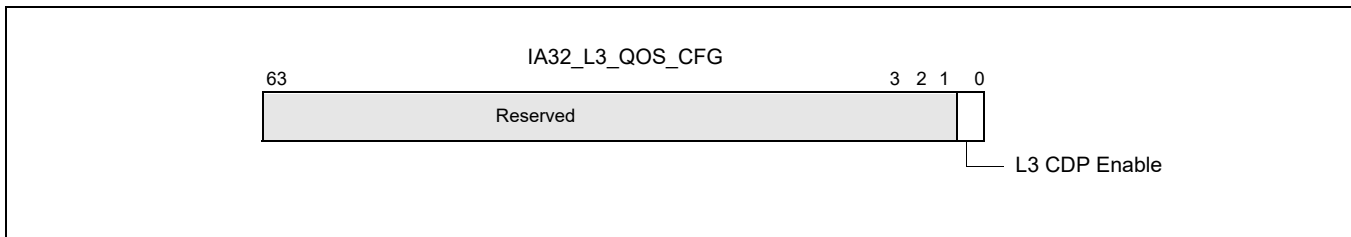


Figure 17-36. Layout of IA32\_L3\_QOS\_CFG

IA32\_L3\_QOS\_CFG default values are all 0s at RESET, the mask MSRs are all 1s. Hence, all logical processors are initialized in COS0 allocated with the entire L3 with CDP disabled, until software programs CAT and CDP. The scope of the IA32\_L3\_QOS\_CFG MSR is defined to be the same scope as the L3 cache (e.g., typically per processor socket). Refer to Section 17.19.7 for software considerations while enabling or disabling L3 CDP.

#### 17.19.5.1 Mapping Between L3 CDP Masks and CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per COS. The re-mapping is shown in Table 17-19.

**Table 17-19. Re-indexing of COS Numbers and Mapping to CAT/CDP Mask MSRs**

Mask MSR	CAT-only Operation	CDP Operation
IA32_L3_QOS_Mask_0	COS0	COS0.Data
IA32_L3_QOS_Mask_1	COS1	COS0.Code
IA32_L3_QOS_Mask_2	COS2	COS1.Data
IA32_L3_QOS_Mask_3	COS3	COS1.Code
IA32_L3_QOS_Mask_4	COS4	COS2.Data
IA32_L3_QOS_Mask_5	COS5	COS2.Code
....	....	....
IA32_L3_QOS_Mask_‘2n’	COS‘2n’	COS‘n’.Data
IA32_L3_QOS_Mask_‘2n+1’	COS‘2n+1’	COS‘n’.Code

One can derive the MSR address for the data mask or code mask for a given COS number ‘n’ by:

- `data_mask_address (n) = base + (n <<1)`, where base is the address of `IA32_L3_QOS_MASK_0`.
- `code_mask_address (n) = base + (n <<1) +1`.

When CDP is enabled, each COS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 17-29).

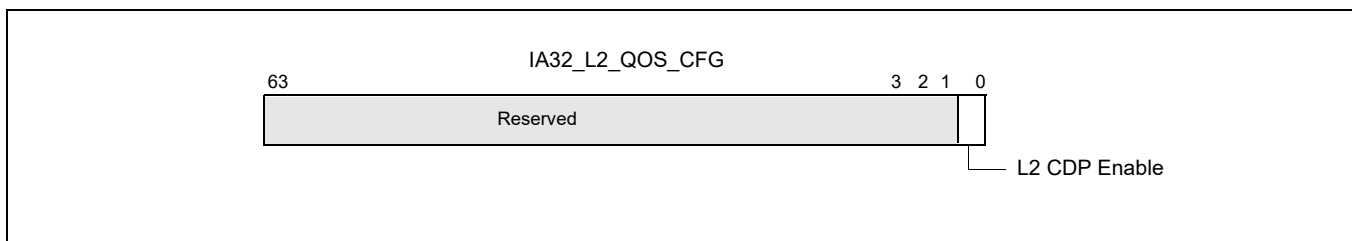
Mask MSR field definitions remain the same. Capacity masks must be formed of contiguous set bits, with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include `0xFFFF`, `0xFF00`, `0x00FF`, `0x00F0`, `0x0001`, `0x0003` and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate `#GP(0)`.

### 17.19.6 Code and Data Prioritization (CDP): Enumerating and Enabling L2 CDP Technology

L2 CDP is an extension of the L2 CAT feature. The presence of the L2 CDP feature is enumerated via CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] (see Figure 17-33). Most of the CPUID.(EAX=10H, ECX=2) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=2):EDX.COS\_MAX\_CAT specifies the maximum COS applicable to CAT-only operation. For CDP operations, `COS_MAX_CDP` is equal to `(CPUID.(EAX=10H, ECX=2):EDX.COS_MAX_CAT >>1)`.

If CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] =1, the processor supports L2 CDP and provides a new MSR `IA32_L2_QOS_CFG` at address `0C82H`. The layout of `IA32_L2_QOS_CFG` is shown in Figure 17-37. The bit field definition of `IA32_L2_QOS_CFG` are:

- Bit 0: L2 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into `IA32_PQR_ASSOC.COS` is `COS_MAX_CDP`.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a `#GP(0)`.



**Figure 17-37. Layout of IA32\_L2\_QOS\_CFG**

IA32\_L2\_QOS\_CFG default values are all 0s at RESET, and the mask MSRs are all 1s. Hence all logical processors are initialized in COS0 allocated with the entire L2 available and with CDP disabled, until software programs CAT and CDP. The IA32\_L2\_QOS\_CFG MSR is defined at the same scope as the L2 cache, typically at the module level for Intel Atom processors for instance. In processors with multiple modules present it is recommended to program the IA32\_L2\_QOS\_CFG MSR consistently across all modules for simplicity.

### 17.19.6.1 Mapping Between L2 CDP Masks and L2 CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per COS. This remapping is the same as the remapping shown in Table 17-19 for L3 CDP, but for the L2 MSR block (IA32\_L2\_QOS\_MASK\_n) instead of the L3 MSR block (IA32\_L3\_QOS\_MASK\_n). The same code / data mask mapping algorithm applies to remapping the MSR block between code and data masks.

As with L3 CDP, when L2 CDP is enabled, each COS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 17-29).

Mask MSR field definitions for L2 CDP remain the same as for L2 CAT. Capacity masks must be formed of contiguous set bits, with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003 and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

### 17.19.6.2 Common L2 and L3 CDP Programming Considerations

Before enabling or disabling L2 or L3 CDP, software should write all 1's to all of the corresponding CAT/CDP masks to ensure proper behavior (e.g., the IA32\_L3\_QOS\_Mask\_n set of MSRs for the L3 CAT feature). When enabling CDP, software should also ensure that only COS number which are valid in CDP operation is used, otherwise undefined behavior may result. For instance in a case with 16 CAT COS, since COS are reduced by half when CDP is enabled, software should ensure that only COS 0-7 are in use before enabling CDP (along with writing 1's to all mask bits before enabling or disabling CDP).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given COS may become a code mask for a different Class of Service when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to COS[0] before enabling or disabling CDP.

### 17.19.6.3 Cache Allocation Technology Dynamic Configuration

All Resource Director Technology (RDT) interfaces including the IA32\_PQR\_ASSOC MSR, CAT/CDP masks, MBA delay values and CQM/MBM registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- Accessing a QOS mask register outside the supported COS (the max COS number is specified in CPUID.(EAX=10H, ECX=ResID):EDX[15:0]), or
- Writing a COS greater than the supported maximum (specified as the maximum value of CPUID.(EAX=10H, ECX=ResID):EDX[15:0] for all valid ResID values) is written to the IA32\_PQR\_ASSOC.CLOS field.

When CDP is enabled, specifying a COS value in IA32\_PQR\_ASSOC.COS outside of the lower half of the COS space will cause undefined performance impact to code and data fetches due to MSR space re-indexing into code/data masks when CDP is enabled.

When reading the IA32\_PQR\_ASSOC register the currently programmed COS on the core will be returned.

When reading an IA32\_resourceType\_MASK\_n register the current capacity bit mask for COS 'n' will be returned.

As noted previously, software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the accuracy of the Cache Allocation feature may result if COS are migrated frequently.

This is aligned with the industry standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

#### 17.19.6.4 Cache Allocation Technology Operation With Power Saving Features

Note that the Cache Allocation Technology feature cannot be used to enforce cache coherency, and that some advanced power management features such as C-states which may shrink or power off various caches within the system may interfere with CAT hints - in such cases the CAT bitmasks are ignored and the other features take precedence. If the highest possible level of CAT differentiation or determinism is required, disable any power-saving features which shrink the caches or power off caches. The details of the power management interfaces are typically implementation-specific, but can be found at *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

If software requires differentiation between threads but not absolute determinism then in many cases it is possible to leave power-saving cache shrink features enabled, which can provide substantial power savings and increase battery life in mobile platforms. In such cases when the caches are powered off (e.g., package C-states) the entire cache of a portion thereof may be powered off. Upon resuming an active state any new incoming data to the cache will be filled subject to the cache capacity bitmasks. Any data in the cache prior to the cache shrink or power off may have been flushed to memory during the process of entering the idle state, however, and is not guaranteed to remain in the cache. If differentiation between threads is the goal of system software then this model allows substantial power savings while continuing to deliver performance differentiation. If system software needs optimal determinism then power saving modes which flush portions of the caches and power them off should be disabled.

#### NOTE

IA32\_PQR\_ASSOC is saved and restored across C6 entry/exit. Similarly, the mask register contents are saved across package C-state entry/exit and are not lost.

#### 17.19.6.5 Cache Allocation Technology Operation with Other Operating Modes

The states in IA32\_PQR\_ASSOC and mask registers are unmodified across an SMI delivery. Thus, the execution of SMM handler code can interact with the Cache Allocation Technology resource and manifest some degree of non-determinism to the non-SMM software stack. An SMM handler may also perform certain system-level or power management practices that affect CAT operation.

It is possible for an SMM handler to minimize the impact on data determinism in the cache by reserving a COS with a dedicated partition in the cache. Such an SMM handler can switch to the dedicated COS immediately upon entering SMM, and switching back to the previously running COS upon exit.

#### 17.19.6.6 Associating Threads with CAT/CDP Classes of Service

Threads are associated with Classes of Service (CLOS) via the per-logical-processor IA32\_PQR\_ASSOC MSR. The same COS concept applies to both CAT and CDP (for instance, COS[5] means the same thing whether CAT or CDP is in use, and the COS has associated resource usage constraint attributes including cache capacity masks). The mapping of COS to mask MSRs does change when CDP is enabled, according to the following guidelines:

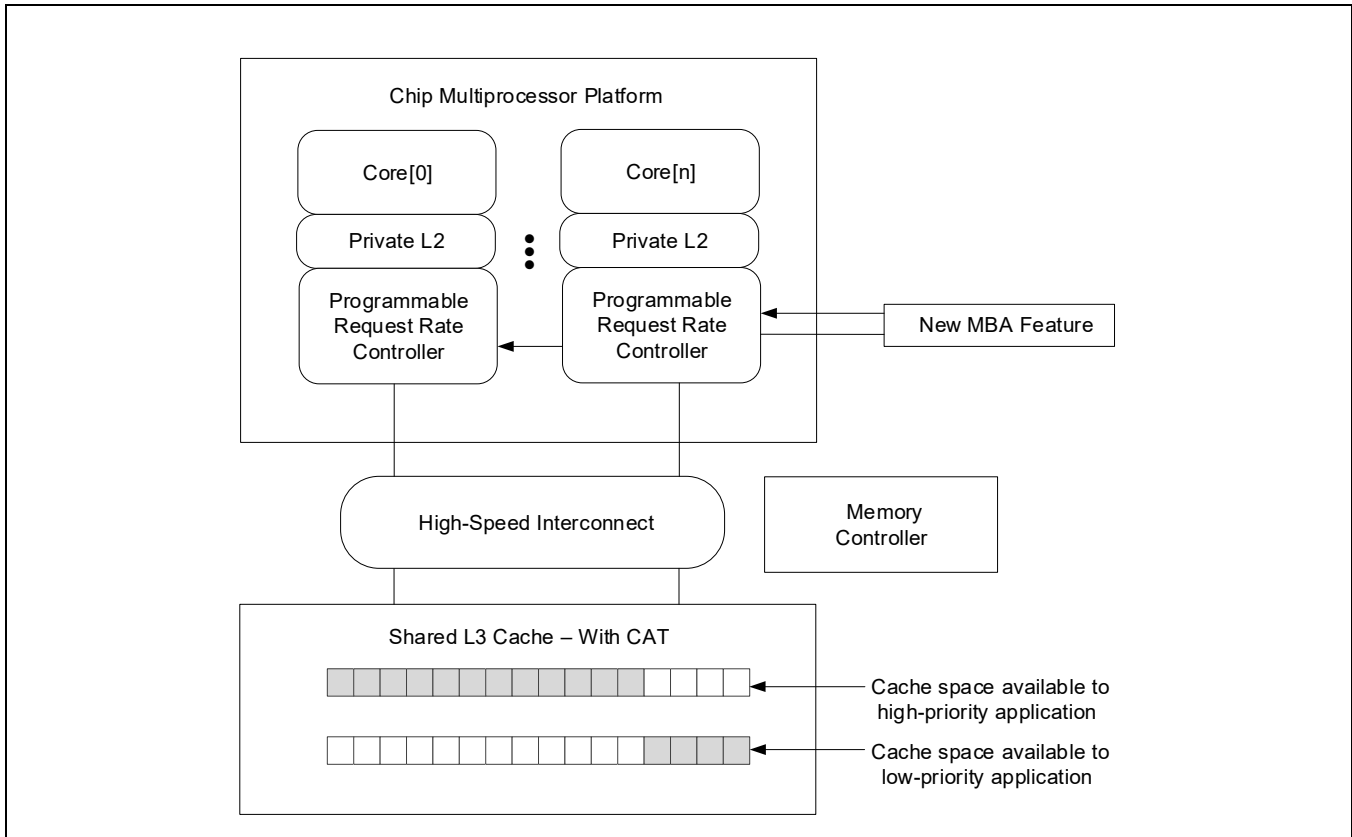
- In CAT-only Mode - one set of bitmasks in one mask MSR control both code and data.
  - Each COS number map 1:1 with a capacity mask on the applicable resource (e.g., L3 cache).
- When CDP is enabled,
  - Two mask sets exist for each COS number, one for code, one for data.
  - Masks for code/data are interleaved in the MSR address space (see Table 17-19).

### 17.19.7 Introduction to Memory Bandwidth Allocation

The Memory Bandwidth Allocation (MBA) feature provides indirect and approximate control over memory bandwidth available per-core, and was introduced on the Intel Xeon Processor Scalable Family. This feature provides a method to control applications which may be over-utilizing bandwidth relative to their priority in environments such as the data-center.

The MBA feature uses existing constructs from the Resource Director Technology (RDT) feature set including Classes of Service (CLOS). A given CLOS used for L3 CAT for instance means the same thing as a CLOS used for MBA. Infrastructure such as the MSR used to associate a thread with a CLOS (the IA32\_PQR\_ASSOC\_MSR) and some elements of the CPUID enumeration (such as CPUID leaf 10H) are shared.

The high-level implementation of Memory Bandwidth Allocation is shown in Figure 17-38.



**Figure 17-38. A High-Level Overview of the MBA Feature**

As shown in Figure 17-38, the MBA feature introduces a programmable request rate controller between the cores and the high-speed interconnect, enabling indirect control over memory bandwidth for cores over-utilizing bandwidth relative to their priority. For instance, high-priority cores may be run un-throttled, but lower priority cores generating an excessive amount of traffic may be throttled to enable more bandwidth availability for the high-priority cores.

Since MBA uses a programmable rate controller between the cores and the interconnect, higher-level shared caches and memory controller, bandwidth to these caches may also be reduced, so care should be taken to throttle only bandwidth-intense applications which do not use the off-core caches effectively.

The throttling values exposed by MBA are approximate, and are calibrated to specific traffic patterns. As work-load characteristics vary, the throttling values provided may affect each workload differently. In cases where precise control is needed, the Memory Bandwidth Monitoring (MBM) feature can be used as input to a software controller which makes decisions about the MBA throttling level to apply.

Enumeration and configuration details are discussed below followed by usage model considerations.

### 17.19.7.1 Memory Bandwidth Allocation Enumeration

Similar to other RDT features, enumeration of the presence and details of the MBA feature is provided via a sub-leaf of the CPUID instruction.

Key components of the enumeration are as follows.

- Support for the MBA feature on the processor, and if MBA is supported, the following details:
  - Number of supported Classes of Service (CLOS) for the processor.
  - The maximum MBA delay value supported (which also implicitly provides a definition of the granularity).
  - An indication of whether the delay values which can be programmed are linearly spaced or not.

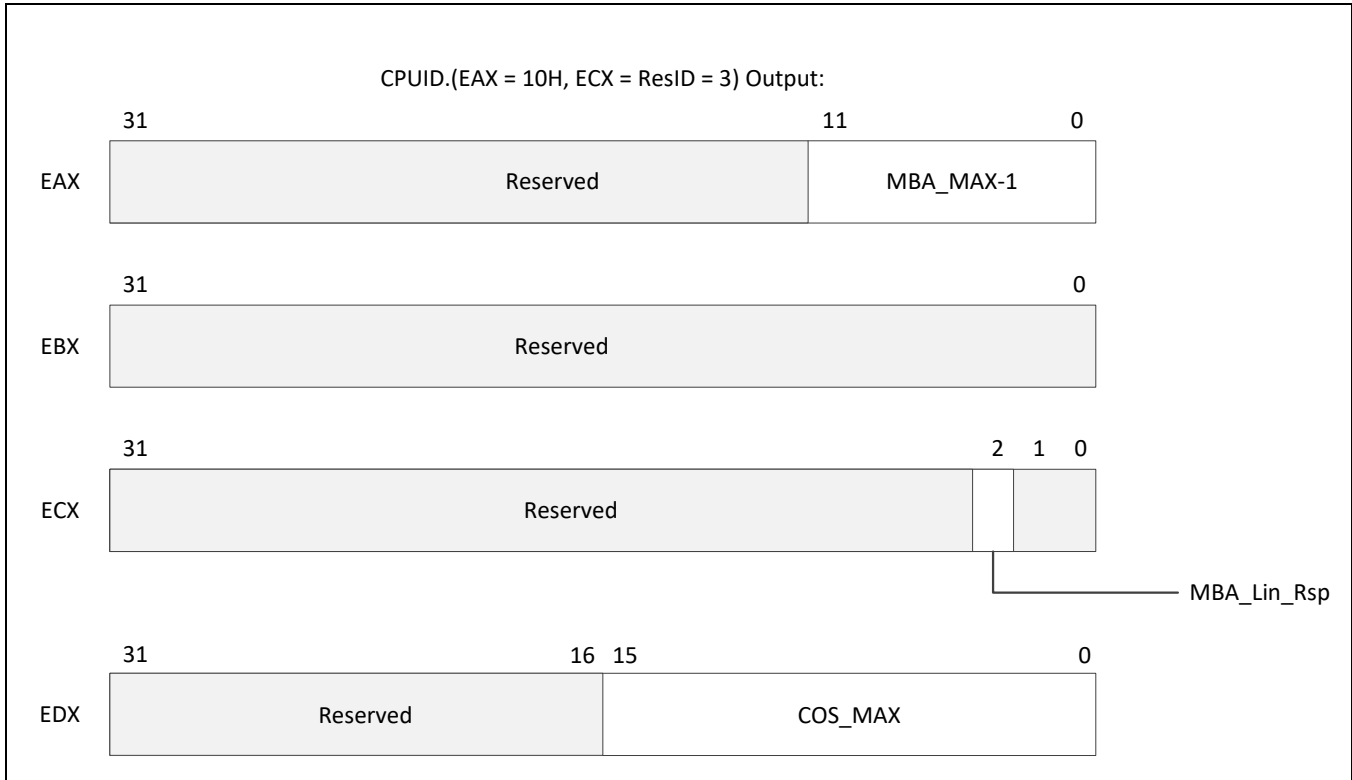
The presence of any of the RDT features which enable control over shared platform resources is enumerated by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software may then use CPUID leaf 10H to enumerate additional details on the specific controls provided.

Through CPUID leaf 10H software may determine whether MBA is supported on the platform. Specifically, as shown in Figure 17-31, bit 3 of the EBX register indicates whether MBA is supported on the processor, and the bit position (3) constitutes a Resource ID (ResID) which allows enumeration of MBA details. For instance, if bit 3 is supported this implies the presence of CPUID.10H.[ResID=3] as shown in Figure 17-38 which provides the following details.

- CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] reports the maximum MBA throttling value supported, minus one. For instance, a value of 89 indicates that a maximum throttling value of 90 is supported. Additionally, in cases where a linear interface (see below) is supported then one hundred minus the maximum throttling value indicates the granularity, 10% in this example.
- CPUID.(EAX=10H, ECX=ResID=3):EBX is reserved.
- CPUID.(EAX=10H, ECX=ResID=3):ECX[2] reports whether the response of the delay values is linear (see text).
- CPUID.(EAX=10H, ECX=ResID=3):EDX[15:0] reports the number of Classes of Service (CLOS) supported for the feature (minus one). For instance, a reported value of 15 implies a maximum of 16 supported MBA CLOS.

The number of CLOS supported for the MBA feature may or may not align with other resources such as L3 CAT. In cases where the RDT features support different numbers of CLOS the lowest numerical CLOS support the common set of features, while higher CLOS may support a subset. For instance, if L3 CAT supports 8 CLOS while MBA supports 4 CLOS, all 8 CLOS would have L3 CAT masks available for cache control, but the upper 4 CLOS would not offer MBA support. In this case the upper 4 CLOS would not be subject to any throttling control. Software can manage supported resources / CLOS in order to either have consistent capabilities across CLOS by using the common subset or enable more flexibility by selectively applying resource control where needed based on careful CLOS and thread mapping. In all cases, CLOS[0] supports all RDT resource control features present on the platform.

Discussion on the interpretation and usage of the MBA delay values is provided in Section 17.19.7.2 on MBA configuration.



**Figure 17-39. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification**

### 17.19.7.2 Memory Bandwidth Allocation Configuration

The configuration of MBA takes consists of two processes once enumeration is complete.

- Association of threads to Classes of Service (CLOS) - accomplished in a common fashion across RDT features as described in Section 17.19.7.1 via the IA32\_PQR\_ASSOC MSR. As with features such as L3 CAT, software may update the CLOS field of the PQR MSR at context swap time in order to maintain the proper association of software threads to Classes of Service on the hardware. While logical processors may each be associated with independent CLOS, see Section 17.19.7.3 for important usage model considerations (initial versions of the MBA feature select the maximum delay value across threads).
- Configuration of the per-CLOS delay values, accomplished via the IA32\_L2\_QoS\_Ext\_BW\_Thrtl\_n MSR set shown in Table 17-20.

The MBA delay values which may be programmed range from zero (implying zero delay, and full bandwidth available) to the maximum (MBA\_MAX) specified in CPUID as discussed in Section 17.19.7.1. The throttling values are approximate and do not sum to 100% across CLOS, rather they should be viewed as a maximum bandwidth “cap” per-CLOS.

Software may select an MBA delay value then write the value into one or more of the IA32\_L2\_QoS\_Ext\_BW\_Thrtl\_n MSRs to update the delay values applied for a specific CLOS. As shown in Table 17-20 the base address of the MSRs is at D50H, and the range corresponds to the maximum supported CLOS from CPUID.(EAX=10H, ECX=ResID=1):EDX[15:0] as described in Section 17.19.7.1. For instance, if 16 CLOS are supported then the valid MSR range will extend from D50H through D5F inclusive.



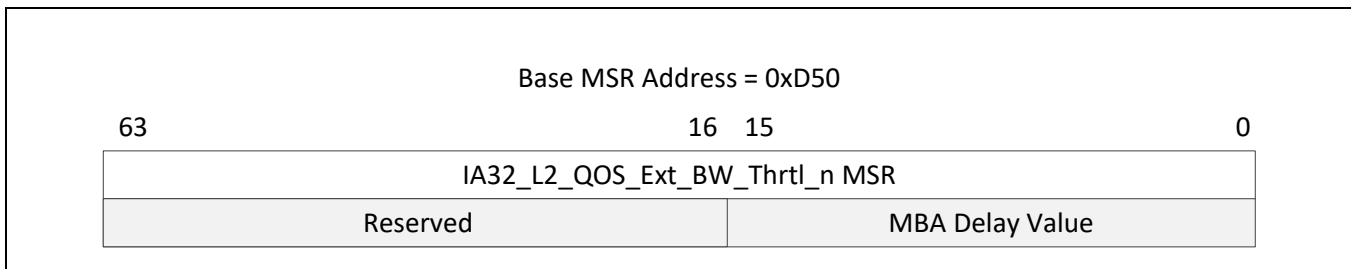
**Table 17-20. MBA Delay Value MSRs**

Delay Value MSR	Address
IA32_L2_QoS_Ext_BW_Thrtl_0	D50H
IA32_L2_QoS_Ext_BW_Thrtl_1	D51H
IA32_L2_QoS_Ext_BW_Thrtl_2	D52H
....	....
IA32_L2_QoS_Ext_BW_Thrtl_'COS_MAX'	D50H + COS_MAX from CPUID.10H.3

The definition for the MBA delay value MSRs is provided in Figure 17.39. The lower 16 bits are used for MBA delay values, and values from zero to the maximum from the CPUID.MBA\_MAX-1 value are supported. Values outside this range will generate #GP(0).

If linear input throttling values are indicated by CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] then values from zero through the MBA\_MAX field from CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] are supported as inputs. In the linear mode the input precision is defined as 100-(MBA\_MAX). For instance, if the MBA\_MAX value is 90, the input precision is 10%. Values not an even multiple of the precision (e.g., 12%) will be rounded down (e.g., to 10% delay applied).

- If linear values are not supported (CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] = 0) then input delay values are powers-of-two from zero to the MBA\_MAX value from CPUID. In this case any values not a power of two will be rounded down the next nearest power of two.



**Figure 17-40. IA32\_L2\_QoS\_Ext\_BW\_Thrtl\_n MSR Definition**

Note that the throttling values provided to software are calibrated through specific traffic patterns, however as workload characteristics may vary the response precision and linearity of the delay values will vary across products, and should be treated as approximate values only.

### 17.19.7.3 Memory Bandwidth Allocation Usage Considerations

As the memory bandwidth control that MBA provides is indirect and approximate, using the feature with a closed-loop controller to also monitor memory bandwidth and how effectively the applications use the cache (via the Cache Monitoring Technology feature) may provide additional value. This approach also allows administrators to provide a band-width target or set-point which a controller could use to guide MBA throttling values applied, and this allows bandwidth control independent of the execution characteristics of the application.

As control is provided per processor core (the max of the delay values of the per-thread CLOS applied to the core) care should be taking in scheduling threads so as to not inadvertently place a high-priority thread (with zero intended MBA throttling) next to a low-priority thread (with MBA throttling intended), which would lead to inadvertent throttling of the high-priority thread.





## 21. Updates to Chapter 18, Volume 3B

Change bars and green text show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes to this chapter: Updates to sections describing performance monitoring for processors based on Ice Lake microarchitecture. Added a note providing the new location for performance monitoring events ( <https://perfmon-events.intel.com/>) and updated previous references to chapter 19 tables to point to online event list.

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

### NOTE

In previous versions of this document, Chapter 19 contained “Performance Monitoring Events”. Chapter 19 has been removed from this document, and the performance monitoring events can now be found here: <https://perfmon-events.intel.com/>.

Additionally, performance monitoring event files for Intel processors are hosted by the Intel Open Source Technology Center. These files can be downloaded here: <https://download.01.org/perfmon/>.

## 18.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, and Intel processors based on Intel NetBurst microarchitecture are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Newer Intel processor generations support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 18.6.3, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture).” Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they can be found at: <https://perfmon-events.intel.com/>.

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and Interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 18.2.

See also:

- Section 18.2, “Architectural Performance Monitoring”
- Section 18.3, “Performance Monitoring (Intel® Core™ Processors and Intel® Xeon® Processors)”
  - Section 18.3.1, “Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem”
  - Section 18.3.2, “Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere”

- Section 18.3.3, “Intel® Xeon® Processor E7 Family Performance Monitoring Facility”
- Section 18.3.4, “Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge”
- Section 18.3.5, “3rd Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.3.6, “4th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.3.7, “5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility”
- Section 18.3.8, “6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.3.9, “10th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.4, “Performance monitoring (Intel® Xeon™ Phi Processors)”
  - Section 18.4.1, “Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring”
- Section 18.5, “Performance Monitoring (Intel Atom® Processors)”
  - Section 18.5.1, “Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)”
  - Section 18.5.2, “Performance Monitoring for Silvermont Microarchitecture”
  - Section 18.5.3, “Performance Monitoring for Goldmont Microarchitecture”
  - Section 18.5.4, “Performance Monitoring for Goldmont Plus Microarchitecture”
  - Section 18.5.5, “Performance Monitoring for Tremont Microarchitecture”
- Section 18.6, “Performance Monitoring (Legacy Intel Processors)”
  - Section 18.6.1, “Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)”
  - Section 18.6.2, “Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)”
  - Section 18.6.3, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)”
  - Section 18.6.4, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
    - Section 18.6.4.5, “Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
  - Section 18.6.5, “Performance Monitoring and Dual-Core Technology”
  - Section 18.6.6, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache”
  - Section 18.6.7, “Performance Monitoring on L3 and Caching Bus Controller Sub-Systems”
  - Section 18.6.8, “Performance Monitoring (P6 Family Processor)”
  - Section 18.6.9, “Performance Monitoring (Pentium Processors)”
- Section 18.7, “Counting Clocks”
- Section 18.8, “IA32\_PERF\_CAPABILITIES MSR Enumeration”
- Section 18.9, “PEBS Facility”

## 18.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support

base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T 7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 1, 2, and 3. Intel processors based on the Skylake, Kaby Lake and Coffee Lake microarchitectures support versionID 4.

Next generation Intel Atom processors are based on the Goldmont microarchitecture. Intel processors based on the Goldmont microarchitecture support versionID 4.

## 18.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32\_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32\_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- Bit field layout of IA32\_PERFEVTSELx is consistent across microarchitectures.
- Addresses of IA32\_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32\_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32\_PERFEVTSELx and IA32\_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available to software in a logical processor (each IA32\_PERFEVTSELx MSR is paired to the corresponding IA32\_PMCx MSR).
- Number of bits supported in each IA32\_PMCx.
- Number of architectural performance monitoring events supported in a logical processor.

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32\_PERFEVTSELx/ IA32\_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

### 18.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32\_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8]. Note that this may vary from the number

of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

- IA32\_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block. Note the number of IA32\_PERFEVTSELx MSRs may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.
- The bit width of an IA32\_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32\_PERFEVTSELx MSRs is defined architecturally.

See Figure 18-1 for the bit field layout of IA32\_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7)** — Selects the event logic unit used to detect microarchitectural conditions (see Table 18-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

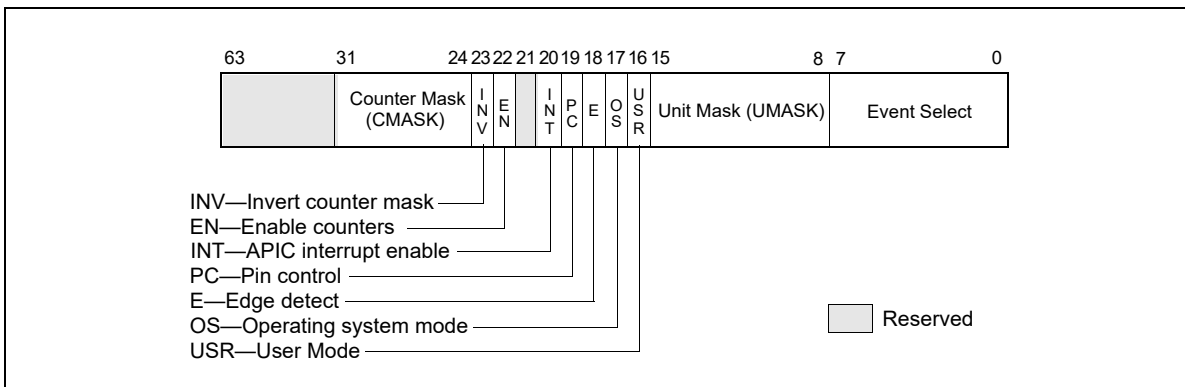


Figure 18-1. Layout of IA32\_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 18-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX.

- **USR (user mode) flag (bit 16)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- **OS (operating system mode) flag (bit 17)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19)** — Beginning with Sandy Bridge microarchitecture, this bit is reserved (not writeable). On processors based on previous microarchitectures, the logical processor toggles the PMi pins and

increments the counter when performance-monitoring events occur; when clear, the processor toggles the PMi pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

- **INT (APIC interrupt enable) flag (bit 20)** — When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22)** — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32\_PERFEVTSELx[bit 22] = 0, before writing to IA32\_PMCx.
- **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31)** — When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

### 18.2.1.2 Pre-defined Architectural Performance Events

Table 18-1 lists architecturally defined events.

**Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events**

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles <sup>1</sup>	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H
7	Topdown Slots	01H	A4H

#### NOTES:

1. Current implementations count at core crystal clock, TSC, or bus clock frequency.

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 18-1). The number of architectural events is reported through CPUID.0AH:EAX[31:24], while non-zero bits in CPUID.0AH:EBX indicate any architectural events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles** — Event select 3CH, Umask 00H  
This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:
  - an ACPI C-state other than C0 for normal operation
  - HLT
  - STPCLK# pin asserted

- being throttled by TM1
- during the frequency switching phase of a performance state transition (see Chapter 14, “Power and Thermal Management”)

The performance counter for this event counts across performance state transitions using different core clock frequencies

- **Instructions Retired** — Event select C0H, Umask 00H

This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

- **UnHalted Reference Cycles** — Event select 3CH, Umask 01H

This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

- **Last Level Cache References** — Event select 2EH, Umask 4FH

This event counts requests originating from the core that reference a cache line in the last level on-die cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Last Level Cache Misses** — Event select 2EH, Umask 41H

This event counts each cache miss condition for references to the last level on-die cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Branch Instructions Retired** — Event select C4H, Umask 00H

This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

- **All Branch Mispredict Retired** — Event select C5H, Umask 00H

This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

- **Topdown Slots** — Event select A4H, Umask 01H

This event counts the total number of available slots for an unhalted logical processor.

The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core, in processors that support Intel Hyper-Threading Technology.

Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

## NOTE

Programming decisions or software precisions on functionality should not be based on the event values or dependent on the existence of performance monitoring events.



## 18.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32\_FIXED\_CTR0 through IA32\_FIXED\_CTR2). Each of the fixed-function PMC can count only one architectural performance event.  
Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32\_FIXED\_CTR\_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32\_PMCx) via UMASK field in (IA32\_PERFEVTSELx), configuring, programming IA32\_FIXED\_CTR\_CTRL for fixed-function PMCs do not require any UMASK.
- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - IA32\_PERF\_GLOBAL\_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32\_FIXED\_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32\_PERF\_GLOBAL\_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32\_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32\_DEBUGCTL.Freeze\_LBR\_On\_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32\_DEBUGCTL.Freeze\_PerfMon\_On\_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

### NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32\_FIXED\_CTR\_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 18-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

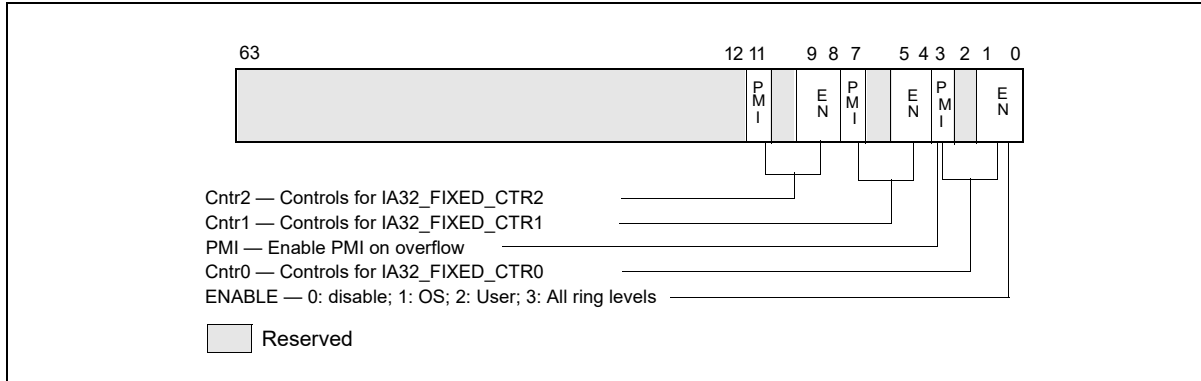


Figure 18-2. Layout of IA32\_FIXED\_CTR\_CTRL MSR

- Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32\_PERF\_GLOBAL\_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 18-3 shows the layout of IA32\_PERF\_GLOBAL\_CTRL. Each enable bit in IA32\_PERF\_GLOBAL\_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or IA32\_PERF\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

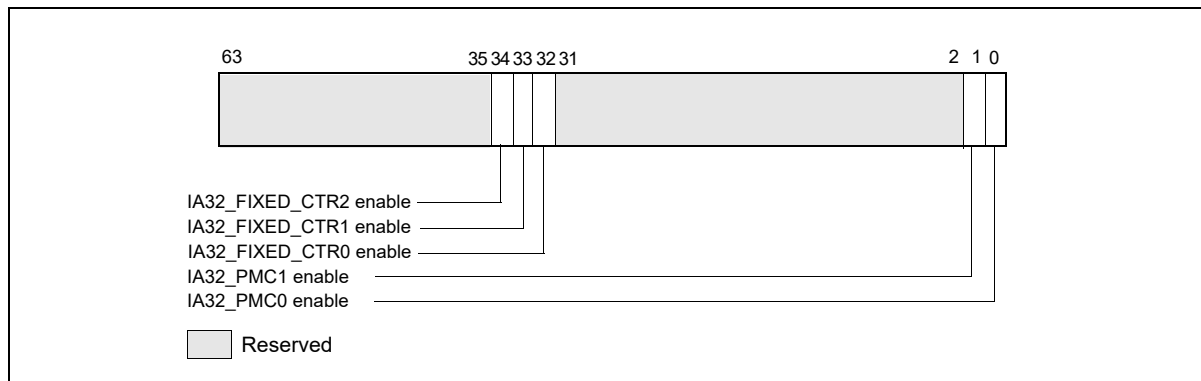


Figure 18-3. Layout of IA32\_PERF\_GLOBAL\_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

**Table 18-2. Association of Fixed-Function Performance Counters with Architectural Performance Events**

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_FIXED_CTR0	309H	INST_RETIRED.ANY	This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.
IA32_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD CPU_CLK_UNHALTED.CORE	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state.  If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core.  The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.
IA32_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.
IA32_FIXED_CTR3	30CH	TOPDOWN.SLOTS	This event counts the number of available slots for an unhalted logical processor. The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core.  Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

IA32\_PERF\_GLOBAL\_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32\_PERF\_GLOBAL\_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32\_PERF\_GLOBAL\_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 18-4 shows the layout of IA32\_PERF\_GLOBAL\_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status, and sets the "OvfBuffer" bit in IA32\_PERF\_GLOBAL\_STATUS.

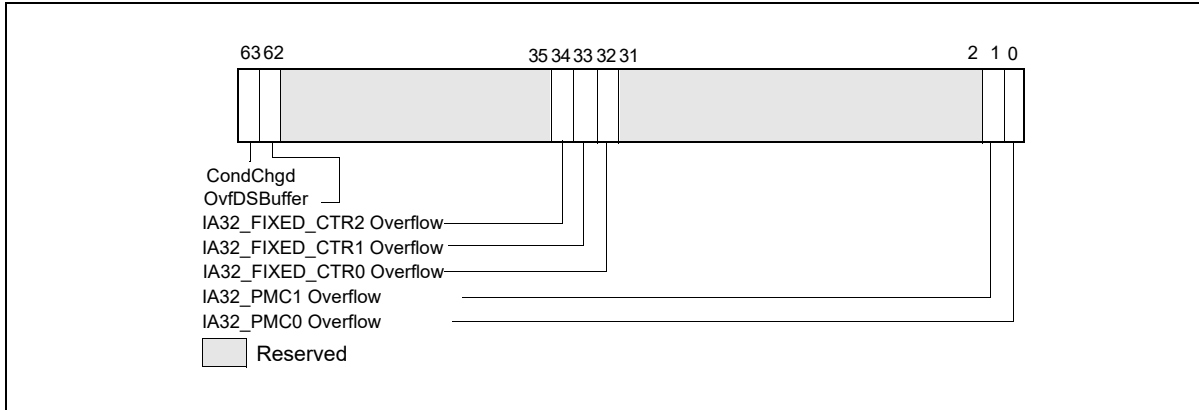


Figure 18-4. Layout of IA32\_PERF\_GLOBAL\_STATUS MSR

IA32\_PERF\_GLOBAL\_OVF\_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32\_PERF\_GLOBAL\_OVF\_CTL is shown in Figure 18-5.

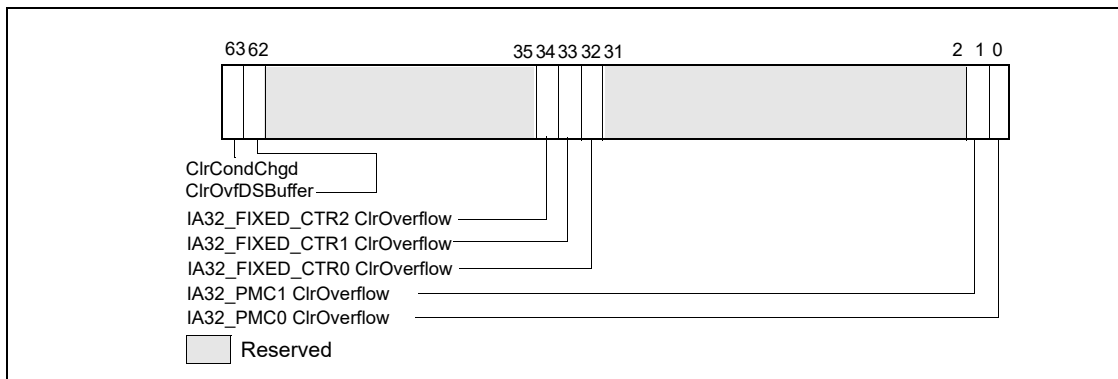


Figure 18-5. Layout of IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR

### 18.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e. a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- AnyThread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  - Each IA32\_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 18-6.

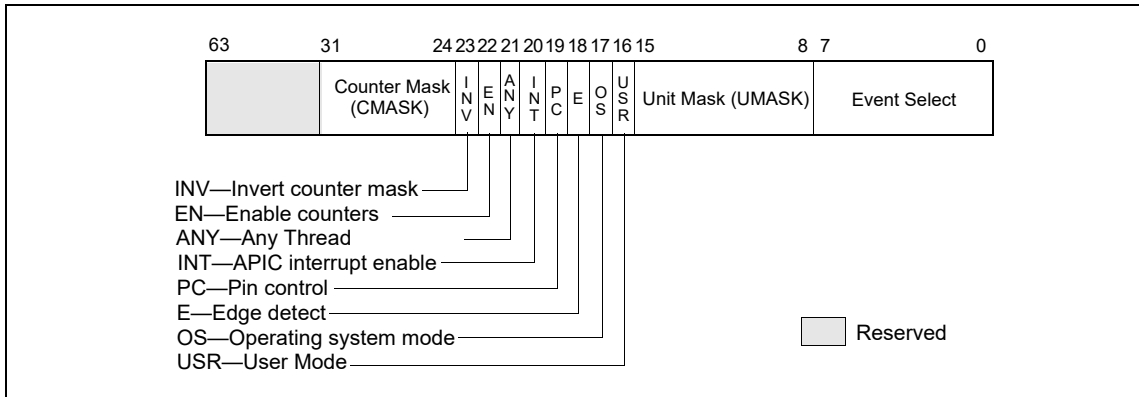


Figure 18-6. Layout of IA32\_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

**Bit 21 (AnyThread)** of IA32\_PERFEVTSELx is supported in architectural performance monitoring version 3 for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions (including matching the thread’s CPL with the OS/USR setting of IA32\_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread’s CPL with the OS/USR setting of IA32\_PERFEVTSELx) occurring in the logical processor which programmed the IA32\_PERFEVTSELx MSR.

- Each fixed-function performance counter IA32\_FIXED\_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32\_PERF\_FIXED\_CTR\_CTRL MSR. The control block also allow thread-specificity configuration using an AnyThread bit. The layout of IA32\_PERF\_FIXED\_CTR\_CTRL MSR is shown.

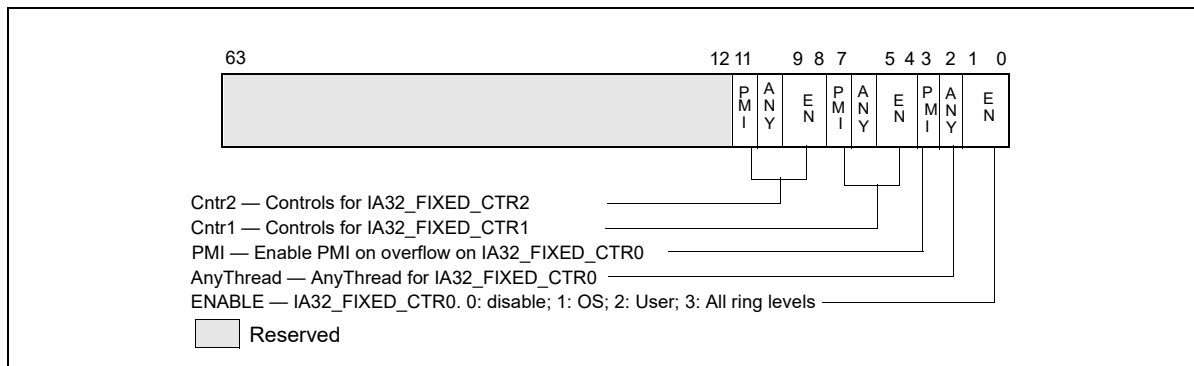


Figure 18-7. IA32\_PERF\_FIXED\_CTR\_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides an **AnyThread** (bit position 2 + 4\*N, N= 0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread’s CPL with the ENABLE setting of the corresponding control block of IA32\_PERF\_FIXED\_CTR\_CTRL) occurring across all logical processors sharing a processor core. When an **AnyThread** bit is 0 in IA32\_PERF\_FIXED\_CTR\_CTRL, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the IA32\_PERF\_FIXED\_CTR\_CTRL MSR.

- The IA32\_PERF\_GLOBAL\_CTRL, IA32\_PERF\_GLOBAL\_STATUS, IA32\_PERF\_GLOBAL\_OVF\_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 18-8 and Figure 18-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.



### 18.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL. While AnyThread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 22, “Introduction to Virtual-Machine Extensions”) where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow an individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- Configure the MSR bitmap to cause VM-exits for WRMSR to IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL in VMX non-Root operation (see CHAPTER 23 for additional information),
- Clear the AnyThread bit of IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL in the MSR-load lists for VM exits and VM entries (see CHAPTER 23, CHAPTER 25, and CHAPTER 26).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted.

## 18.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32\_DEBUGCTL.Freeze\_LBRs\_On\_PMI and Freeze\_PerfMon\_On\_PMI. Specifically version 4 provides the following enhancements:

- New indicators (LBR\_FRZ, CTR\_FRZ) in IA32\_PERF\_GLOBAL\_STATUS, see Section 18.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32\_DEBUGCTL.Freeze\_LBRs\_On\_PMI and IA32\_DEBUGCTL.Freeze\_PerfMon\_On\_PMI: see Section 18.2.4.1. Legacy semantics of Freeze\_LBRs\_On\_PMI and Freeze\_PerfMon\_On\_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32\_PERF\_GLOBAL\_STATUS and read-only performance counter enabling interface in IA32\_PERF\_GLOBAL\_STATUS: see Section 18.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

### 18.2.4.1 Enhancement in IA32\_PERF\_GLOBAL\_STATUS

The IA32\_PERF\_GLOBAL\_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ[bit 58]: This bit is set due to the following conditions:
  - IA32\_DEBUGCTL.FREEZE\_LBR\_ON\_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

Effectively, the IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ bit also serves as a control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:

$$\text{— } ( \text{IA32\_DEBUGCTL.LBR} \ \& \ ( \text{!IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ} ) ) = 1$$

- IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ[bit 59]: This bit is set due to the following conditions:
  - IA32\_DEBUGCTL.FREEZE\_PERFMON\_ON\_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

Effectively, the IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:

- $(IA32\_PERFEVTSELn.EN \ \& \ IA32\_PERF\_GLOBAL\_CTRL.PMCn \ \& \ (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ)) = 1$  for programmable counter 'n', or
- $(IA32\_PERF\_FIXED\_CTRL.ENi \ \& \ IA32\_PERF\_GLOBAL\_CTRL.FCi \ \& \ (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ)) = 1$  for fixed counter 'i'

The read-only enable interface IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ provides a more efficient flow for a PMI handler to use IA32\_DEBUGCTL.Freeze\_Perfmon\_On\_PMI to filter out data that may distort target workload analysis, see Table 17-3. It should be noted the IA32\_PERF\_GLOBAL\_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32\_PERF\_GLOBAL\_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g. race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32\_PERF\_GLOBAL\_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32\_PERF\_GLOBAL\_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support SGX (For additional information about Intel SGX, see "Intel® Software Guard Extensions Programming Reference".):

- IA32\_PERF\_GLOBAL\_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e. IA32\_PMCx or IA32\_FIXED\_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32\_PERF\_GLOBAL\_STATUS.ASCI was cleared).

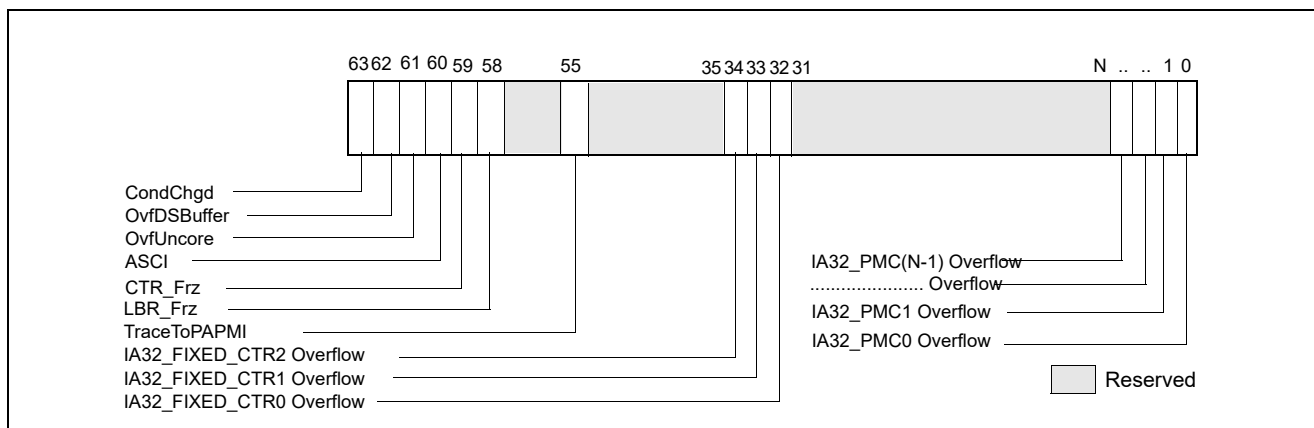


Figure 18-10. IA32\_PERF\_GLOBAL\_STATUS MSR and Architectural Perfmon Version 4

Note, a processor’s support for IA32\_PERF\_GLOBAL\_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32\_PERF\_GLOBAL\_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.



### 18.2.4.2 IA32\_PERF\_GLOBAL\_STATUS\_RESET and IA32\_PERF\_GLOBAL\_STATUS\_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32\_PERF\_GLOBAL\_STATUS MSR by software is done via IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32\_PERF\_GLOBAL\_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32\_PERF\_GLOBAL\_OVF\_CTRL is inherited by IA32\_PERF\_GLOBAL\_STATUS\_RESET (see Figure 18-11). Further, IA32\_PERF\_GLOBAL\_STATUS\_RESET provides additional bit fields to clear the new indicators in IA32\_PERF\_GLOBAL\_STATUS described in Section 18.2.4.1.

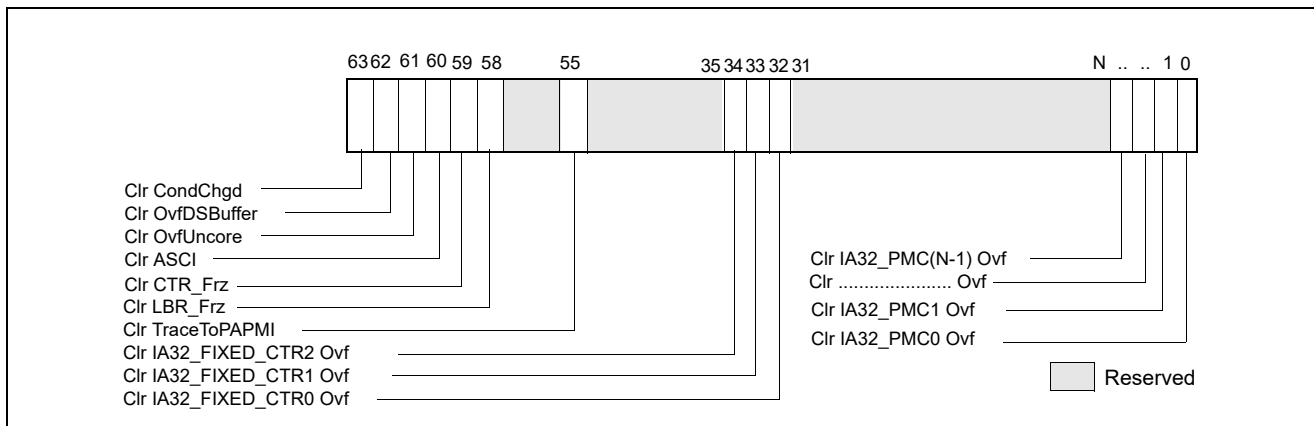


Figure 18-11. IA32\_PERF\_GLOBAL\_STATUS\_RESET MSR and Architectural Perfmon Version 4

The IA32\_PERF\_GLOBAL\_STATUS\_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32\_PERF\_GLOBAL\_STATUS. The IA32\_PERF\_GLOBAL\_STATUS\_SET interface can be used by a VMM to virtualize the state of IA32\_PERF\_GLOBAL\_STATUS across VMs.

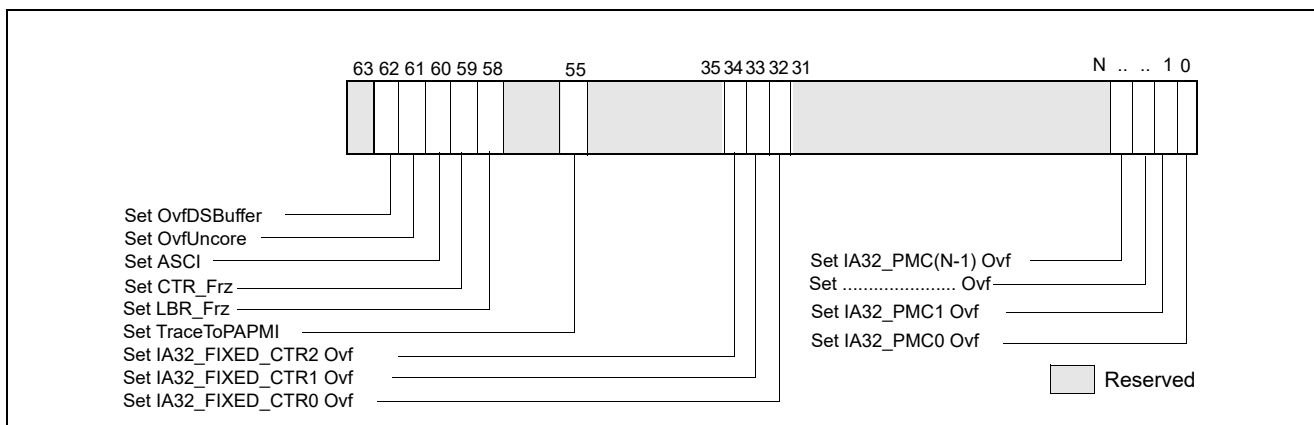


Figure 18-12. IA32\_PERF\_GLOBAL\_STATUS\_SET MSR and Architectural Perfmon Version 4

### 18.2.4.3 IA32\_PERF\_GLOBAL\_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor’s performance monitoring facilities. The IA32\_MISC\_ENABLE.PERFMON\_AVAILABLE[bit 7] interface could not serve

the need of multiple agent adequately. A white paper, “Performance Monitoring Unit Sharing Guideline”<sup>1</sup>, proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32\_PERF\_GLOBAL\_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32\_PERF\_GLOBAL\_INUSE is shown in Figure 18-13.

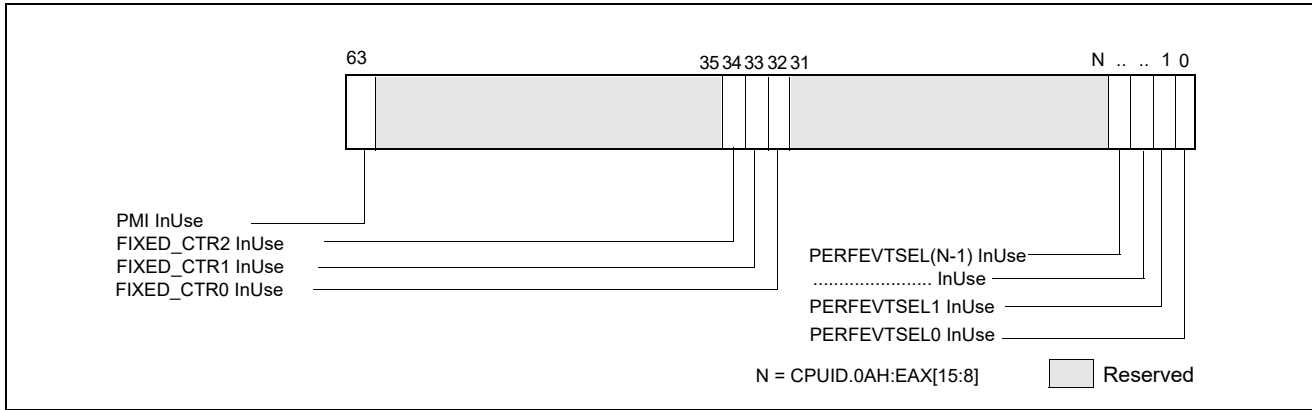


Figure 18-13. IA32\_PERF\_GLOBAL\_INUSE MSR and Architectural Perfmon Version 4

The IA32\_PERF\_GLOBAL\_INUSE MSR provides an “InUse” bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSEL0\_InUse[bit 0]: This bit reflects the logical state of (IA32\_PERFEVTSEL0[7:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSEL1\_InUse[bit 1]: This bit reflects the logical state of (IA32\_PERFEVTSEL1[7:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSEL2\_InUse[bit 2]: This bit reflects the logical state of (IA32\_PERFEVTSEL2[7:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSELn\_InUse[bit n]: This bit reflects the logical state of (IA32\_PERFEVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- IA32\_PERF\_GLOBAL\_INUSE.FC0\_InUse[bit 32]: This bit reflects the logical state of (IA32\_FIXED\_CTR\_CTRL[1:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.FC1\_InUse[bit 33]: This bit reflects the logical state of (IA32\_FIXED\_CTR\_CTRL[5:4] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.FC2\_InUse[bit 34]: This bit reflects the logical state of (IA32\_FIXED\_CTR\_CTRL[9:8] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PMI\_InUse[bit 63]: This bit is set if any one of the following bit is set:
  - IA32\_PERFEVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8].
  - IA32\_FIXED\_CTR\_CTRL.ENi\_PMI, i = 0, 1, 2.
  - Any IA32\_PEBS\_ENABLES bit which enables PEBS for a general-purpose or fixed-function performance counter.

1. Available at <http://www.intel.com/sdm>

## 18.2.5 Architectural Performance Monitoring Version 5

Processors supporting architectural performance monitoring version 5 also support versions 1, 2, 3 and 4, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 5 provides the following enhancements:

- Deprecation of Anythread mode, see Section 18.2.5.1.
- Individual enumeration of Fixed counters in CPUID.0AH, see Section 18.2.5.2.
- Domain separation, see Section 18.2.5.3.

### 18.2.5.1 AnyThread Mode Deprecation

With Architectural Performance Monitoring Version 5, a processor that supports AnyThread mode deprecation is enumerated by CPUID.0AH.EDX[15]. If set, software will not have to follow guidelines in Section 18.2.3.1.

### 18.2.5.2 Fixed Counter Enumeration

With Architectural Performance Monitoring Version 5, register CPUID.0AH.ECX indicates Fixed Counter enumeration. It is a bit mask which enumerates the supported Fixed Counters in a processor. If bit 'i' is set, it implies that Fixed Counter 'i' is supported. Software is recommended to use the following logic to check if a Fixed Counter is supported on a given processor:

```
FxCtr[i]_is_supported := ECX[i] || (EDX[4:0] > i);
```

### 18.2.5.3 Domain Separation

A counter stops counting when the logical processor exits the C0 ACPI C-state.

## 18.2.6 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32\_PMCx are writable via WRMSR instruction. However, the value written into IA32\_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32\_PERF\_CAPABILITIES[13] to enumerate its full-width-write capability. See Figure 18-63.

If IA32\_PERF\_CAPABILITIES.FW\_WRITE[bit 13] = 1, each IA32\_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32\_A\_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32\_A\_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32\_A\_PMCi will cause IA32\_PMCi to be updated by:

```
COUNTERWIDTH = CPUID.0AH:EAX[23:16] bit width of the performance monitoring counter
IA32_PMCi[COUNTERWIDTH-1:32] := EDX[COUNTERWIDTH-33:0];
IA32_PMCi[31:0] := EAX[31:0];
EDX[63:COUNTERWIDTH] are reserved
```

## 18.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS)

### 18.3.1 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem

Intel Core i7 processor family<sup>2</sup> supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® microarchitecture code name Nehalem, and provides four general-purpose performance counters (IA32\_PMC0, IA32\_PMC1, IA32\_PMC2, IA32\_PMC3) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32\_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: <https://perfmon-events.intel.com/>. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as “uncore”.
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32\_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32\_PERFEVTSELx MSR.

The bit fields within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

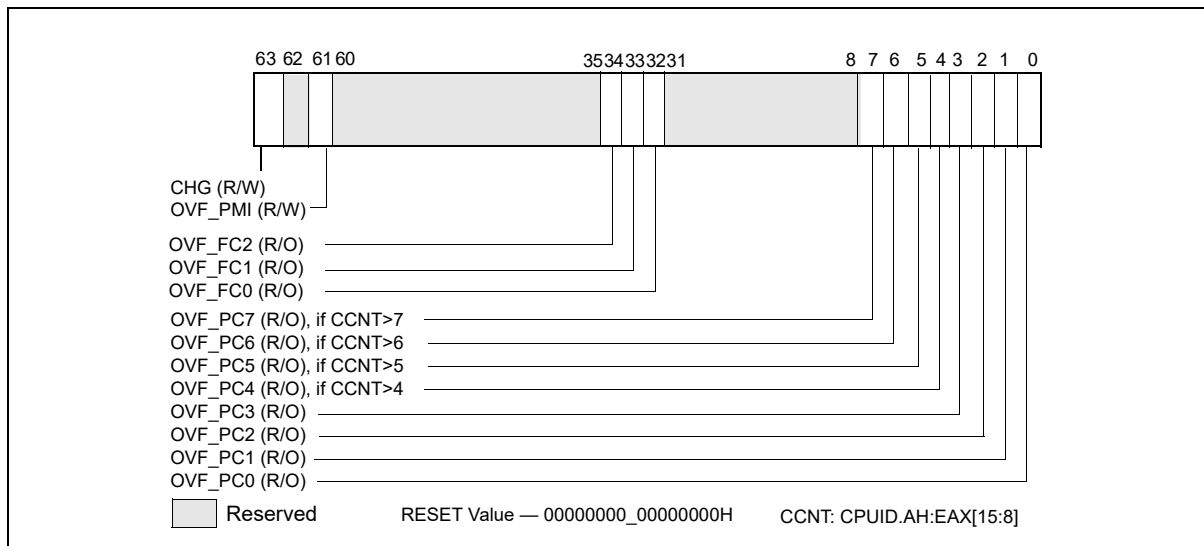


Figure 18-14. IA32\_PERF\_GLOBAL\_STATUS MSR

2. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem; the performance monitoring facilities described in this section generally also apply.

### 18.3.1.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32\_PMCx, associated counter configuration MSRs, IA32\_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32\_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Intel micro-architecture code name Nehalem has been enhanced to include new data format to capture additional information, such as load latency.
- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Intel microarchitecture code name Nehalem. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32\_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32\_PERFEVTSELx.

#### NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 18.2.1.

#### 18.3.1.1.1 Processor Event Based Sampling (PEBS)

All general-purpose performance counters, IA32\_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32\_MISC\_ENABLE[7] and IA32\_MISC\_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32\_PEBS\_ENABLE provides 4 bits that software must use to enable which IA32\_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32\_PEBS\_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32\_PMCx overflow condition. The layout of IA32\_PEBS\_ENABLE for processors based on Intel microarchitecture code name Nehalem is shown in Figure 18-15.

When a counter is enabled to capture machine state (PEBS\_EN\_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32\_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

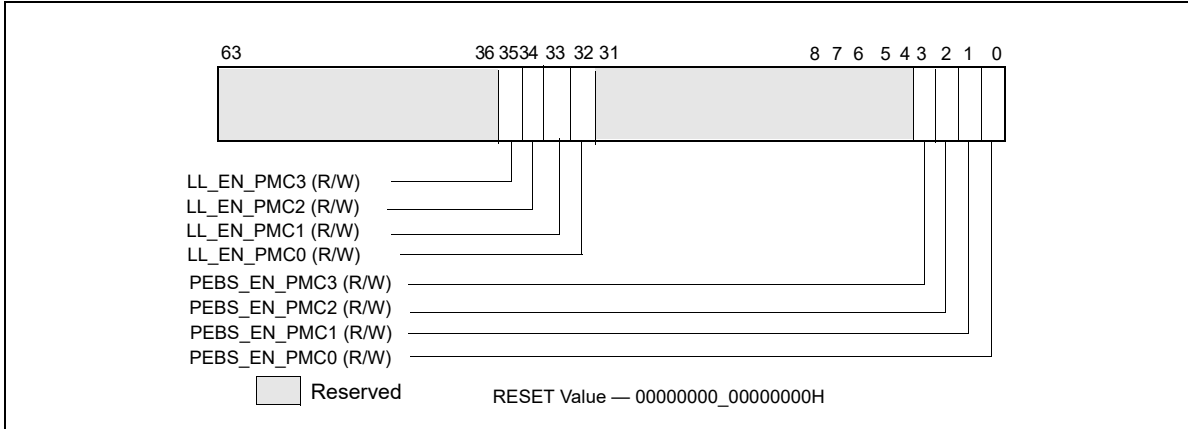


Figure 18-15. Layout of IA32\_PEBS\_ENABLE MSR

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32\_PERF\_CAPABILITIES[11:8] (see Figure 18-63).

The behavior of PEBS assists is reported by IA32\_PERF\_CAPABILITIES[6] (see Figure 18-63). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

```
mov eax, [eax] ; causes PEBS assist
nop
```

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 18-3, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-3. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	58H	R9
08H	R/EIP	60H	R10
10H	R/EAX	68H	R11
18H	R/EBX	70H	R12
20H	R/ECX	78H	R13
28H	R/EDX	80H	R14
30H	R/ESI	88H	R15
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding
50H	R8	A8H	Latency value (core cycles)

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32\_PERF\_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32\_PERF\_GLOBAL\_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

### Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 18-78. In addition to using IA32\_PERFEVTSELx to specify event unit/mask settings and setting the EN\_PMCx bit in the IA32\_PEBS\_ENABLE register for the respective counter, the software must also initialize the DS\_BUFFER\_MANAGEMENT\_AREA data structure in memory to support capturing PEBS records for precise events.

The recording of PEBS records may not operate properly if accesses to the linear addresses in the DS buffer management area or in the PEBS buffer (see below) cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do “lazy” page-table entry propagation for these pages. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.

### NOTE

PEBS events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS\_BUFFER\_MANAGEMENT\_AREA data structure must be programmed into the IA32\_DS\_AREA register. The layout of the DS\_BUFFER\_MANAGEMENT\_AREA is shown in Figure 18-16.

- **PEBS Buffer Base:** This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer.
- **PEBS Index:** This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.
- **PEBS Absolute Maximum:** This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when **PEBS Index** equals **PEBS Absolute Maximum**. No signaling is generated when PEBS buffer is full. Software must reset the **PEBS Index** field to the beginning of the PEBS buffer address to continue capturing PEBS records.

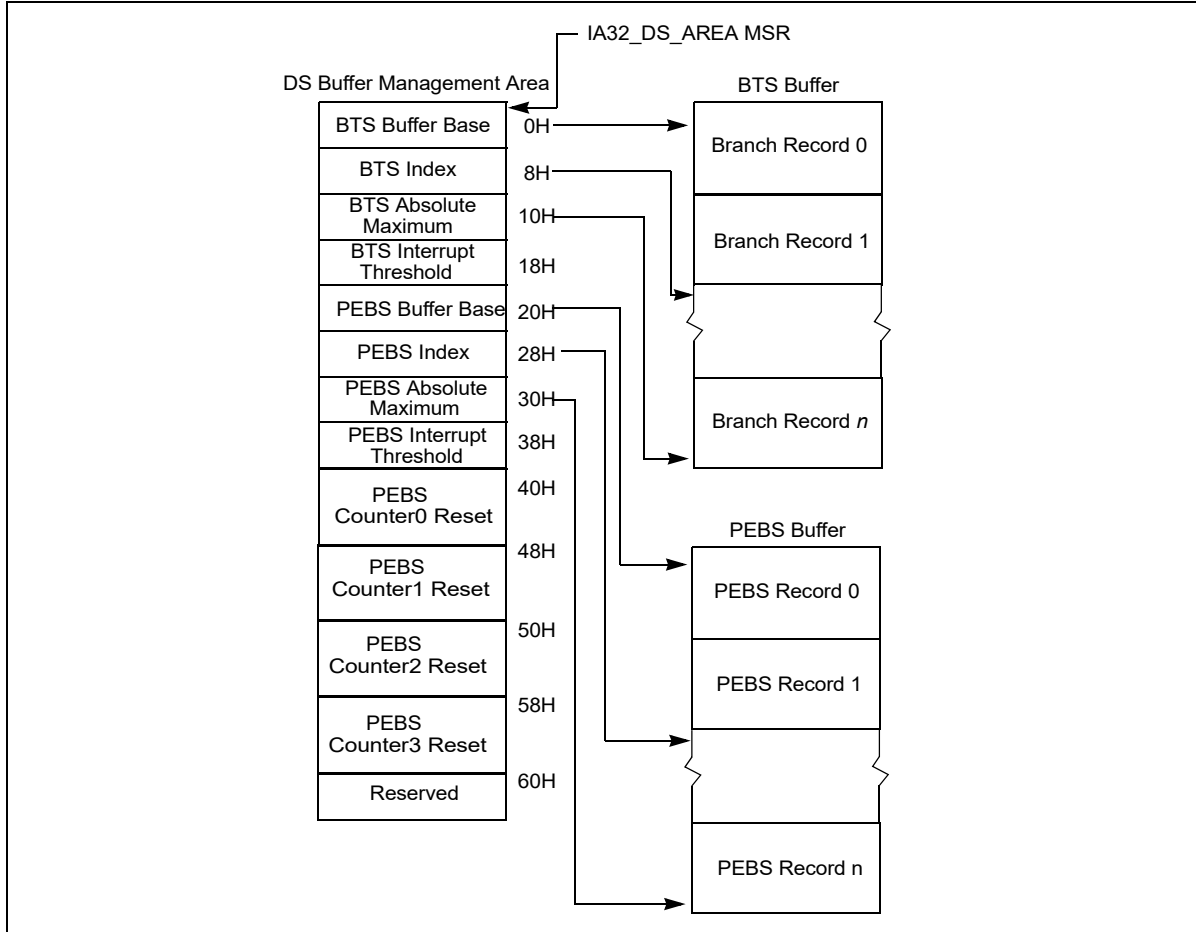


Figure 18-16. PEBS Programming Environment

- PEBS Interrupt Threshold:** This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the **PEBS Index** reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the IA32\_PERF\_GLOBAL\_STATUS.PEBS\_Ovf bit will be set.
- PEBS CounterX Reset:** This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32\_PEBS\_ENABLED was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter IA32\_PMC0 caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter IA32\_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

**Performance Counter Prioritization**

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming IA32\_PerfEvtSelX.INT is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.



Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32\_PMC0 takes precedence over all other counters. Counter IA32\_PMC1 takes precedence over counters IA32\_PMC2 and IA32\_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32\_PMC1 cause an overflow interrupt and IA32\_PMC2 causes an PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the

threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 18.3.1.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

### 18.3.1.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-3. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32\_PERFEVTSELx MSR is programmed to specify the event unit MEM\_INST\_RETIRED, and the LATENCY\_ABOVE\_THRESHOLD event mask must be specified (IA32\_PerfEvtSelX[15:0] = 100H). The corresponding counter IA32\_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32\_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR\_PEBS\_LD\_LAT\_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32\_PEBS\_ENABLE register is set for the corresponding IA32\_PMCx counter register. This means that both the PEBS\_EN\_CTRX and LL\_EN\_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32\_PMC0, the IA32\_PEBS\_ENABLE register must be programmed with the 64-bit value 00000001\_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 18-3, bytes AFH:98H) consists of:

- **Data Linear Address:** This is the linear address of the target of the load operation.
- **Latency Value:** This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.

- Data Source:** The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 18-4. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory refers to system memory physically attached to another processor package.

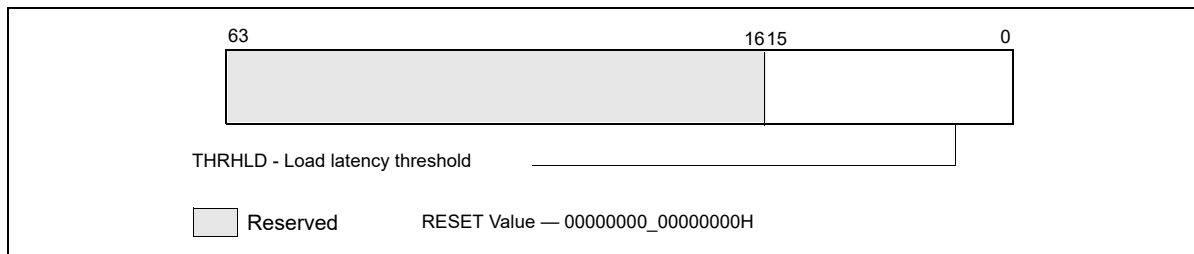
**Table 18-4. Data Source Encoding for Load Latency Record**

Encoding	Description
00H	Unknown L3 cache miss.
01H	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
02H	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
03H	This data request was satisfied by the L2.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found.
07H <sup>1</sup>	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and were serviced by another core with a cross core snoop where modified copies were found.
08H	Reserved/L3 MISS. Local homed requests that missed the L3 cache and were serviced by forwarded data following a cross package snoop where no modified copies were found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation.
0FH	The request was to un-cacheable memory.

**NOTES:**

- Bit 7 is supported only for processors with a CPUID DisplayFamily\_DisplayModel signature of 06\_2A, and 06\_2E; otherwise it is reserved.

The layout of MSR\_PEBS\_LD\_LAT\_THRESHOLD is shown in Figure 18-17.



**Figure 18-17. Layout of MSR\_PEBS\_LD\_LAT MSR**

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32\_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

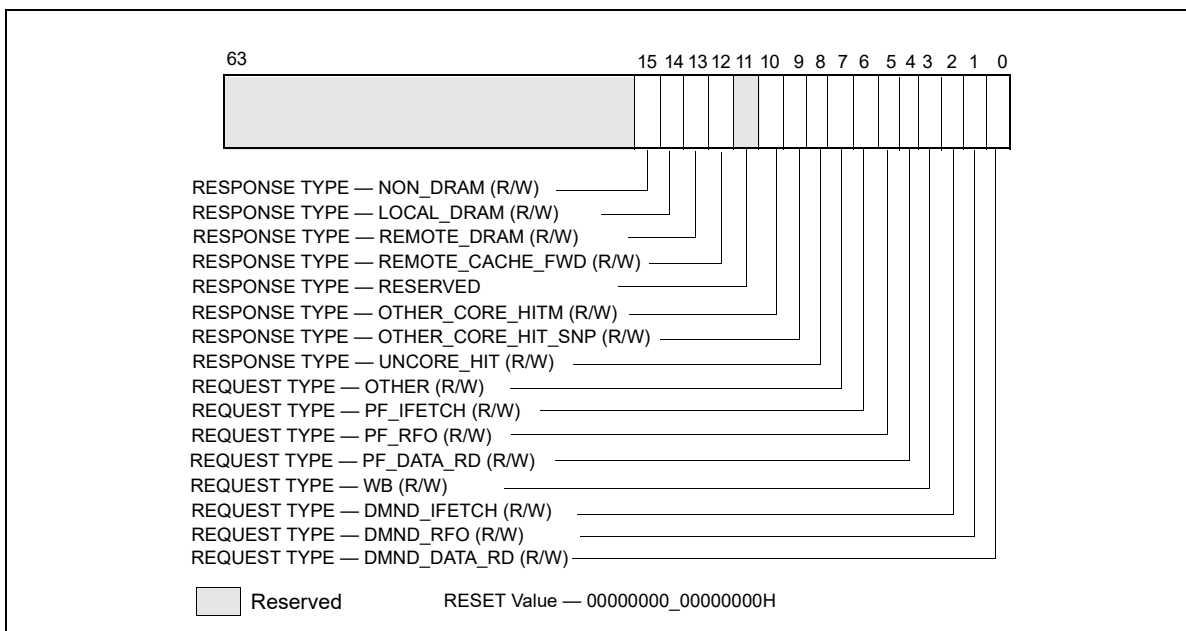
### 18.3.1.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32\_PERFEVTSELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_0. There is only one off-core response configuration MSR. Table 18-5 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

**Table 18-5. Off-Core Response Event Encoding**

Event code in IA32_PERFEVTSELx	Mask Value in IA32_PERFEVTSELx	Required Off-core Response MSR
B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)

The layout of MSR\_OFFCORE\_RSP\_0 is shown in Figure 18-18. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.



**Figure 18-18. Layout of MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 to Configure Off-core Response Events**

**Table 18-6. MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 Bit Field Definition**

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.

**Table 18-6. MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 Bit Field Definition (Contd.)**

Bit Name	Offset	Description
OTHER	7	Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HI T_SNP	9	L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HI TM	10	L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_ FWD	12	L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	Non-DRAM requests that were serviced by IOH.

### 18.3.1.2 Performance Monitoring Facility in the Uncore

The “uncore” in Intel microarchitecture code name Nehalem refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06\_1AH, 06\_1EH, 06\_1FH (see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- Eight General-purpose counters (MSR\_UNCORE\_PerfCntr0 through MSR\_UNCORE\_PerfCntr7). The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR\_UNCORE\_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.
- Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.
- One fixed-function counter, MSR\_UNCORE\_FixedCntr0. The fixed-function uncore counter increments at the rate of the U-clock when enabled.

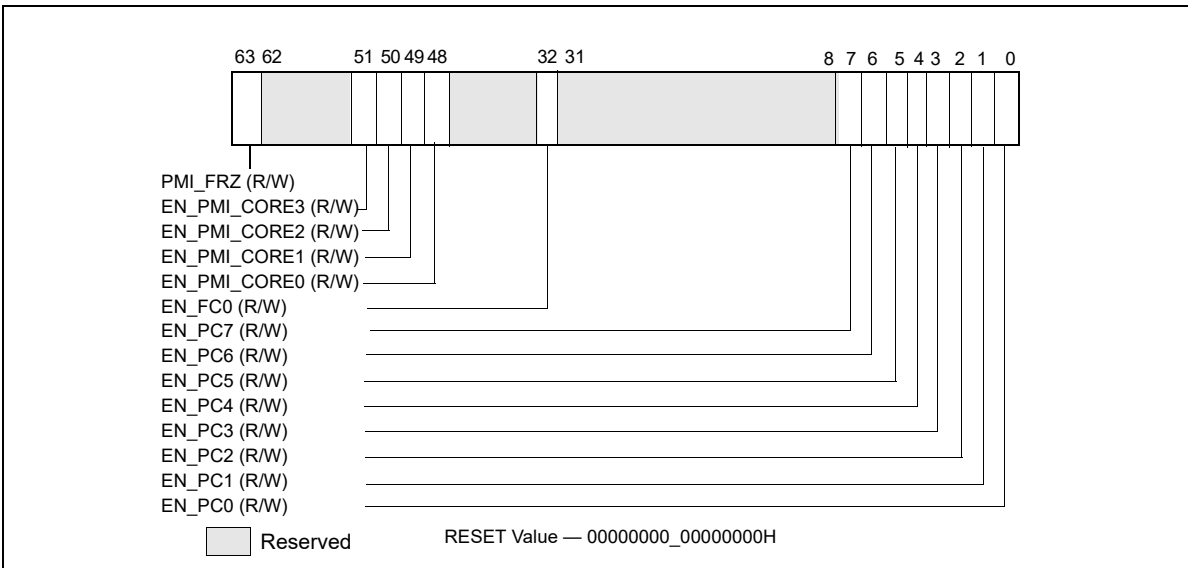
The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

#### 18.3.1.2.1 Uncore Performance Monitoring Management Facility

MSR\_UNCORE\_PERF\_GLOBAL\_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 18-19 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN\_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR\_UNCORE\_PerfCntr n.
- EN\_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0.
- EN\_PMI\_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32\_DEBUGCTL.Offcore\_PMI\_EN to 1.

- **PMI\_FRZ (bit 63):** When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in `MSR_UNCORE_PERF_GLOBAL_CTRL` upon exit from the ISR.



**Figure 18-19. Layout of `MSR_UNCORE_PERF_GLOBAL_CTRL` MSR**

`MSR_UNCORE_PERF_GLOBAL_STATUS` provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 18-20 shows the layout of `MSR_UNCORE_PERF_GLOBAL_STATUS`.

- **OVF\_PCn (bit n, n = 0, 7):** When set, indicates general-purpose uncore counter `MSR_UNCORE_PerfCntr n` has overflowed.
- **OVF\_FC0 (bit 32):** When set, indicates the fixed-function uncore counter `MSR_UNCORE_FixedCntr0` has overflowed.
- **OVF\_PMI (bit 61):** When set indicates that an uncore counter overflowed and generated an interrupt request.
- **CHG (bit 63):** When set indicates that at least one status bit in `MSR_UNCORE_PERF_GLOBAL_STATUS` register has changed state.

MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear the status bits in the UNCORE\_PERF\_GLOBAL\_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

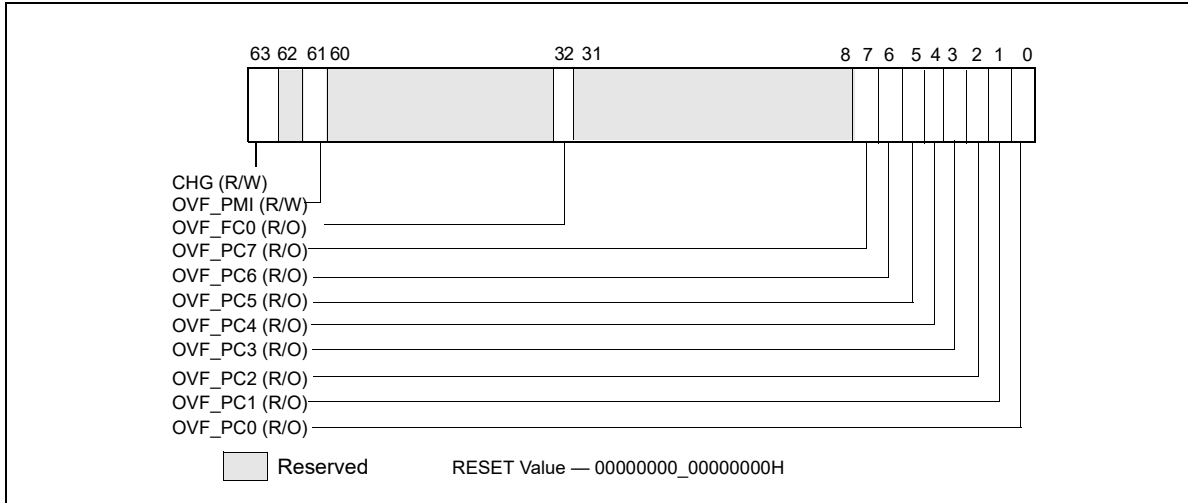


Figure 18-20. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_STATUS MSR

Figure 18-21 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL.

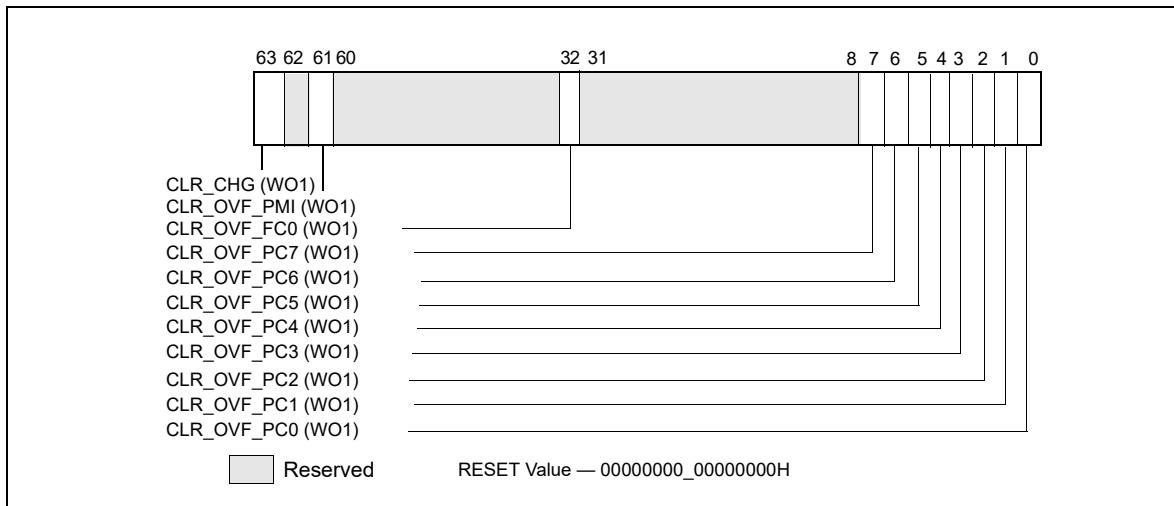


Figure 18-21. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL MSR

- CLR\_OVF\_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR\_UNCORE\_PerfCntr n. Writing a value other than 1 is ignored.
- CLR\_OVF\_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0. Writing a value other than 1 is ignored.
- CLR\_OVF\_PMI (bit 61): Set this bit to clear the OVF\_PMI flag in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS. Writing a value other than 1 is ignored.
- CLR\_CHG (bit 63): Set this bit to clear the CHG flag in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS register. Writing a value other than 1 is ignored.

### 18.3.1.2.2 Uncore Performance Event Configuration Facility

MSR\_UNCORE\_PerfEvtSel0 through MSR\_UNCORE\_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR\_UNCORE\_PerfEvtSelx and counting must be enabled using the respective EN\_PCx bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL. Figure 18-22 shows the layout of MSR\_UNCORE\_PERFEVTSELx.

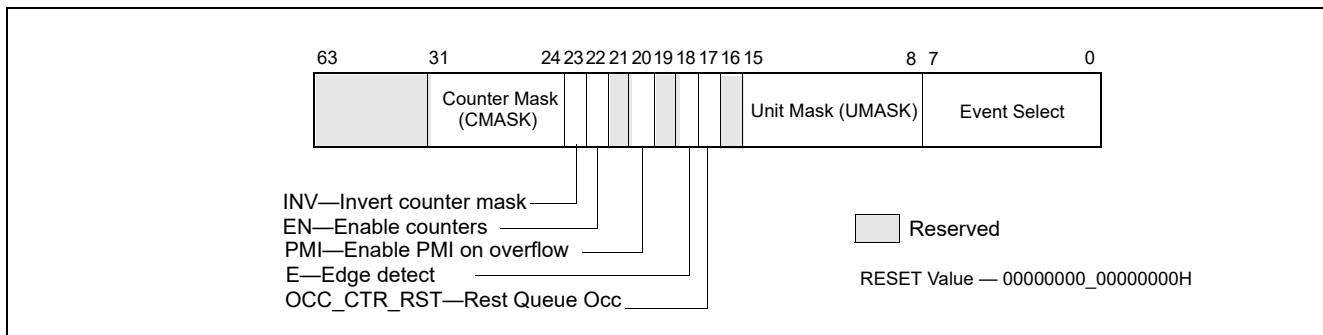


Figure 18-22. Layout of MSR\_UNCORE\_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC\_CTR\_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN\_PMI\_COREx) in the register MSR\_UNCORE\_PERF\_GLOBAL\_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN\_PCx bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 18-23 shows the layout of MSR\_UNCORE\_FIXED\_CTR\_CTRL.

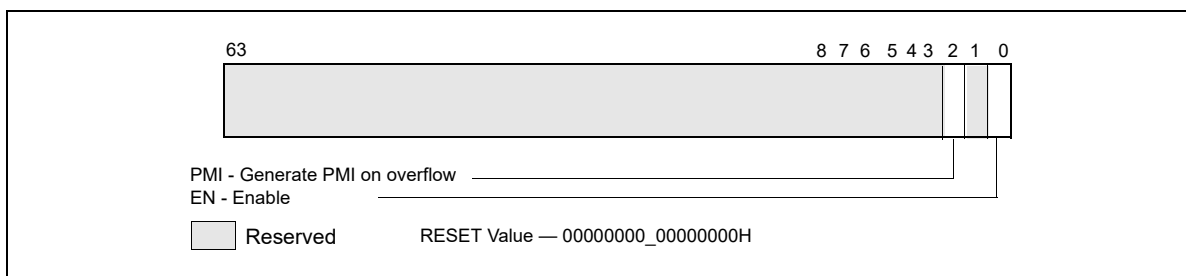


Figure 18-23. Layout of MSR\_UNCORE\_FIXED\_CTR\_CTRL MSR

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN\_FC0 bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN\_PMI\_COREx) in the register MSR\_UNCORE\_PERF\_GLOBAL\_CTRL.

Both the general-purpose counters (MSR\_UNCORE\_PerfCnt) and the fixed-function counter (MSR\_UNCORE\_FixedCnt0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

### 18.3.1.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR\_UNCORE\_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR\_OPCODE\_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR\_UNCORE\_ADDR\_OPCODE\_MATCH register. The layout is shown in Figure 18-24.

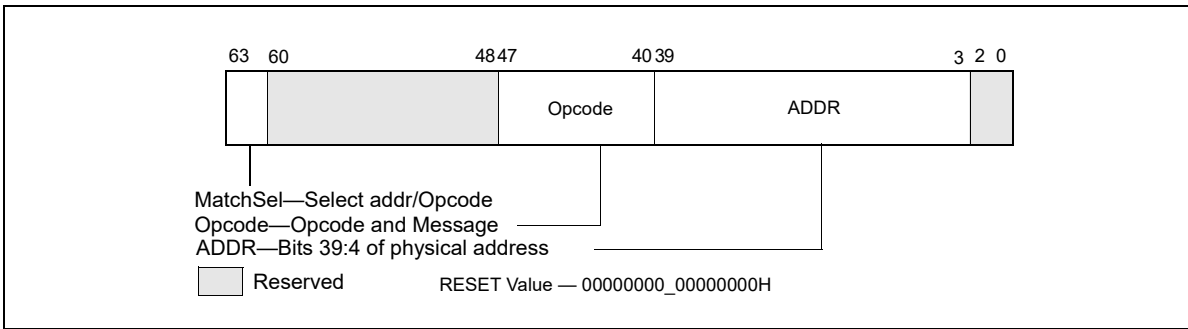


Figure 18-24. Layout of MSR\_UNCORE\_ADDR\_OPCODE\_MATCH MSR

- Addr (bits 39:3): The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - Bits 43:40 specify the QPI packet header opcode.
  - Bits 47:44 specify the QPI message classes.

Table 18-7 lists the encodings supported in the opcode field.



**Table 18-7. Opcode Field Encoding for MSR\_UNCORE\_ADDR\_OPCODE\_MATCH**

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr\_opcode match hardware.
  - 100B: Count if only the address field matches.
  - 010B: Count if only the opcode field matches.
  - 110B: Count if either opcode field matches or the address field matches.
  - 001B: Count only if both opcode and address field match.
  - Other encoding are reserved.

### 18.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different. The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 18-25.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

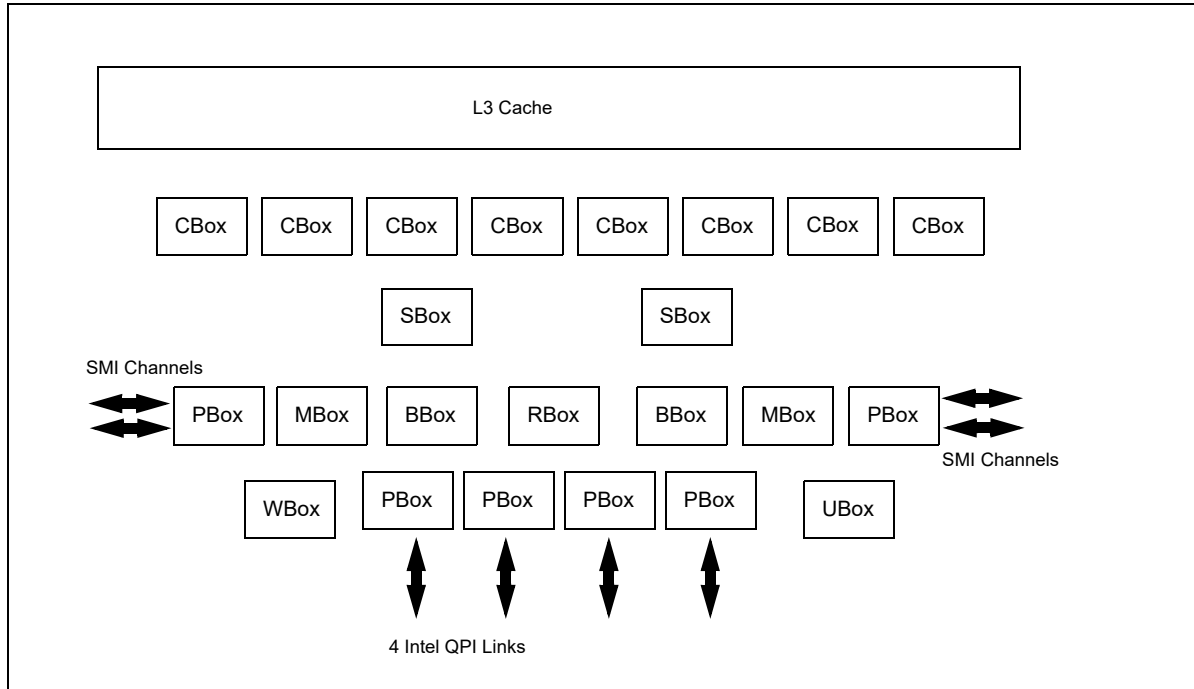


Figure 18-25. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 18-8 summarizes the number MSRs for uncore PMU for each box.

Table 18-8. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 ( 2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the “global control” programming interface, IA32\_PERF\_GLOBAL\_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR\_U\_PMON\_GLOBAL\_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, Table 2-17 under the general naming style of MSR\_%box#%\_PMON\_%scope\_function%, where %box#% designates the type of box and zero-based index if there are more than one box of the same type, %scope\_function% follows the examples below:

- Multi-counter enabling MSRs: MSR\_U\_PMON\_GLOBAL\_CTL, MSR\_S0\_PMON\_BOX\_CTL, MSR\_C7\_PMON\_BOX\_CTL, etc.
- Multi-counter status MSRs: MSR\_U\_PMON\_GLOBAL\_STATUS, MSR\_S0\_PMON\_BOX\_STATUS, MSR\_C7\_PMON\_BOX\_STATUS, etc.
- Multi-counter overflow control MSRs: MSR\_U\_PMON\_GLOBAL\_OVF\_CTL, MSR\_S0\_PMON\_BOX\_OVF\_CTL, MSR\_C7\_PMON\_BOX\_OVF\_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g. MSR\_U\_PMON\_CTR, MSR\_S0\_PMON\_CTR0, MSR\_C7\_PMON\_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g. MSR\_U\_PMON\_EVNT\_SEL, MSR\_S0\_PMON\_EVNT\_SEL0, MSR\_C7\_PMON\_EVNT\_SEL5, etc.
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g. MSR\_M0\_PMON\_TIMESTAMP, MSR\_R0\_PMON\_IPERF0\_P1, MSR\_S1\_PMON\_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document “Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide”.

### 18.3.2 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 18.6.3 also apply to processors based on Intel® microarchitecture code name Westmere.

Table 18-5 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR\_OFFCORE\_RSP\_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR\_OFFCORE\_RSP\_1 (address 1A7H) are supported in processors based on Intel microarchitecture code name Westmere. The event code and event mask definitions of non-architectural performance monitoring events can be found at: <https://perfmon-events.intel.com/>.

The load latency facility is the same as described in Section 18.3.1.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB\_MISS and LOCK, see Table 18-13.

### 18.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series<sup>3</sup>. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore subsystem is similar to that shown in Figure 18-25, with the additional capability that up to 10 C-Box units are supported.

3. Exceptions are indicated for event code 0FH in the event list for this processor (<https://perfmon-events.intel.com/>); and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 18-13.

Table 18-9 summarizes the number MSRs for uncore PMU for each box.

**Table 18-9. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 ( 2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the “Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual”.

### 18.3.4 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Intel microarchitecture code name Sandy Bridge; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU’s capability is similar to those described in Section 18.3.1.1 and Section 18.6.3, with some differences and enhancements relative to Intel microarchitecture code name Westmere summarized in Table 18-10.

**Table 18-10. Core PMU Comparison**

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to determine # of counters. See Section 18.2.1.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12.	See Table 18-78.	IA32_PMC4-IA32_PMC7 do not support PEBS.

Table 18-10. Core PMU Comparison (Contd.)

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
PEBS-Load Latency	See Section 18.3.4.4.2; <ul style="list-style-type: none"> <li>Data source encoding</li> <li>STLB miss encoding</li> <li>Lock transaction encoding</li> </ul>	Data source encoding	
PEBS-Precise Store	Section 18.3.4.4.3	No	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL).	No	
Off-core Response Event	MSR 1A6H and 1A7H, extended request and response types.	MSR 1A6H and 1A7H, limited response types.	Nehalem supports 1A6H only.

### 18.3.4.1 Global Counter Control Facilities In Intel® Microarchitecture Code Name Sandy Bridge

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Intel microarchitecture code name Sandy Bridge. Software must use CPUID to determine the number performance counters/event select registers (See Section 18.2.1.1).

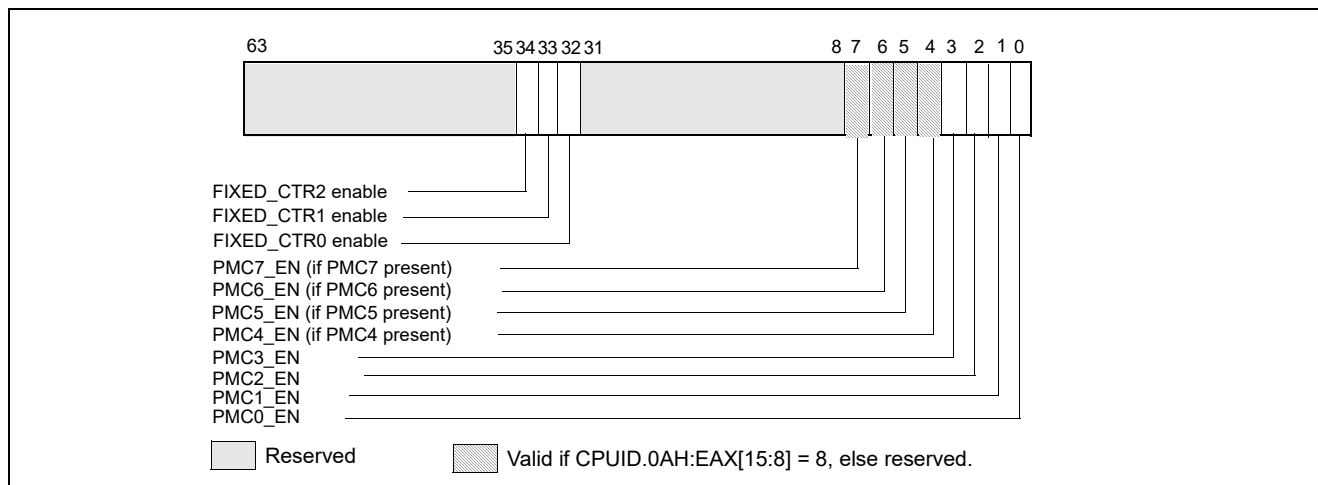


Figure 18-26. IA32\_PERF\_GLOBAL\_CTRL MSR in Intel® Microarchitecture Code Name Sandy Bridge

Figure 18-42 depicts the layout of IA32\_PERF\_GLOBAL\_CTRL MSR. The enable bits (PMC4\_EN, PMC5\_EN, PMC6\_EN, PMC7\_EN) corresponding to IA32\_PMC4-IA32\_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32\_PERF\_GLOBAL\_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or IA32\_PERF\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

IA32\_PERF\_GLOBAL\_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32\_PERF\_GLOBAL\_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 18-27). A value of 1 in each bit of the PMCx\_OVF field indicates an overflow condition has occurred in the associated counter.

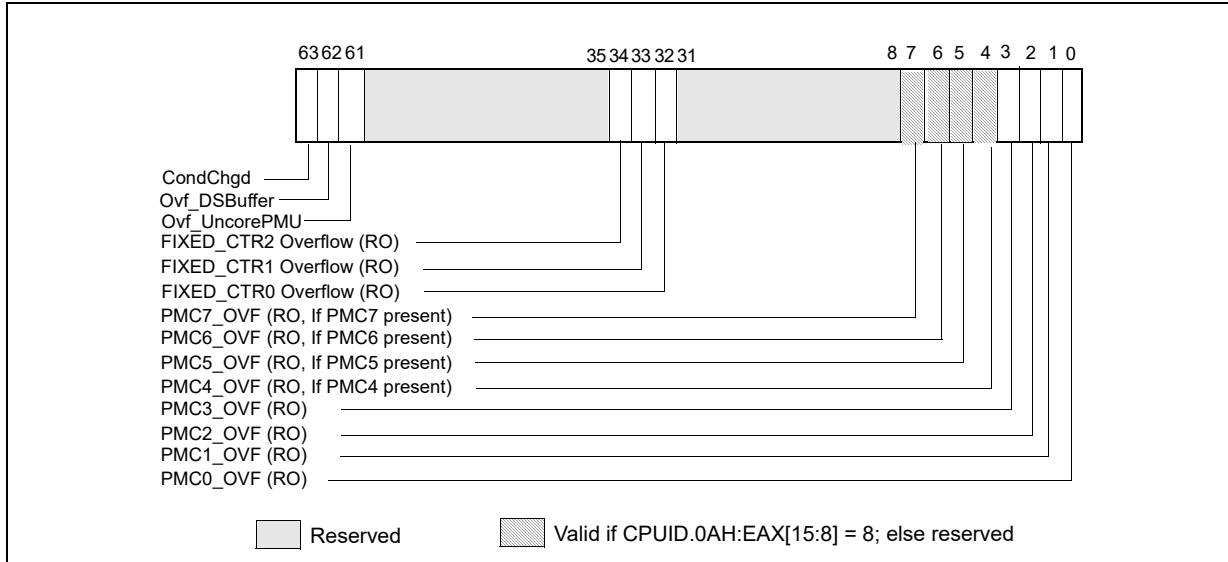


Figure 18-27. IA32\_PERF\_GLOBAL\_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR\_PERF\_GLOBAL\_STATUS.

IA32\_PERF\_GLOBAL\_OVF\_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-28). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling.
- Reloading counter values to continue sampling.
- Disabling event counting or interrupt based sampling.

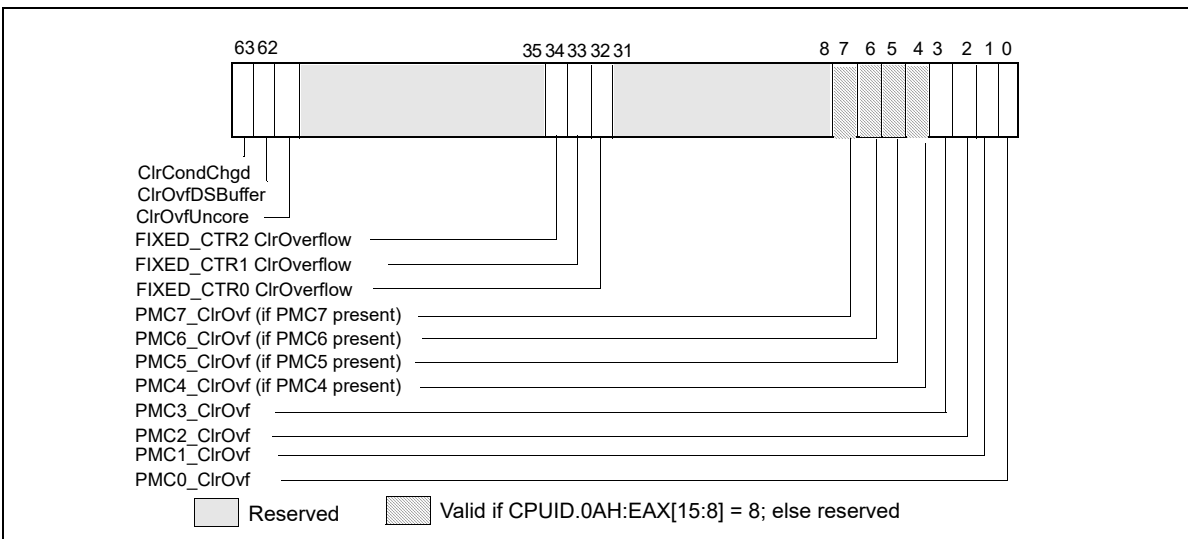


Figure 18-28. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR in Intel microarchitecture code name Sandy Bridge

### 18.3.4.2 Counter Coalescence

In processors based on Intel microarchitecture code name Sandy Bridge, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report the number of counters visible to software.

If a processor core is shared by two logical processors, each logical processors can access up to four counters (IA32\_PMC0-IA32\_PMC3). This is the same as in the prior generation for processors based on Intel microarchitecture code name Nehalem.

If a processor core is not shared by two logical processors, up to eight general-purpose counters are visible. If CPUID.0AH:EAX[15:8] reports 8 counters, then IA32\_PMC4-IA32\_PMC7 would occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32\_PERFEVTSEL4-IA32\_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32\_PMC4-IA32\_PMC7, IA32\_PMC4-IA32\_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32\_PERF\_GLOBAL\_CTRL, IA32\_PERF\_GLOBAL\_STATUS, or IA32\_PERF\_GLOBAL\_OVF\_CTL will also cause #GP.

### 18.3.4.3 Full Width Writes to Performance Counters

Processors based on Intel microarchitecture code name Sandy Bridge support full-width writes to the general-purpose counters, IA32\_PMCx. Support of full-width writes are enumerated by IA32\_PERF\_CAPABILITIES.FW\_WRITES[13] (see Section 18.2.4).

The default behavior of IA32\_PMCx is unchanged, i.e. WRMSR to IA32\_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32\_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32\_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32\_A\_PMCx by testing IA32\_PERF\_CAPABILITIES[13].

### 18.3.4.4 PEBS Support in Intel® Microarchitecture Code Name Sandy Bridge

Processors based on Intel microarchitecture code name Sandy Bridge support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Westmere is summarized in Table 18-11.

**Table 18-11. PEBS Facility Comparison**

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-29	Figure 18-15	
PEBS record layout	Physical Layout same as Table 18-3.	Table 18-3	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 18-12.	See Table 18-78.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-13.	Table 18-4	
PEBS-Precise Store	Yes; see Section 18.3.4.4.3.	No	IA32_PMC3 only
PEBS-PDIR	Yes	No	IA32_PMC1 only
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

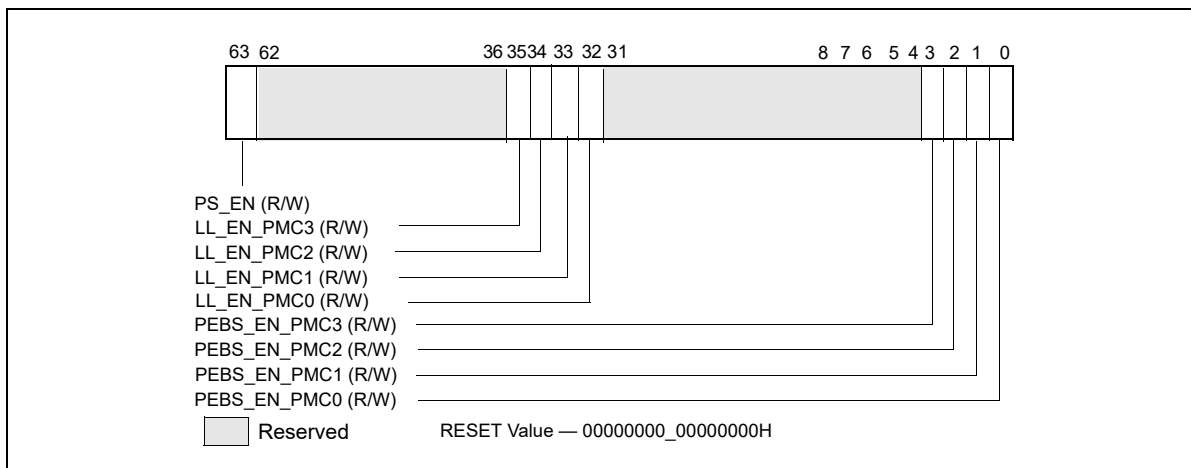
Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

**NOTE**

PEBS events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32\_PEBS\_ENABLE MSR, bit 63 is defined as PS\_ENABLE: When set, this enables IA32\_PMC3 to capture precise store information. Only IA32\_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32\_PEBS\_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.



**Figure 18-29. Layout of IA32\_PEBS\_ENABLE MSR**

**18.3.4.4.1 PEBS Record Format**

The layout of PEBS records physically identical to those shown in Table 18-3, but the fields at offset 98H, A0H and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H): When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 18-4. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32\_PERF\_GLOBAL\_STATUS corresponding to those counters that both overflowed and were enabled in the IA32\_PEBS\_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Intel microarchitecture code name Sandy Bridge is shown in Table 18-12.



Table 18-12. PEBS Performance Events for Intel® Microarchitecture Code Name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	COH	PREC_DIST	01H <sup>1</sup>
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H
MEM_UOPS_RETIRED	DOH	STLB_MISS_LOADS	11H
		STLB_MISS_STORES	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

**NOTES:**

1. Only available on IA32\_PMC1.

### 18.3.4.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Intel microarchitecture code name Sandy Bridge is similar to that in prior microarchitecture. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-3 and Section 18.3.4.4.1. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32\_PERFEVTSELx MSR is programmed to specify the event unit MEM\_TRANS\_RETIRED, and the LATENCY\_ABOVE\_THRESHOLD event mask must be specified (IA32\_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32\_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is

programmed. The CMASK or INV fields of the IA32\_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.

- The MSR\_PEBS\_LD\_LAT\_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32\_PEBS\_ENABLE register is set for the corresponding IA32\_PMCx counter register. This means that both the PEBS\_EN\_CTRX and LL\_EN\_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32\_PMC0, the IA32\_PEBS\_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM\_TRANS\_RETIRE event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

**Table 18-13. Layout of Data Source Field of Load Latency Record**

Field	Position	Description
Source	3:0	See Table 18-4
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR\_PEBS\_LD\_LAT\_THRESHOLD is the same as shown in Figure 18-17.

### 18.3.4.4.3 Precise Store Facility

Processors based on Intel microarchitecture code name Sandy Bridge offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32\_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.

- Program the MEM\_TRANS\_RETIRED.PRECISE\_STORE event in IA32\_PERFEVTSEL3. Only counter 3 (IA32\_PMC3) supports collection of precise store information.
- Set IA32\_PEBS\_ENABLE[3] and IA32\_PEBS\_ENABLE[63]. This enables IA32\_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

**Table 18-14. Layout of Precise Store Information In PEBS Record**

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	<b>L1D Hit</b> (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. <b>STLB Miss</b> (bit 4): The store missed the STLB if set, otherwise the store hit the STLB <b>Locked Access</b> (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Reserved	A8H	Reserved

#### 18.3.4.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST\_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable “skids” in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the “skid” problem by providing an early indication of when the INST\_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow. On processors based on Intel microarchitecture code name Sandy Bridge skid is significantly reduced, and can be as little as one instruction. On future implementations PDIR may eliminate skid.

PDIR applies only to the INST\_RETIRED.ALL precise event, and processors based on Sandy Bridge microarchitecture must use IA32\_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32\_PEBS\_ENABLE set to 1. INST\_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily\_DisplayModel signatures of 06\_2A and 06\_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

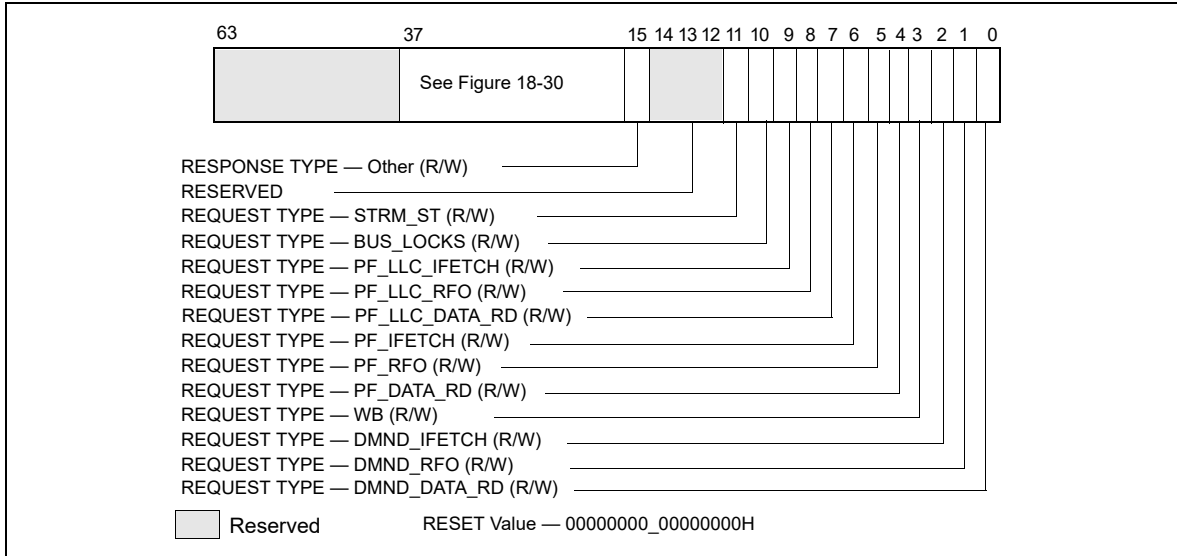
#### 18.3.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Table 18-15 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

**Table 18-15. Off-Core Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMCO-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 are shown in Figure 18-30 and Figure 18-31. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.



**Figure 18-30. Request\_Type Fields for MSR\_OFFCORE\_RSP\_x**

**Table 18-16. MSR\_OFFCORE\_RSP\_x Request\_Type Field Definition**

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests
STRM_ST	11	Streaming store requests
OTHER	15	Any other request that crosses IDI, including I/O.

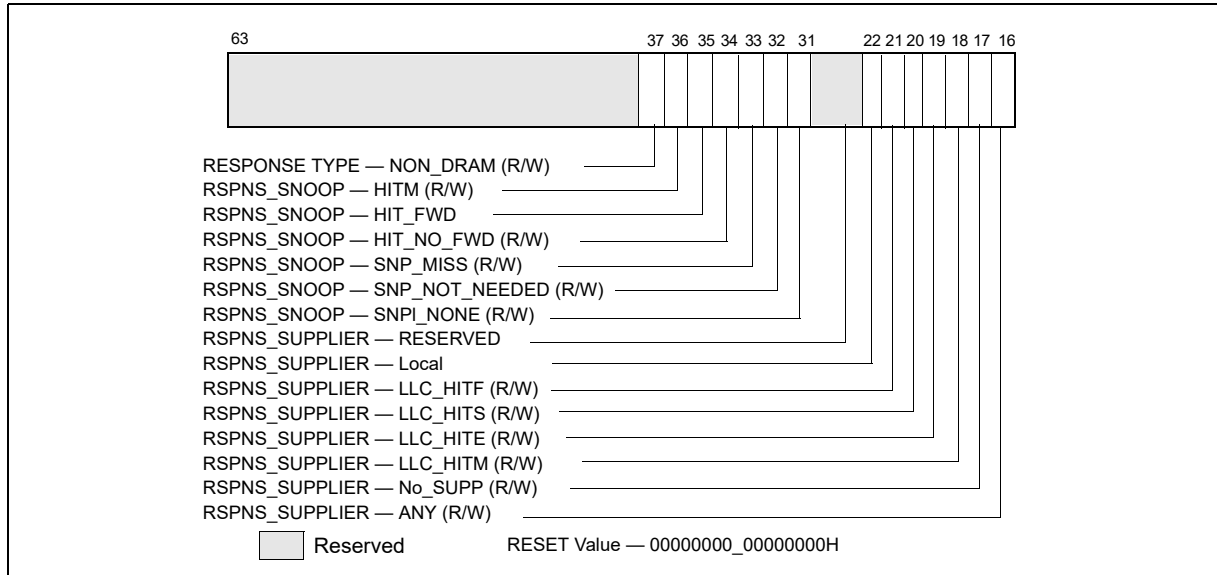


Figure 18-31. Response\_Supplier and Snoop Info Fields for MSR\_OFFCORE\_RSP\_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR\_OFFCORE\_RSP\_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-17. MSR\_OFFCORE\_RSP\_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

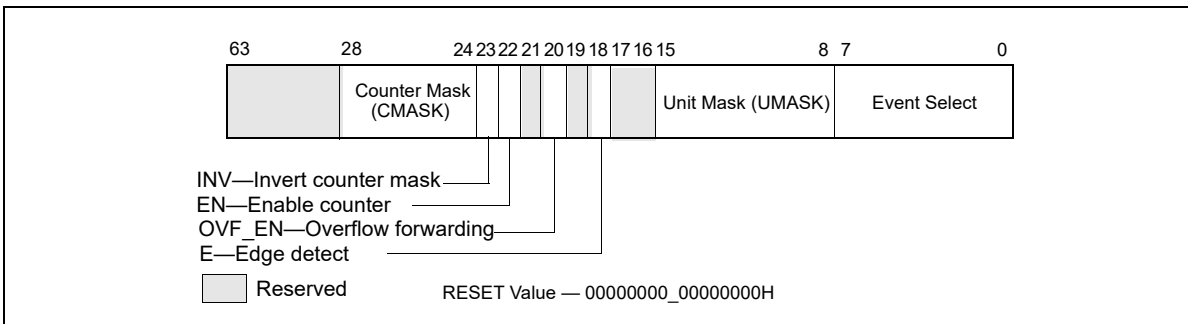
If “ANY” bit is set, the supplier and snoop info bits are ignored.

**Table 18-18. MSR\_OFFCORE\_RSP\_x Snoop Info Field Definition**

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	SNP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNP_MISS	33	A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.

**18.3.4.6 Uncore Performance Monitoring Facilities In Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series**

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 18.3.1.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 18-32.



**Figure 18-32. Layout of Uncore PERFVTSSEL MSR for a C-Box Unit or the ARB Unit**

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event\_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see the event list at: <https://perfmon-events.intel.com/>.
- E (bit 18): Enables edge detection filtering, if 1.
- OVF\_EN (bit 20): Enables the overflow indicator from the uncore counter forwarded to MSR\_UNC\_PERF\_GLOBAL\_CTRL, if 1.
- EN (bit 22): Enables the local counter associated with this event select MSR.
- INV (bit 23): Event count increments with non-negative value if 0, with negated value if 1.
- CMASK (bits 28:24): Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-33 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR\_UNC\_PERF\_GLOBAL\_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32\_DEBUGCTL (at address 1D9H) to enable this capability.

- PMI\_SEL\_Core#: Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is '1', a wake request is sent to the respective processor core prior to sending the PMI.
- EN: Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- WakePMI: Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- FREEZE: Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

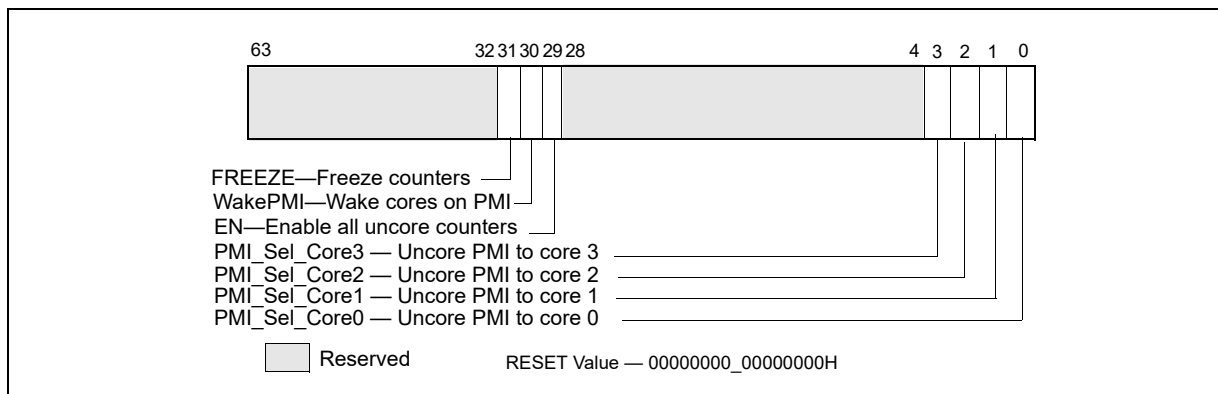


Figure 18-33. Layout of MSR\_UNC\_PERF\_GLOBAL\_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSRs for uncore PMU for each box.

**Table 18-19. Uncore PMU MSR Summary**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, see Table 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

#### 18.3.4.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.
- Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE\_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE\_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic cannot associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

#### 18.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Intel microarchitecture code name Sandy Bridge-E. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 18.3.4.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 18.6.3 through Section 18.3.4.5. However, the MSR\_OFFCORE\_RSP\_0/MSR\_OFFCORE\_RSP\_1 Response Supplier Info field shown in Table 18-17 applies to Intel Core Processors with CPUID signature of DisplayFamily\_DisplayModel encoding of 06\_2AH; Intel Xeon processor with CPUID signature of DisplayFamily\_DisplayModel encoding of 06\_2DH supports an additional field for remote DRAM controller shown in Table 18-20. Additionally, there are some small differences in the non-architectural performance monitoring events (see [event list available at: https://perfmon-events.intel.com/](https://perfmon-events.intel.com/)).



**Table 18-20. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definitions**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Remote	30:23	Remote DRAM Controller (either all 0s or all 1s).

### 18.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 18-21 summarizes the uncore PMU facilities providing MSR interfaces.

**Table 18-21. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	4	44	Yes	per-box	None
PCU	1	4	48	Yes	per-box	Match/Mask
U-Box	1	2	44	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in “Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-24.

### 18.3.5 3rd Generation Intel® Core™ Processor Performance Monitoring Facility

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 18.6.3 through Section 18.3.4.5. The non-architectural performance monitoring events supported by the processor core can be found at: <https://perfmon-events.intel.com/>.

#### 18.3.5.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in “Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-28.

### 18.3.6 4th Generation Intel® Core™ Processor Performance Monitoring Facility

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU’s capability is similar to those described in Section 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.3.6.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

**Table 18-22. Core PMU Comparison**

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to determine # of counters. See Section 18.2.1.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12 and Section 18.3.6.5.1.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 18.3.4.4.3	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL)	Yes (using precise INST_RETIRED.ALL)	
PEBS-EventingIP	Yes	No	
Data Address Profiling	Yes	No	
LBR Profiling	Yes	Yes	
Call Stack Profiling	Yes, see Section 17.11.	No	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; extended request and response types.	MSR 1A6H and 1A7H; extended request and response types.	
Intel TSX support for Perfmon	See Section 18.3.6.5.	No	

### 18.3.6.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Intel micro-architecture code name Sandy Bridge, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Sandy Bridge is summarized in Table 18-23.

**Table 18-23. PEBS Facility Comparison**

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-15	Figure 18-29	
PEBS record layout	Table 18-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 18-3; enhanced fields at offsets 98H, A0H, A8H.	
Precise Events	See Table 18-12.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-13.	Table 18-13	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 18.3.4.4.3.	
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

#### NOTE

PEBS events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

### 18.3.6.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 18-24. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

**Table 18-24. PEBS Record Format for 4th Generation Intel Core Processor Family**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Data Linear Address
40H	R/EBP	A0H	Data Source Encoding
48H	R/ESP	A8H	Latency value (core cycles)
50H	R8	B0H	EventingIP
58H	R9	B8H	TX Abort Information (Section 18.3.6.5.1)

The layout of PEBS records are almost identical to those shown in Table 18-3. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and the equivalent capability of precise store in prior generation (see Section 18.3.6.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

### 18.3.6.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

**Table 18-25. Precise Events That Supports Data Linear Address Profiling**

Event Name	Event Name
MEM_UOPS_RETIRED.STLB_MISS_LOADS	MEM_UOPS_RETIRED.STLB_MISS_STORES
MEM_UOPS_RETIRED.LOCK_LOADS	MEM_UOPS_RETIRED.SPLIT_STORES
MEM_UOPS_RETIRED.SPLIT_LOADS	MEM_UOPS_RETIRED.ALL_STORES
MEM_UOPS_RETIRED.ALL_LOADS	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM
MEM_LOAD_UOPS_RETIRED.L1_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT
MEM_LOAD_UOPS_RETIRED.L3_HIT	MEM_LOAD_UOPS_RETIRED.L1_MISS
MEM_LOAD_UOPS_RETIRED.L2_MISS	MEM_LOAD_UOPS_RETIRED.L3_MISS
MEM_LOAD_UOPS_RETIRED.HIT_LFB	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS

**Table 18-25. Precise Events That Supports Data Linear Address Profiling (Contd.)**

Event Name	Event Name
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM
UOPS_RETIRED.ALL (if load or store is tagged)	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE

DataLA can use any one of the IA32\_PMC0-IA32\_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program an event listed in Table 18-25 using any one of IA32\_PERFVTSEL0-IA32\_PERFVTSEL3.
- Set the corresponding IA32\_PEBS\_ENABLE.PEBS\_EN\_CTRx bit. This enables the corresponding IA32\_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.

**Table 18-26. Layout of Data Linear Address Information In PEBS Record**

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	<ul style="list-style-type: none"> <li>▪ <b>DCU Hit</b> (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES</li> <li>▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 18-25.</li> </ul>
Reserved	A8H	Always zero.

### 18.3.6.3.1 EventingIP Record

The PEBS record layout for processors based on Intel microarchitecture code name Haswell adds a new field at offset 0B0H. This is the eventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

### 18.3.6.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. The event codes are listed in Table 18-15. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Software must program MSR\_OFFCORE\_RSP\_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-27.
- Supplier information (bits 30:16): see Table 18-28.
- Snoop response information (bits 37:31): see Table 18-18.

**Table 18-27. MSR\_OFFCORE\_RSP\_x Request\_Type Definition (Haswell microarchitecture)**

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
COREWB	3	Counts the number of modified cachelines written back.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_LOCK	10	Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	Counts the number of streaming store requests electronically.
Reserved	14:12	Reserved
OTHER	15	Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 18-28. The fields vary across products (according to CPUID signatures) and is noted in the description.

**Table 18-28. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CPUID Signature 06\_3CH, 06\_46H)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

**Table 18-29. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CUID Signature 06\_45H)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	L4 Cache
	Reserved	30:26	Reserved

#### 18.3.6.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 18-28 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CUID signature 06\_3FH).

**Table 18-30. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	L3_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

#### 18.3.6.5 Performance Monitoring and Intel® TSX

Chapter 16 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* describes the details of Intel® Transactional Synchronization Extensions (Intel® TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances its IA32\_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32\_PERFEVTSELx is shown in Figure 18-34. The two additional bit fields are:

- **IN\_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN\_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32\_PERFEVTSEL2.

When the IA32\_PERFEVTSELx MSR is programmed with both IN\_TX=0 and IN\_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN\_TXCP bit. See Table 2-29.

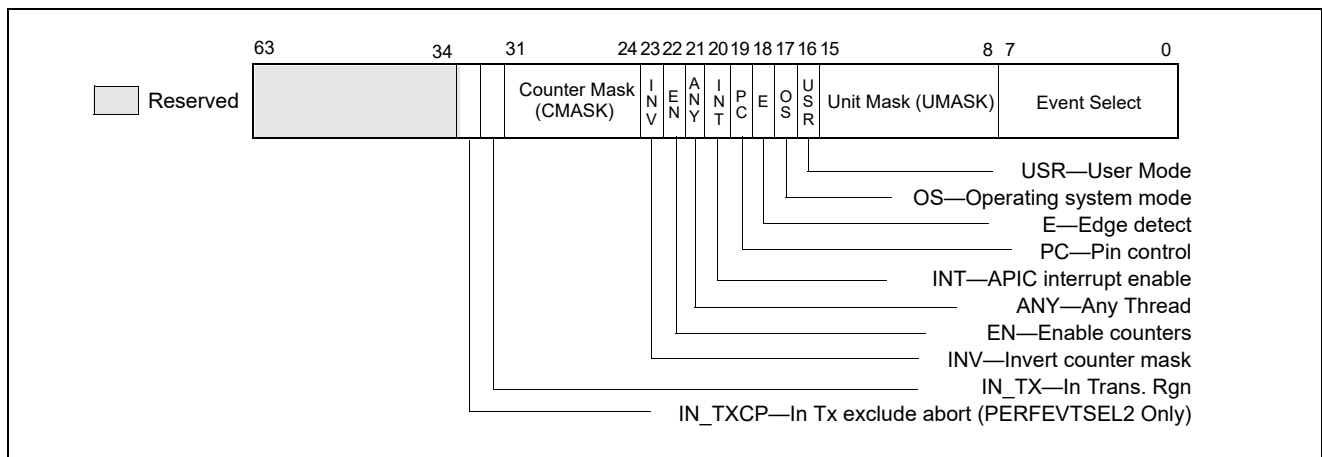


Figure 18-34. Layout of IA32\_PERFEVTSELx MSRs Supporting Intel TSX

A common usage of setting IN\_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32\_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN\_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32\_PERFEVTSELx.IN\_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32\_PERFEVTSELx but occurring outside a transactional region are discarded.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they can be found at: <https://perfmon-events.intel.com/>.

### 18.3.6.5.1 Intel TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events also support using the PEBS facility to capture additional information. They are:

- HLE\_RETIRED.ABORTED (encoding C8H mask 04H),
- RTM\_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE\_RETIRED.ABORTED,RTM\_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.



Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pending at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 18-31.

**Table 18-31. TX Abort Information Field Definition**

Bit Name	Offset	Description
Cycles_Last_TX	31:0	The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
HLE_Abort	32	If set, the abort information corresponds to an aborted HLE execution
RTM_Abort	33	If set, the abort information corresponds to an aborted RTM execution
Instruction_Abort	34	If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.
Non_Instruction_Abort	35	If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.
Retry	36	If set, retrying the transactional execution may have succeeded.
Data_Conflict	37	If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.
Capacity Writes	38	If set, the transactional region aborted due to exceeding resources for transactional writes.
Capacity Reads	39	If set, the transactional region aborted due to exceeding resources for transactional reads.
Reserved	63:40	Reserved

### 18.3.6.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 18.3.4.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 18-32.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-33 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSRs for uncore PMU for each box.

**Table 18-32. Uncore PMU MSR Summary**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, see Table 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

The uncore performance events for the C-Box and ARB units can be found at: <https://perfmon-events.intel.com/>.

### 18.3.6.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

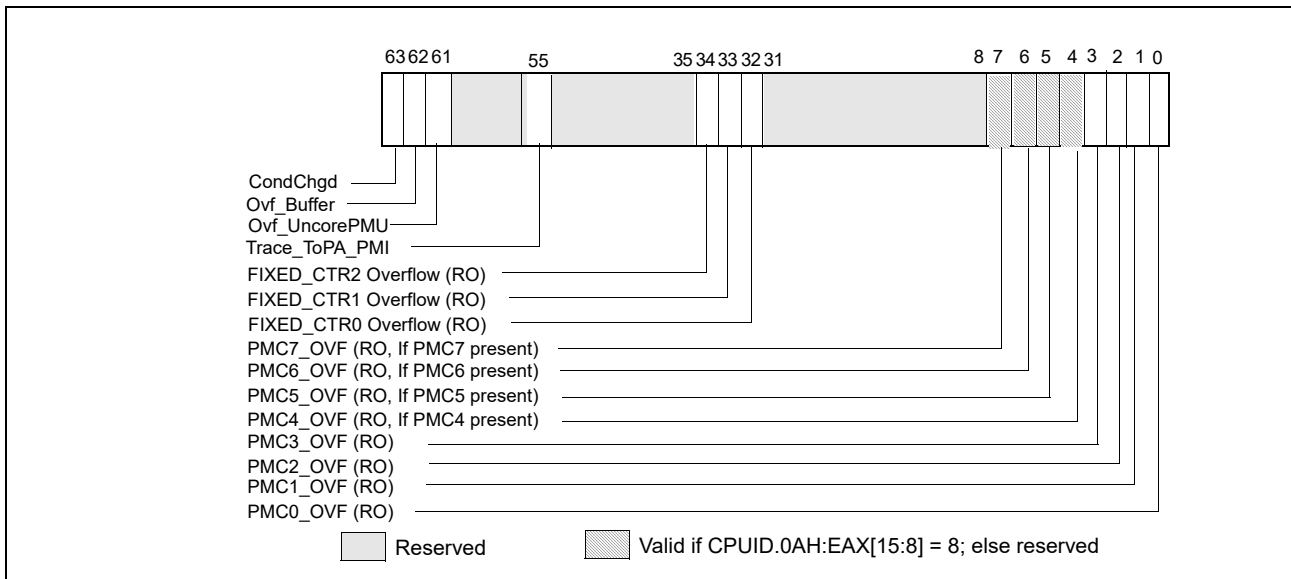
Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in “Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-33.

### 18.3.7 5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility

The 5th Generation Intel® Core™ processor and the Intel® Core™ M processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU has the same capability as those described in Section 18.3.6. IA32\_PERF\_GLOBAL\_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.



**Figure 18-35. IA32\_PERF\_GLOBAL\_STATUS MSR in Broadwell Microarchitecture**

Details of Intel Processor Trace is described in Chapter 31, “Intel® Processor Trace”. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR provide a corresponding reset control bit.

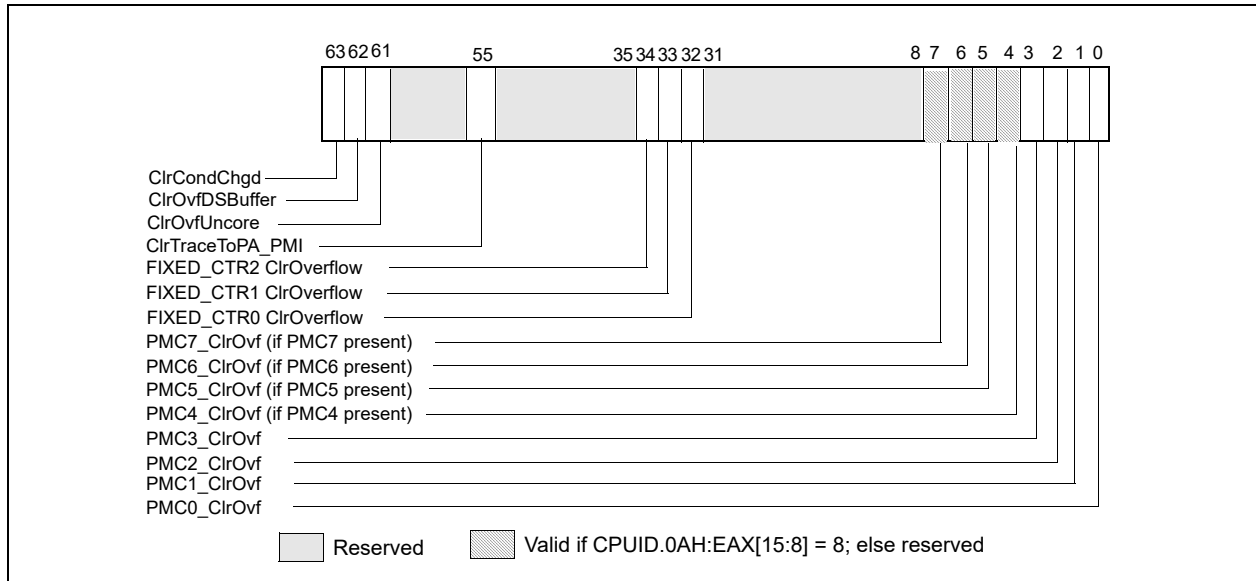


Figure 18-36. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events can be found at: <https://perfmon-events.intel.com/>.

### 18.3.8 6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The 8th generation Intel® Core™ processor is based on the Coffee Lake microarchitecture. For these microarchitectures, the core PMU supports architectural performance monitoring capability with version ID 4 (see Section 18.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The core PMU's capability is similar to those described in Section 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.3.6.5. For details of Intel TSX, see Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 38, "Enclave Code Debug and Profiling".

**Table 18-33. Core PMU Comparison**

Box	Intel® Microarchitecture Code Name Skylake, Kaby Lake and Coffee Lake	Intel® Microarchitecture Code Name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to determine # of counters. See Section 18.2.1.
Architectural Perfmon version	4	3	See Section 18.2.4
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>Freeze_Perfmon_on_PMI with streamlined semantics.</li> <li>Freeze_LBR_on_PMI with streamlined semantics.</li> <li>Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> <li>Freeze_while_SMM.</li> </ul>	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> <li>Query via IA32_PERF_GLOBAL_STATUS</li> <li>Reset via IA32_PERF_GLOBAL_STATUS_RESET</li> <li>Set via IA32_PERF_GLOBAL_STATUS_SET</li> </ul>	<ul style="list-style-type: none"> <li>Query via IA32_PERF_GLOBAL_STATUS</li> <li>Reset via IA32_PERF_GLOBAL_OVF_CTRL</li> </ul>	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	<ul style="list-style-type: none"> <li>Individual counter overflow</li> <li>PEBS buffer overflow</li> <li>ToPA buffer overflow</li> <li>CTR_Frz, LBR_Frz, ASCI</li> </ul>	<ul style="list-style-type: none"> <li>Individual counter overflow</li> <li>PEBS buffer overflow</li> <li>ToPA buffer overflow (applicable to Broadwell microarchitecture)</li> </ul>	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> <li>CTR_Frz</li> <li>LBR_Frz</li> </ul>	NA	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 18.2.4.3.
Precise Events	See Table 18-36.	See Table 18-12.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 18.3.8.1.4.	No	
LBR Record Format Encoding	000101b	000100b	Section 17.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 17.12
LBR Timing	Yes	No	Section 17.12.1
Call Stack Profiling	Yes, see Section 17.11	Yes, see Section 17.11	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for Perfmon	See Section 18.3.6.5.	See Section 18.3.6.5.	

### 18.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 18-34.

**Table 18-34. PEBS Facility Comparison**

Box	Intel® Microarchitecture Code Name Skylake, Kaby Lake and Coffee Lake	Intel® Microarchitecture Code Name Haswell and Broadwell	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-15	Figure 18-15	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 18-35; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 18-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 18-36.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
Data Address Profiling	Yes	Yes	
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

#### NOTES

Precise events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

#### 18.3.8.1.1 PEBS Data Format

The PEBS record format for the 6th generation, 7th generation and 8th generation Intel Core processors is reporting with encoding 0011b in IA32\_PERF\_CAPABILITIES[11:8]. The lay out is shown in Table 18-35. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

**Table 18-35. PEBS Record Format for 6th Generation, 7th Generation and 8th Generation Intel Core Processor Families**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 18-24.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and data address profiling (Section 18.3.6.3).

In the core PMU of the 6th generation, 7th generation and 8th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th generation and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32\_PERF\_GLOBAL\_STATUS may be useful to resolve the situations when more than one of IA32\_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

### 18.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake, Kaby Lake and Coffee Lake microarchitectures is shown in Table 18-36.

**Table 18-36. Precise Events for the Skylake, Kaby Lake and Coffee Lake Microarchitectures**

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST <sup>1</sup>	01H
		ALL_CYCLES <sup>2</sup>	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable <sup>3</sup> >	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED <sup>2</sup>	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED <sup>4</sup>	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H
MEM_LOAD_L3_HIT_RETIRED <sup>2</sup>	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

**NOTES:**

1. Only available on IA32\_PMC1.
2. INST\_RETIRED.ALL\_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR\_PEBBS\_FRONTEND, see Section 18.3.8.2
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

**18.3.8.1.3 Data Address Profiling**

The PEBS Data address profiling on the 6th generation, 7th generation and 8th generation Intel Core processors is largely unchanged from the prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.

**Table 18-37. Layout of Data Linear Address Information In PEBS Record**

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	AOH	<ul style="list-style-type: none"> <li>▪ <b>DCU Hit</b> (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES.</li> <li>▪ Other bits are zero.</li> </ul>
Reserved	A8H	Always zero.

### 18.3.8.1.4 PEBS Facility for Front End Events

In the 6th generation, 7th generation and 8th generation Intel Core processors, the PEBS facility has been extended to allow capturing PEBS data for some microarchitectural conditions related to front end events. The front-end microarchitectural conditions supported by PEBS requires the following interfaces:

- The IA32\_PERFEVTSELx MSR must select "FrontEnd\_Retired" (C6H) in the EventSelect field (bits 7:0) and umask = 01H,
- The "FRONTEND\_RETIRED" event employs a new MSR, MSR\_PEBS\_FRONTEND, to specify the supported frontend event details, see Table 18-38.
- Program the PEBS\_EN\_PMCx field of IA32\_PEBS\_ENABLE MSR as required.

Note the AnyThread field of IA32\_PERFEVTSELx is ignored by the processor for the "FRONTEND\_RETIRED" event.

The sub-event encodings supported by MSR\_PEBS\_FRONTEND.EVTSEL is given in Table 18-38.

**Table 18-38. FrontEnd\_Retired Sub-Event Encodings Supported by MSR\_PEBS\_FRONTEND.EVTSEL**

Sub-Event Name	EVTSEL	Description
DSB_MISS	11H	Retired Instructions which experienced decode stream buffer (DSB) miss.
L1L_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss <sup>1</sup> . Additional requests to the same cache line as an in-flight L1L cache miss will not be counted.
L2L_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	<p>An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles.</p> <p>The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.</p>

**NOTES:**

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).



The layout of MSR\_PEBS\_FRONTEND is given in Table 18-39.

**Table 18-39. MSR\_PEBS\_FRONTEND Layout**

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 18-38.
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.
Reserved	63:23	Reserved

### 18.3.8.1.5 FRONTEND\_RETIRED

The FRONTEND\_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) front-end events, i.e. events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a front-end (miss) event occurs outside instruction boundary (e.g. due to processor handling of architectural event), it may be reported for the next instruction to retire.

### 18.3.8.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Software must program MSR\_OFFCORE\_RSP\_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-40.
- Supplier information (bits 29:16): see Table 18-41.
- Snoop response information (bits 37:30): see Table 18-42.

**Table 18-40. MSR\_OFFCORE\_RSP\_x Request\_Type Definition (Skylake, Kaby Lake and Coffee Lake Microarchitectures)**

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	14:3	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

Table 18-41 lists the supplier information field that applies to 6th generation, 7th generation and 8th generation Intel Core processors. (6th generation Intel Core processor CPUID signatures: 06\_4EH, 06\_5EH; 7th generation and 8th generation Intel Core processor CPUID signatures: 06\_8EH, 06\_9EH).

**Table 18-41. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CPUID Signatures 06\_4EH, 06\_5EH and 06\_8EH, 06\_9EH)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT	22	L4 Cache (if L4 is present in the processor).
	Reserved	25:23	Reserved
	DRAM	26	Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).

Table 18-42 lists the snoop information field that apply to processors with CPUID signatures 06\_4EH, 06\_5EH, 06\_8EH, 06\_9E, and 06\_55H.

**Table 18-42. MSR\_OFFCORE\_RSP\_x Snoop Info Field Definition  
(CPUID Signatures 06\_4EH, 06\_5EH, 06\_8EH, 06\_9E and 06\_55H)**

Subtype	Bit Name	Offset	Description
Snoop Info	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).
	SNOOP_NONE	31	No details on snoop-related information.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores. -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO). -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD). -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S). In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD). -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO). -Snoop MtoS (LLC Hit, IFetch/Data_RD).
SNOOP_NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.	

### 18.3.8.2.1 Off-core Response Performance Monitoring for the Intel® Xeon® Processor Scalable Family

The following tables list the requestor and supplier information fields that apply to the Intel® Xeon® Processor Scalable Family.

- Transaction request type encoding (bits 15:0): see Table 18-43.
- Supplier information (bits 29:16): see Table 18-44.
- Supplier information (bits 29:16) with support for Intel® Optane™ DC Persistent Memory support: see Table 18-45.
- Snoop response information has not been changed and is the same as in (bits 37:30): see Table 18-42.

**Table 18-43. MSR\_OFFCORE\_RSP\_x Request\_Type Definition (Intel® Xeon® Processor Scalable Family)**

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DEMAND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DEMAND_CODE_RD	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	3	Reserved.
PF_L2_DATA_RD	4	Counts the number of prefetch data reads into L2.
PF_L2_RFO	5	Counts the number of RFO Requests generated by the MLC prefetches to L2.
Reserved	6	Reserved.
PF_L3_DATA_RD	7	Counts the number of MLC data read prefetches into L3.
PF_L3_RFO	8	Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	9	Reserved.
PF_L1D_AND_SW	10	Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.
Reserved	14:11	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

Table 18-44 lists the supplier information field that applies to the Intel Xeon Processor Scalable Family (CPUID signature: 06\_55H).

**Table 18-44. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CPUID Signature 06\_55H)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	Reserved	25:22	Reserved
	L3_MISS_LOCAL_DRAM	26	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
Reserved	30	Reserved	

Table 18-45 lists the supplier information field that applies to the Intel Xeon Processor Scalable Family (CPUID signature: 06\_55H, Steppings 0x5H - 0xFH).

**Table 18-45. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition  
(CUID Signature 06\_55H, Steppings 0x5H - 0xFH)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	LOCAL_PMM	22	Local home requests that were serviced by local PMM.
	REMOTE_HOP0_PMM	23	Hop 0 Remote supplier.
	REMOTE_HOP1_PMM	24	Hop 1 Remote supplier.
	REMOTE_HOP2P_PMM	25	Hop 2 or more Remote supplier.
	L3_MISS_LOCAL_DRAM	26	L3 Miss: Local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
Reserved		30	Reserved

### 18.3.8.3 Uncore Performance Monitoring Facilities on Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Cannon Lake microarchitecture introduces LLC support of up to six processor cores. To support six processor cores and eight LLC slices, existing MSR addresses have been rearranged and new CBo MSR addresses have been added. Uncore performance monitoring software drivers from prior generations of Intel Core processors will need to update the MSR addresses. The new MSR addresses and updated MSR addresses have been added to the Uncore PMU listing in Section 2.17.2, “MSRs Specific to 8th Generation Intel® Core™ i3 Processors” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

### 18.3.9 10th Generation Intel® Core™ Processor Performance Monitoring Facility

The 10th generation Intel® Core™ processor is based on Ice Lake microarchitecture. For this microarchitecture, the core PMU supports architectural performance monitoring capability with version Id 5 (see Section 18.2.5) and a host of non-architectural monitoring capabilities.

The core PMU's capability is similar to those described in Section 18.3.1 through Section 18.3.8, with some differences and enhancements summarized in Table 18-46.

**Table 18-46. PEBS Facility Comparison**

Box	Ice Lake Microarchitecture	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Comment
Architectural Perfmon version	5	4	See Section 18.2.5.
PEBS: Basic functionality	Yes	Yes	See Section 18.3.9.1.
PEBS record format encoding	0100b	0011b	See Section 18.6.2.4.2.

**Table 18-46. PEBS Facility Comparison**

Box	Ice Lake Microarchitecture	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Comment
Extended PEBS	PEBS is extended to all Fixed and General Purpose counters and to all performance monitoring events.	No	See Section 18.9.1.
Adaptive PEBS	Yes	No	See Section 18.9.2.
Performance Metrics	Yes (4)	No	See Section 18.3.9.3.
PEBS-PDIR	IA32_FIXED0 only (Corresponding counter control MSRs must be enabled.)	IA32_PMC1 only.	

### 18.3.9.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 10th generation Intel Core processors provides a number of enhancements relative to PEBS in processors based on the Skylake, Kaby Lake, and Coffee Lake microarchitectures. Enhancement of PEBS facility with Extended PEBS and Adaptive PEBS features are described in detail in Section 18.9.

### 18.3.9.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Software must program MSR\_OFFCORE\_RSP\_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-[N1].
- Response type encoding (bits 16-37) of
  - Supplier information: see Table [18-N2].
  - Snoop response information: see Table [18-N3].
- All transactions are tracked at cacheline granularity except some in request type OTHER.

**Table 18-47. MSR\_OFFCORE\_RSP\_x Request\_Type Definition  
(Future Processors Based on Ice Lake Microarchitecture)**

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data and page table entry reads.
DEMAND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DEMAND_CODE_RD	2	Counts demand instruction fetches and instruction prefetches targeting the L1 instruction cache.
Reserved	3	Reserved
HWPf_L2_DATA_RD	4	Counts hardware generated data read prefetches targeting the L2 cache.
HWPf_L2_RFO	5	Counts hardware generated prefetches for exclusive ownership (RFO) targeting the L2 cache.
Reserved	6	Reserved
HWPf_L3	9:7 and 13 <sup>1</sup>	Counts hardware generated prefetches of any type targeting the L3 cache.
HWPf_L1D_AND_SWPF	10	Counts hardware generated data read prefetches targeting the L1 data cache and the following software prefetches (PREFETCHNTA, PREFETCHT0/1/2).
STREAMING_WR	11	Counts streaming stores.
Reserved	12	Reserved
Reserved	14	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

**NOTES:**

1. All bits need to be set to 1 to count this type.

Ice Lake microarchitecture has added a new category of Response subtype, called a Combined Response Info. To count a feature in this type, all the bits specified must be set to 1.

A valid response type must be a non-zero value of the following expression:

Any | ['OR' of Combined Response Info Bits | (('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits))]

If "ANY" bit[16] is set, other response type bits [17-39] are ignored.

Table 18-48 lists the supplier information field that applies to processors based on Ice Lake microarchitecture.

**Table 18-48. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition  
(Processors Based on Ice Lake Microarchitecture)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Combined Response Info	DRAM	26, 31, 32 <sup>1</sup>	Requests that are satisfied by DRAM.
	NON_DRAM	26, 37 <sup>1</sup>	Requests that are satisfied by a NON_DRAM system component. This includes MMIO transactions.
	L3_MISS	22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37 <sup>1</sup>	Requests that were not supplied by the L3 Cache. The event includes some currently reserved bits in anticipation of future memory designs.
Supplier Info	L3_HIT	18,19, 20 <sup>1</sup>	Requests that hit in L3 cache. Depending on the snoop response the L3 cache may have retrieved the cacheline from another core's cache.
Reserved		17, 21:25, 27:29	Reserved.

**NOTES:**

1. All bits need to be set to 1 to count this type.

Table 18-49 lists the snoop information field that applies to processors based on Ice Lake microarchitecture.

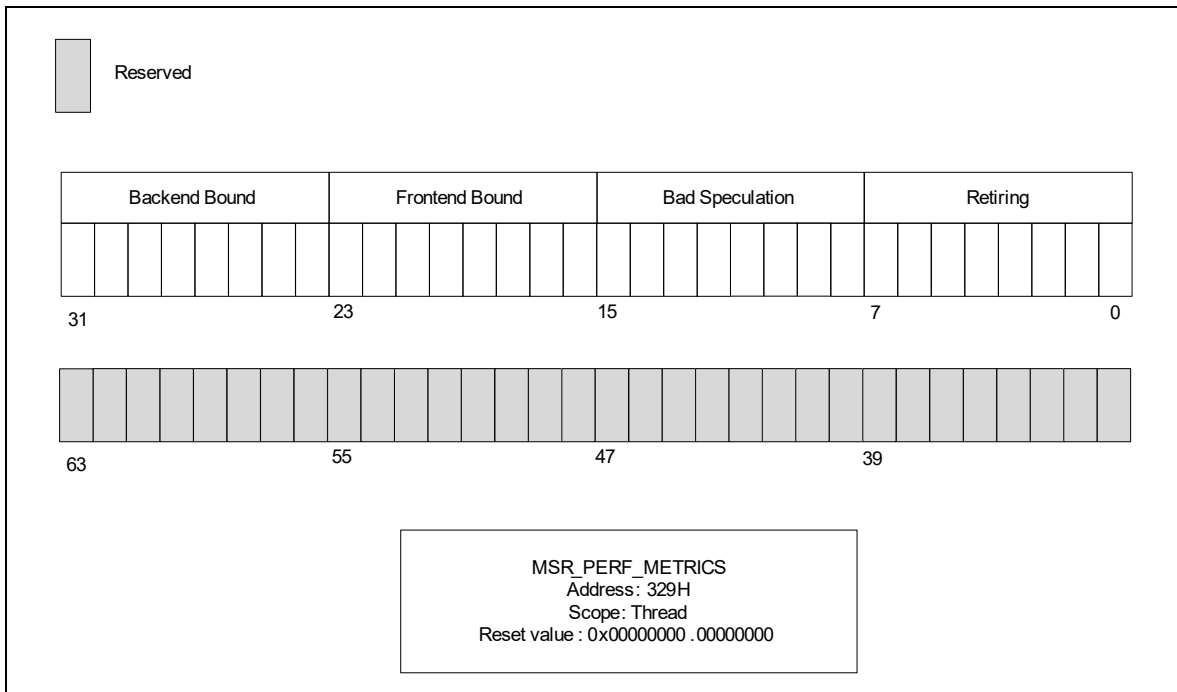
**Table 18-49. MSR\_OFFCORE\_RSP\_x Snoop Info Field Definition (Processors Based on Ice Lake Microarchitecture)**

Subtype	Bit Name	Offset	Description
Snoop Info	Reserved	30	Reserved.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was sent and none of the snooped caches contained the cacheline.
	SNOOP_HIT_NO_FWD	34	A snoop was sent and hit in at least one snooped cache. The unmodified cacheline was not forwarded back, because the L3 already has a valid copy.
	Reserved	35	Reserved.
	SNOOP_HITM	36	A snoop was sent and the cacheline was found modified in another core's caches. The modified cacheline was forwarded to the requesting core.

### 18.3.9.3 Performance Metrics

The Ice Lake core PMU provides built-in support for Top-down Microarchitecture Analysis (TMA) method level 1 metrics. These metrics are always available to cross-validate performance observations, freeing general purpose counters to count other events in high counter utilization scenarios. For more details about the method, refer to Top-Down Analysis Method chapter (Appendix B.1) of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

A new MSR called MSR\_PERF\_METRICS reports the metrics directly. Software can check (and/or expose to its guests) the availability of the PERF\_METRICS feature using IA32\_PERF\_CAPABILITIES.PERF\_METRICS\_AVAILABLE (bit 15).



**Figure 18-37. MSR\_PERF\_METRICS Definition**

This register exposes the four TMA Level 1 metrics. The lower 32 bits are divided into four 8-bit fields, as shown by the above figure, each of which is an integer fraction of 255.



To support built-in performance metrics, new bits have been added to the following MSRs:

- IA32\_PERF\_GLOBAL\_CTRL. EN\_PERF\_METRICS[48]: If this bit is set and fixed counter 3 is enabled, built-in performance metrics are enabled.
- IA32\_PERF\_GLOBAL\_STATUS\_SET. SET\_OVF\_PERF\_METRICS[48]: If this bit is set, it will set the status bit in the IA32\_PERF\_GLOBAL\_STATUS register for PERF\_METRICS.
- IA32\_PERF\_GLOBAL\_STATUS\_RESET. RESET\_OVF\_PERF\_METRICS[48]: If this bit is set, it will clear the status bit in the IA32\_PERF\_GLOBAL\_STATUS register for PERF\_METRICS.
- IA32\_PERF\_GLOBAL\_STATUS. OVF\_PERF\_METRICS[48]: If this bit is set, it indicates that a PERF\_METRICS-related resource has overflowed and a PMI is triggered<sup>4</sup>. If this bit is clear, no such overflow has occurred.

### NOTE

Software has to synchronize, e.g., re-start, fixed counter 3 as well as PERF\_METRICS when either bit 35 or 48 in IA32\_PERF\_GLOBAL\_STATUS is set. Otherwise, PERF\_METRICS may return undefined values.

The values in MSR\_PERF\_METRICS are derived from fixed counter 3. Software should start both registers, PERF\_METRICS and fixed counter 3, from zero. Additionally, software is recommended to periodically clear both registers in order to maintain accurate measurements for certain scenarios that involve sampling metrics at high rates.

In order to save/restore PERF\_METRICS, software should follow these guidelines:

- PERF\_METRICS and fixed counter 3 should be saved and restored together.
- To ensure that PERF\_METRICS and fixed counter 3 remain synchronized, both should be disabled during both save and restore. Software should enable/disable them atomically, with a single write to IA32\_PERF\_GLOBAL\_CTRL to set/clear both EN\_PERF\_METRICS[bit 48] and EN\_FIXED\_CTR3[bit 35].
- On state restore, fixed counter 3 must be restored **before** PERF\_METRICS, otherwise undefined results may be observed.

## 18.3.10 3rd Generation Intel® Xeon® Scalable Family Performance Monitoring Facility

The 3rd generation Intel® Xeon® Scalable Family of processors are based on Ice Lake microarchitecture. For this microarchitecture, the core PMU supports architectural performance monitoring capability with version Id 5 (see Section 18.2.5) and a host of non-architectural monitoring capabilities.

The core PMU's capability is similar to those described in Section 18.3.1 through Section 18.3.8, with some differences and enhancements summarized in Table 18-46.

### 18.3.10.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 3rd generation Intel Xeon Scalable Family of processors provide a number of enhancements relative to PEBS in previous generations of processors. Enhancement of PEBS facility with Extended PEBS and Adaptive PEBS features are described in detail in Section 18.9.

Unlike earlier processors, the 3rd generation Intel Xeon Scalable Family of processors (and later generations) can support the recording of PEBS records even if accesses to the linear addresses in the DS buffer management area or in the PEBS buffer cause VM exits. When a VM exit occurs, the PEBS record is not lost but instead will "skid" and be recorded after the virtual machine is resumed (assuming that the cause of the VM exit is resolved). For precise events the guest will observe that the record skid by one event occurrence, while for non-precise events the record will skid by one instruction. With this "EPT-friendly" enhancement, a virtual-machine monitor may allow use of PEBS by guest software even if EPT does not map all guest-physical memory as present and read/write. It remains

---

4. An overflow of fixed counter 3 should normally happen first if software follows Intel's recommendations.

the case that an operating system should establish its paging structures to prevent page faults in those paging structures.

The 3rd generation Intel Xeon Scalable Family of processors based on the Ice Lake microarchitecture introduce EPT-friendly PEBS. This allows EPT violations and other VM Exits to be taken on PEBS accesses to the DS Area. See Section 18.9.5 for details.

## 18.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

### NOTE

This section also applies to the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture.

### 18.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.

A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel® Atom™ processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32\_PMC0, IA32\_PMC1) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32\_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. The processor supports AnyThread counting in three architectural performance monitoring events.

#### 18.4.1.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ processor Tile

The Intel® Xeon Phi™ processor tile includes the following enhancements to the Silvermont microarchitecture.

- AnyThread support. This facility is limited to following three architectural events: Instructions Retired, Unhalted Core Cycles, Unhalted Reference Cycles using IA32\_FIXED\_CTR0-2 and Unhalted Core Cycles, Unhalted Reference Cycles using IA32\_PERFEVTSELx.
- PEBS-DLA (Processor Event-Based Sampling-Data Linear Address) fields. The processor provides memory address in addition to the Silvermont PEBS record support on select events. The PEBS recording format as reported by IA32\_PERF\_CAPABILITIES [11:8] is 2.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor tile to subsystems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32\_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32\_PERFEVTSELx. Two cores do not share the off-core response MSRs. Knights Landing expands off-core response capability to match the processor untile changes.

- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests. This facility is updated to match the processor until changes.

#### 18.4.1.1.1 Processor Event-Based Sampling

The processor supports processor event based sampling (PEBS). PEBS is supported using IA32\_PMC0 (see also Section 17.4.9, "BTS and DS Save Area").

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.6.2.4).

The list of PEBS events supported in the processor is shown in the following table.

**Table 18-50. PEBS Performance Events for the Knights Landing Microarchitecture**

Event Name	Event Select	Sub-event	UMask	Data Linear Address Support
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		CALL	F9H	No
		REL_CALL	FDH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		FAR_BRANCH	BFH	No
		RETURN	F7H	No
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		RETURN	F7H	No
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H	Yes
		L2_MISS_LOADS	04H	Yes
		DLTB_MISS_LOADS	08H	Yes
RECYCLEQ	03H	LD_BLOCK_ST_FORWARD	01H	Yes
		LD_SPLITS	08H	Yes

The PEBS record format 2 supported by processors based on the Knights Landing microarchitecture is shown in Table 18-51, and each field in the PEBS record is 64 bits long.

**Table 18-51. PEBS Record Format for the Knights Landing Microarchitecture**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14

**Table 18-51. PEBS Record Format for the Knights Landing Microarchitecture (Contd.)**

Byte Offset	Field	Byte Offset	Field
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	PSDLA
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

#### 18.4.1.1.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR\_OFFCORE\_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR\_OFFCORE\_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-52 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

**Table 18-52. OffCore Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMCO-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

Some of the MSR\_OFFCORE\_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 registers are defined in Table 18-53. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR\_OFFCORE\_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.5.2.3 for details.

Table 18-53. Bit fields of the MSR\_OFFCORE\_RESP [0, 1] Registers

Main	Sub-field	Bit	Name	Description
Request Type		0	DEMAND_DATA_RD	Demand cacheable data and L1 prefetch data reads.
		1	DEMAND_RFO	Demand cacheable data writes.
		2	DEMAND_CODE_RD	Demand code reads and prefetch code reads.
		3	Reserved	Reserved.
		4	Reserved	Reserved.
		5	PF_L2_RFO	L2 data RFO prefetches (includes PREFETCHW instruction).
		6	PF_L2_CODE_RD	L2 code HW prefetches.
		7	PARTIAL_READS	Partial reads (UC or WC).
		8	PARTIAL_WRITES	Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		9	UC_CODE_READS	UC code reads.
		10	BUS_LOCKS	Bus locks and split lock requests.
		11	FULL_STREAMING_STORES	Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		12	SW_PREFETCH	Software prefetches.
		13	PF_L1_DATA_RD	L1 data HW prefetches.
		14	PARTIAL_STREAMING_STORES	Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.
		19	L2_HIT_OTHER_TILE_NEAR	Other tile L2 hit E Near.
		20	Reserved	Reserved.
		21	MCDRAM_NEAR	MCDRAM Local.
		22	MCDRAM_FAR_OR_L2_HIT_OTHER_TILE_FAR	MCDRAM Far or Other tile L2 hit far.
		23	DRAM_NEAR	DRAM Local.
		24	DRAM_FAR	DRAM Far.
	Data Supply from within same tile	25	L2_HITM_THIS_TILE	M-state.
		26	L2_HITE_THIS_TILE	E-state.
		27	L2_HITS_THIS_TILE	S-state.
		28	L2_HITF_THIS_TILE	F-state.
		29	Reserved	Reserved.
		30	Reserved	Reserved.

**Table 18-53. Bit fields of the MSR\_OFFCORE\_RESP [0, 1] Registers (Contd.)**

Main	Sub-field	Bit	Name	Description
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMCO. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMCO. This bit is reserved for OFFCORE_RESP_1 event.

### 18.4.1.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR\_OFFCORE\_RSP0.[bit 38] with the choice of request type specified in MSR\_OFFCORE\_RSP0.[bit 15:0].

Refer to Section 18.5.2.3, "Average Offcore Request Latency Measurement," for typical usage. Note that MSR\_OFFCORE\_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.

## 18.5 PERFORMANCE MONITORING (INTEL ATOM® PROCESSORS)

### 18.5.1 Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 18.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32\_PMC0, IA32\_PMC1) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2).

#### NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 18.2.1.

Non-architectural performance monitoring in Intel Atom processor family uses the IA32\_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events [can be found at: https://perfmon-events.intel.com/](https://perfmon-events.intel.com/).

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32\_PERFEVTSELx MSR, i.e., if IA32\_PERFEVTSELx.AnyThread = 1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

Valid event mask (Umask) bits can be found at: <https://perfmon-events.intel.com/>. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 18-71, Table 18-72, Table 18-73, and Table 18-74. One or more of these sub-fields may apply to specific events on an event-by-event basis. Precise Event Based Monitoring is supported using IA32\_PMC0 (see also Section 17.4.9, "BTS and DS Save Area").

## 18.5.2 Performance Monitoring for Silvermont Microarchitecture

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32\_PMC0, IA32\_PMC1) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32\_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: <https://perfmon-events.intel.com/>.

The bit fields (except bit 21) within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32\_PERFEVTSELx MSR.

### 18.5.2.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32\_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32\_PERFEVTSELx.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

#### 18.5.2.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32\_PMC0 (see also Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.6.2.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 18-54.

**Table 18-54. PEBS Performance Events for the Silvermont Microarchitecture**

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H
		L2_MISS_LOADS	04H
		DLTB_MISS_LOADS	08H
		HITM	20H
REHABQ	03H	LD_BLOCK_ST_FORWARD	01H
		LD_SPLITS	08H

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 18-55, and each field in the PEBS record is 64 bits long.

**Table 18-55. PEBS Record Format for the Silvermont Microarchitecture**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Reserved
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved



### 18.5.2.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR\_OFFCORE\_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR\_OFFCORE\_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-56 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

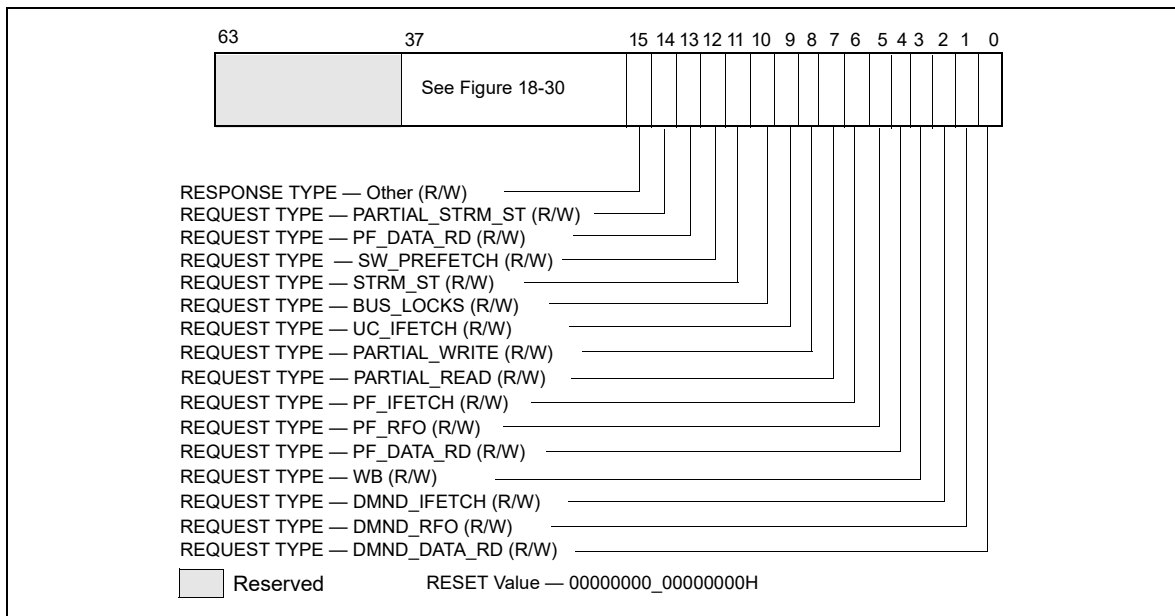
In the Silvermont microarchitecture, each MSR\_OFFCORE\_RSPx is shared by two processor cores.

**Table 18-56. OffCore Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMCO-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 are shown in Figure 18-38 and Figure 18-39. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR\_OFFCORE\_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.5.2.3 for details.



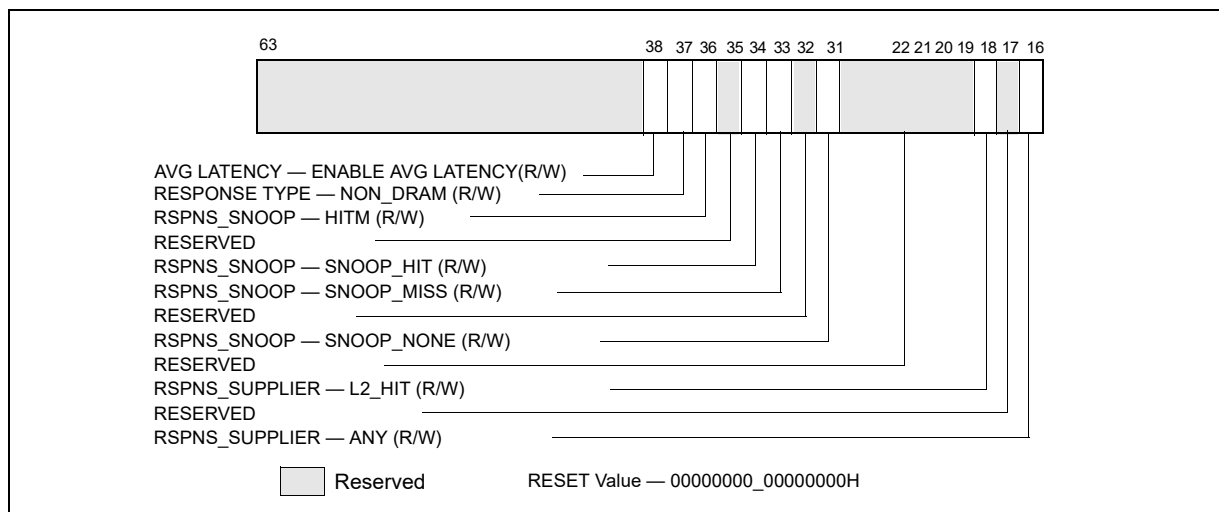
**Figure 18-38. Request\_Type Fields for MSR\_OFFCORE\_RSPx**

**Table 18-57. MSR\_OFFCORE\_RSPx Request\_Type Field Definition**

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.

**Table 18-57. MSR\_OFFCORE\_RSPx Request\_Type Field Definition (Contd.)**

Bit Name	Offset	Description
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT and WP)
UC_IFETCH	9	Counts the number of UC instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests
STRM_ST	11	Streaming store requests
SW_PREFETCH	12	Counts software prefetch requests
PF_DATA_RD	13	Counts DCU hardware prefetcher data read requests
PARTIAL_STRM_ST	14	Streaming store requests
ANY	15	Any request that crosses IDI, including I/O.



**Figure 18-39. Response\_Supplier and Snoop Info Fields for MSR\_OFFCORE\_RSPx**

To properly program this extra register, software must set at least one request type bit (Table 18-57) and a valid response type pattern (Table 18-58, Table 18-59). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR\_OFFCORE\_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

**Table 18-58. MSR\_OFFCORE\_RSP\_x Response Supplier Info Field Definition**

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY” bit is set, the supplier and snoop info bits are ignored.

**Table 18-59. MSR\_OFFCORE\_RSPx Snoop Info Field Definition**

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved
	HITM	36	Counts the number of snoops hit in the other module where modified copies were found in other core’s L1 cache.
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0).  This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

### 18.5.2.3 Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR\_OFFCORE\_RSP registers. Using two performance monitoring counters, program the two OFFCORE\_RESPONSE event encodings into the corresponding IA32\_PERFEVTSELx MSRs. Count the weighted cycles via MSR\_OFFCORE\_RSP0 by programming a request type in MSR\_OFFCORE\_RSP0.[15:0] and setting MSR\_OFFCORE\_RSP0.OUTSTANDING[38] to 1, while setting the remaining bits to 0. Count the number of requests via MSR\_OFFCORE\_RSP1 by programming the same request type from MSR\_OFFCORE\_RSP0 into MSR\_OFFCORE\_RSP1[bit 15:0], and setting MSR\_OFFCORE\_RSP1.ANY\_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32\_PMCx register that counted weight cycles by the register that counted requests.

## 18.5.3 Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture report architectural performance monitoring versionID = 4 (see Section 18.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The bit fields (except bit 21) within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. The Goldmont microarchitecture does not support Hyper-Threading and thus architectural and non-architectural performance monitoring events ignore the AnyThread qualification regardless of its setting in the IA32\_PERFEVTSELx MSR. However, Goldmont does not set the AnyThread deprecation bit (CPUID.0AH:EDX[15]).

The core PMU’s capability is similar to that of the Silvermont microarchitecture described in Section 18.5.2 , with some differences and enhancements summarized in Table 18-60.

**Table 18-60. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures**

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	4	2	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 18.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to determine # of counters. See Section 18.2.1.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with streamlined semantics.</li> <li>▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> </ul>	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET</li> <li>▪ Set via IA32_PERF_GLOBAL_STATUS_SET</li> </ul>	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL</li> </ul>	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> <li>▪ ToPA buffer overflow</li> <li>▪ CTR_Frz, LBR_Frz</li> </ul>	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> </ul>	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> <li>▪ CTR_Frz,</li> <li>▪ LBR_Frz</li> </ul>	No	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 18.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-61.	See Section 18.5.2.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 18-54).	IA32_PMC0 only.
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 18-62; enhanced fields at offsets 90H- 98H; and TSC record field at COH.	Table 18-55.	
PEBS EventingIP	Yes	Yes	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register.	MSR 1A6H and 1A7H, shared by a pair of cores.	Nehalem supports 1A6H only.

### 18.5.3.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32\_PMC0 for all events (see Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way on Goldmont microarchitecture as on the Silvermont microarchitecture. The record will be generated after an instruction that causes the event when the counter is already overflowed and will capture the architectural state at this point (see Section 18.6.2.4 and Section 17.4.9). The eventingIP in the record will indicate the instruction that caused the event. The list of precise events supported in the Goldmont microarchitecture is shown in Table 18-61.

In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor's current retirement point. It is likely that the instruction fetch that caused the event to increment was beyond that current retirement point. Other examples of non-precise events are CPU\_CLK\_UNHALTED.CORE\_P and HARDWARE\_INTERRUPTS.RECEIVED. CPU\_CLK\_UNHALTED.CORE\_P will increment each cycle that the processor is awake. When the counter overflows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE\_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32\_PERF\_GLOBAL\_STATUS\_SET.

**Table 18-61. Precise Events Supported by the Goldmont Microarchitecture**

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	COH	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH

**Table 18-61. Precise Events Supported by the Goldmont Microarchitecture (Contd.)**

Event Name	Event Select	Sub-event	UMask
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Intel Goldmont microarchitecture is shown in Table 18-62, and each field in the PEBS record is 64 bits long.

**Table 18-62. PEBS Record Format for the Goldmont Microarchitecture**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counters
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Reserved
40H	R/EBP	A8H	Reserved
48H	R/ESP	B0H	EventingRIP
50H	R8	B8H	Reserved
58H	R9	C0H	TSC
60H	R10		

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the "Applicable Counter" field, which indicates which counters actually requested generating a PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event even when multiple counters are configured to record PEBS records and multiple bits are set in the field. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

#### 18.5.3.1.1 PEBS Data Linear Address Profiling

Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling; those fields are present in the record but are reserved.

For Goldmont microarchitecture, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g. the one that triggered a precise event that caused the PEBS record to be generated). Goldmont microarchitecture may record a Data Linear Address for the instruction that caused the event even for events not related to memory accesses. This may differ from other microarchitectures.

#### 18.5.3.1.2 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the "skid" problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 18.3.4.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST\_RETIRE, except for UOPS\_RETIRE. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

With Reduced Skid PEBS, the skid is precisely one event occurrence. Hence if counting INST\_RETIRE, PEBS will indicate the instruction that follows that which caused the counter to overflow.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g. via IA32\_PERF\_GLOBAL\_CTRL MSR) before changes to the configuration (e.g. what event is specified in IA32\_PERFEVTSELx or whether PEBS is enabled for that counter via IA32\_PEBS\_ENABLE) or counter value (MSR write to IA32\_PMCx and IA32\_A\_PMCx).

#### 18.5.3.1.3 Enhancements to IA32\_PERF\_GLOBAL\_STATUS.OvfDSBuffer[62]

In addition to IA32\_PERF\_GLOBAL\_STATUS.OvfDSBuffer[62] being set when PEBS\_Index reaches the PEBS\_Interrupt\_Threshold, the bit is also set when PEBS\_Index is out of bounds. That is, the bit will be set when PEBS\_Index < PEBS\_Buffer\_Base or PEBS\_Index > PEBS\_Absolute\_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32\_PERF\_GLOBAL\_STATUS will be cleared according to Applicable Counters, however the IA32\_PMCx values will not be reloaded with the Reset values stored in the DS\_AREA.

#### 18.5.3.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR\_OFFCORE\_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR\_OFFCORE\_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-56 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR\_OFFCORE\_RSPx registers per core.

The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 are organized as follows:

- Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 18-63.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 18-58.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 18-64.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 18.5.2.3 for details.

**Table 18-63. MSR\_OFFCORE\_RSPx Request\_Type Field Definition**

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	Counts demand instruction cacheline and l-side prefetch requests that miss the instruction cache.
COREWB	3	Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.
PARTIAL_READS	7	Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	Counts partial cacheline writes due to streaming stores.
ANY_REQUEST	15	Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 18-57) and a valid response type pattern (either Table 18-58 or Table 18-64). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR\_OFFCORE\_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.



**Table 18-64. MSR\_OFFCORE\_RSPx For L2 Miss and Outstanding Requests**

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests <sup>1</sup>	OUTSTANDING	38	Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

**NOTES:**

1. See Section 18.5.2.3, "Average Offcore Request Latency Measurement" for details on how to use this bit to extract average latency.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

Any\_Response Bit | L2 Hit | 'OR' of Snoop Info Bits | Outstanding Bit

### 18.5.3.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 18.5.2.3.

## 18.5.4 Performance Monitoring for Goldmont Plus Microarchitecture

Intel Atom processors based on the Goldmont Plus microarchitecture report architectural performance monitoring versionID = 4 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

Goldmont Plus performance monitoring capabilities are similar to Goldmont capabilities. The differences are in specific events and in which counters support PEBS. Goldmont Plus introduces the ability for fixed performance monitoring counters to generate PEBS records.

Goldmont Plus will set the AnyThread deprecation CPUID bit (CPUID.0AH:EDX[15]) to indicate that the Any-Thread bits in IA32\_PERFVTSELx and IA32\_FIXED\_CTR\_CTRL have no effect.

The core PMU's capability is similar to that of the Goldmont microarchitecture described in Section 18.6.3, with some differences and enhancements summarized in Table 18-65.

**Table 18-65. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures**

Box	Goldmont Plus Microarchitecture	Goldmont Microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change.
Architectural Performance Monitoring version ID	4	4	No change.
Processor Event Based Sampling (PEBS) Events	All General-Purpose and Fixed counters. Each General-Purpose counter supports all events (precise and non-precise).	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-61.	Goldmont Plus supports PEBS on all counters.
PEBS record format encoding	0011b	0011b	No change.

### 18.5.4.1 Extended PEBS

The PEBS facility in Goldmont Plus microarchitecture provides a number of enhancements relative to PEBS in processors from previous generations. Enhancement of PEBS facility with the Extended PEBS feature are described in detail in section 18.9.

## 18.5.5 Performance Monitoring for Tremont Microarchitecture

Intel Atom processors based on the Tremont microarchitecture report architectural performance monitoring versionID = 5 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 5 capabilities are described in Section 18.2.5.

Tremont performance monitoring capabilities are similar to Goldmont Plus capabilities, with the following extensions:

- Support for Adaptive PEBS.
- Support for PEBS output to Intel® Processor Trace.
- Precise Distribution support on Fixed Counter0.
- Compatibility enhancements to off-core response MSRs, MSR\_OFFCORE\_RSPx.

The differences and enhancements between Tremont microarchitecture and Goldmont Plus microarchitecture are summarized in Table 18-66.

**Table 18-66. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures**

Box	Tremont Microarchitecture	Goldmont Plus Microarchitecture	Comment
# of fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change. See Section 18.2.2.
Architectural Performance Monitoring version ID	5	4	
PEBS record format encoding	0100b	0011b	See Section 18.6.2.4.2.
Reduce skid PEBS	IA32_PMC0 and IA32_FIXED_CTR0	IA32_PMC0 only	
Extended PEBS	Yes	Yes	See Section 18.5.4.1.
Adaptive PEBS	Yes	No	See Section 18.9.2.
PEBS output	DS Save Area or Intel® Processor Trace	DS Save Area only	See Section 18.5.5.2.1.
PEBS record layout	See Section 18.9.2.3 for output to DS, Section 18.5.5.2.2 for output to Intel PT.	Table 18-62; enhanced fields at offsets 90H- 98H; and TSC record field at COH.	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register, extended request and response types.	MSR 1A6H and 1A7H, each core has its own register.	

### 18.5.5.1 Adaptive PEBS

The PEBS record format and configuration interface has changed versus Goldmont Plus, as the Tremont microarchitecture includes support for the configurable Adaptive PEBS records; see Section 18.9.2.

### 18.5.5.2 PEBS output to Intel® Processor Trace

Intel Atom processors based on the Tremont microarchitecture introduce the following Precise Event-Based Sampling (PEBS) extensions:

- A mechanism to direct PEBS output into the Intel® Processor Trace (Intel® PT) output stream. In this scenario, the PEBS record is written in packetized form, in order to co-exist with other Intel PT trace data.
- New Performance Monitoring counter reload MSRs, which are used by PEBS in place of the counter reload values stored in the DS Management area when PEBS output is directed into the Intel PT output stream.

Processors that indicate support for Intel PT by setting CPUID.07H.0.EBX[25]=1, and set the new IA32\_PERF\_CAPABILITIES.PEBS\_OUTPUT\_PT\_AVAIL[16] bit, support these extensions.

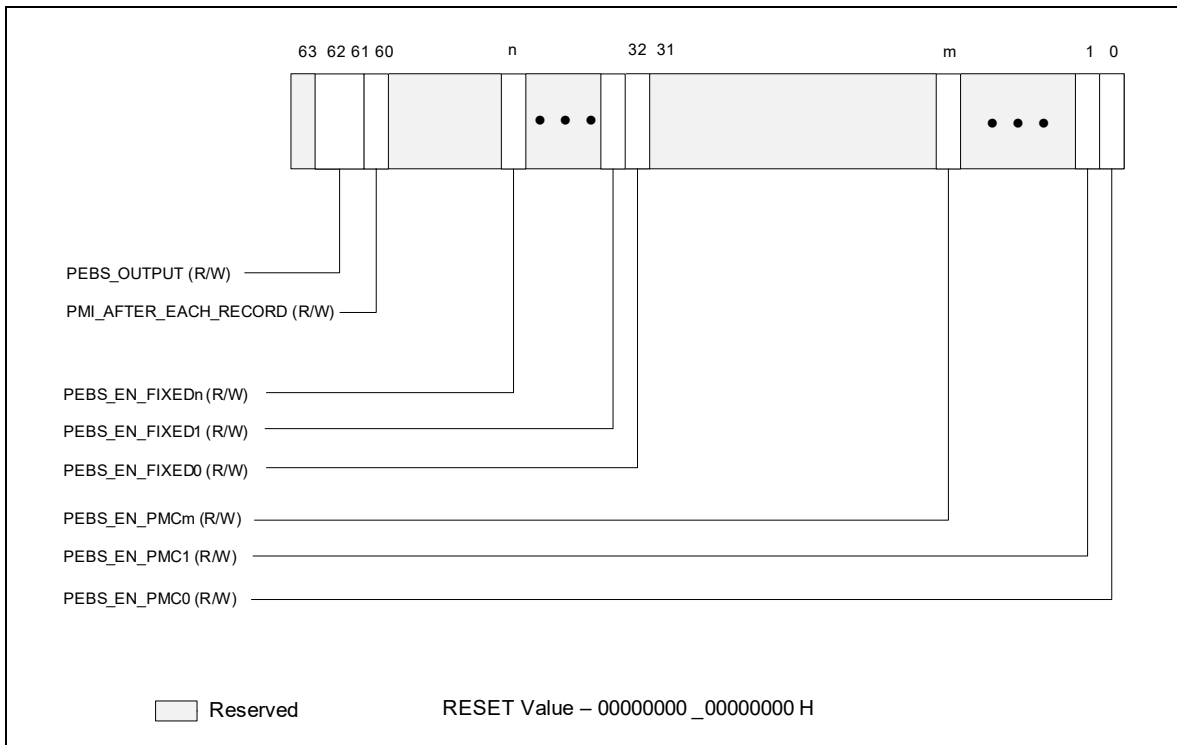
#### 18.5.5.2.1 PEBS Configuration

PEBS output to Intel Processor Trace includes support for two new fields in IA32\_PEBS\_ENABLE.

**Table 18-67. New Fields in IA32\_PEBS\_ENABLE**

Field	Description
PMI_AFTER_EACH_RECORD[60]	Pend a PerfMon Interrupt (PMI) after each PEBS event.
PEBS_OUTPUT[62:61]	Specifies PEBS output destination. Encodings: 00B: DS Save Area. Matches legacy PEBS behavior, output location defined by IA32_DS_AREA. 01B: Intel PT trace output. 10B: Reserved. 11B: Reserved.

When PEBS\_OUTPUT is set to 01B, the DS Management Area is not used and need not be configured. Instead, the output mechanism is configured through IA32\_RTIT\_CTL and other Intel PT MSRs, while counter reload values are configured in the MSR\_RELOAD\_PMCx MSRs. Details on configuring Intel PT can be found in Section 31.2.6.



**Figure 18-40. IA32\_PEBS\_ENABLE MSR with PEBS Output to Intel® Processor Trace**

**18.5.5.2.2 PEBS Record Format in Intel® Processor Trace**

The format of the PEBS record changes when output to Intel PT, as the PEBS state is packetized. Each PEBS grouping is emitted as a Block Begin (BBP) and following Block Item (BIP) packets. A PEBS grouping ends when either a new PEBS grouping begins (indicated by a BBP packet) or a Block End (BEP) packet is encountered. See Section 31.4.1.1 for details of these Intel PT packets.

Because the packet headers describe the state held in the packet payload, PEBS state ordering is not fixed. PEBS state groupings may be emitted in any order, and the PEBS state elements within those groupings may be emitted in any order. Further, there is no packet that provides indication of “Record Format” or “Record Size”.

If Intel PT tracing is not enabled (IA32\_RTIT\_STATUS.TriggerEn=0), any PEBS records triggered will be dropped. PEBS packets do not depend on ContextEn or FilterEn in IA32\_RTIT\_STATUS, any filtering of PEBS must be enabled from within the PerfMon configuration. Counter reload will occur in all scenarios where PEBS is triggered, regardless of TriggerEn.

The PEBS threshold mechanism for generating PerfMon Interrupts (PMIs) is not available in this mode. However, there exist other means to generate PMIs based on PEBS output. When the Intel PT ToPA output mechanism is chosen, a PMI can optionally be pended when a ToPA region is filled; see Section 31.2.6.2 for details. Further, software can opt to generate a PMI on each PEBS record by setting the new IA32\_PEBS\_ENABLE.PMI\_AFTER\_EACH\_RECORD[60] bit.

The IA32\_PERF\_GLOBAL\_STATUS.OvfDSBuffer bit will not be set in this mode.

### 18.5.5.2.3 PEBS Counter Reload

When PEBS output is directed into Intel PT (IA32\_PEBS\_ENABLE.PEBS\_OUTPUT = 01B), new MSR\_RELOAD\_PMCx MSRs are used by the PEBS routine to reload PerfMon counters. The value from the associated reload MSR will be loaded to the appropriate counter on each PEBS event.

### 18.5.5.3 Precise Distribution Support on Fixed Counter 0

The Tremont microarchitecture supports the PDIR (Precise Distribution of Retired Instructions) facility, as described in Section 18.3.4.4.4, on Fixed Counter 0. Fixed Counter 0 counts the INST\_RETIRED.ALL event. PEBS skid for Fixed Counter 0 will be precisely one instruction.

This is in addition to the reduced skid PEBS behavior on IA32\_PMC0; see Section 18.5.3.1.2.

### 18.5.5.4 Compatibility Enhancements to Offcore Response MSRs

The Off-core Response facility is similar to that described in Section 18.5.3.2.

The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 are organized as shown below. RequestType bits are defined in Table 18-68, ResponseType bits in Table 18-69, and SnoopInfo bits in Table 18-70.

**Table 18-68. MSR\_OFFCORE\_RSPx Request Type Definition**

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data reads.
DEMAND_RFO	1	Counts all demand reads for ownership (RFO) requests and software based prefetches for exclusive ownership (prefetchw).
DEMAND_CODE_RD	2	Counts demand instruction fetches and L1 instruction cache prefetches.
COREWB_M	3	Counts modified write backs from L1 and L2.
HWPF_L2_DATA_RD	4	Counts prefetch (that bring data to L2) data reads.
HWPF_L2_RFO	5	Counts all prefetch (that bring data to L2) RFOs.
HWPF_L2_CODE_RD	6	Counts all prefetch (that bring data to L2 only) code reads.
Reserved	9:7	Reserved.
HWPF_L1D_AND_SWPF	10	Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).
STREAMING_WR	11	Counts all streaming stores.
COREWB_NONM	12	Counts non-modified write backs from L2.
Reserved	14:13	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O accesses that have any response type.
UC_RD	44	Counts uncached memory reads (PRd, UCRdF).
UC_WR	45	Counts uncached memory writes (wIL).
PARTIAL_STREAMING_WR	46	Counts partial (less than 64 byte) streaming stores (wCiL).
FULL_STREAMING_WR	47	Counts full, 64 byte streaming stores (wCiLF).

**Table 18-68. MSR\_OFFCORE\_RSPx Request Type Definition**

Bit Name	Offset	Description
L1WB_M	48	Counts modified WriteBacks from L1 that miss the L2.
L2WB_M	49	Counts modified WriteBacks from L2.

**Table 18-69. MSR\_OFFCORE\_RSPx Response Type Definition**

Bit Name	Offset	Description
ANY_RESPONSE	16	Catch all value for any response types.
L3_HIT_M	18	LLC/L3 Hit - M-state.
L3_HIT_E	19	LLC/L3 Hit - E-state.
L3_HIT_S	20	LLC/L3 Hit - S-state.
L3_HIT_F	21	LLC/L3 Hit - I-state.
LOCAL_DRAM	26	LLC/L3 Miss, DRAM Hit.
OUTSTANDING	63	Average latency of outstanding requests with the other counter counting number of occurrences; can also can be used to count occupancy.

**Table 18-70. MSR\_OFFCORE\_RSPx Snoop Info Definition**

Bit Name	Offset	Description
SNOOP_NONE	31	None of the cores were snooped. <ul style="list-style-type: none"> <li>▪ LLC miss and Dram data returned directly to the core.</li> </ul>
SNOOP_NOT_NEEDED	32	No snoop needed to satisfy the request. <ul style="list-style-type: none"> <li>▪ LLC hit and CV bit(s) (core valid) was not set.</li> <li>▪ LLC miss and Dram data returned directly to the core.</li> </ul>
SNOOP_MISS	33	A snoop was sent but missed. <ul style="list-style-type: none"> <li>▪ LLC hit and CV bit(s) was set but snoop missed (silent data drop in core), data returned from LLC.</li> <li>▪ LLC miss and Dram data returned directly to the core.</li> </ul>
SNOOP_HIT_NO_FWD	34	A snoop was sent but no data forward. <ul style="list-style-type: none"> <li>▪ LLC hit and CV bit(s) was set but no data forward from the core, data returned from LLC.</li> <li>▪ LLC miss and Dram data returned directly to the core.</li> </ul>
SNOOP_HIT_WITH_FWD	35	A snoop was sent and non-modified data was forward. <ul style="list-style-type: none"> <li>▪ LLC hit and CV bit(s) was set, non-modified data was forward from core.</li> </ul>
SNOOP_HITM	36	A snoop was sent and modified data was forward. <ul style="list-style-type: none"> <li>▪ LLC hit E or M and the CV bit(s) was set, modified data was forward from core.</li> </ul>
NON_DRAM_BIT	37	Target was non-DRAM system address, MMIO access. <ul style="list-style-type: none"> <li>▪ LLC miss and Non-Dram data returned.</li> </ul>

The Off-core Response capability behaves as follows:

- To specify a complete offcore response filter, software must properly program at least one RequestType and one ResponseType. A valid request type must have at least one bit set in the non-reserved bits of 15:0 or 49:44. A valid response type must be a non-zero value of one the following expressions:
  - Read requests:  
Any\_Response Bit | ('OR' of Supplier Info Bits) 'AND' ( 'OR' of Snoop Info Bits) | Outstanding Bit
  - Write requests:  
Any\_Response Bit | ('OR' of Supplier Info Bits) | Outstanding Bit
- When the ANY\_RESPONSE bit in the ResponseType is set, all other response type bits will be ignored.
- True Demand Cacheable Loads include neither L1 Prefetches nor Software Prefetches.
- Bits 15:0 and Bits 49:44 specifies the request type of a transaction request to the uncore. This is described in Table 18-68.
- Bits 30:16 specifies common supplier information.
- "Outstanding Requests" (bit 63) is only available on MSR\_OFFCORE\_RSP0; a #GP fault will occur if software attempts to write a 1 to this bit in MSR\_OFFCORE\_RSP1. It is mutually exclusive with any ResponseType. Software must guarantee that all other ResponseType bits are set to 0 when the "Outstanding Requests" bit is set.
- "Outstanding Requests" bit 63 can enable measurement of the average latency of a specific type of off-core transaction; two programmable counters must be used simultaneously and the RequestType programming for MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 must be the same when using this Average Latency feature. See Section 18.5.2.3 for further details.

## 18.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS)

### 18.6.1 Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 18-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 8.6, "Detecting Hardware Multi-Threading Support and Topology," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). As a result, the unit mask field (for example, IA32\_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 18-71. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see: <https://perfmon-events.intel.com/>) and for Intel Core Duo processors. Such events are referred to as core-specific events.

**Table 18-71. Core Specificity Encoding within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	Description
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 18-72.

**Table 18-72. Agent Specificity Encoding within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	Description
0	This agent
1	Include all agents

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 18-73.

**Table 18-73. HW Prefetch Qualification Encoding within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	Description
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Agent-specificity and hardware prefetch qualification are mutually exclusive.

In addition, some L2 events permit qualifications that distinguish cache coherent states. The sub-field definition for cache coherency state qualification is shown in Table 18-74. If no bits in the MESI qualification sub-field are set for an event that requires setting MESI qualification bits, the event count will not increment.

**Table 18-74. MESI Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 11	Counts modified state
Bit 10	Counts exclusive state
Bit 9	Counts shared state
Bit 8	Counts Invalid state

### 18.6.2 Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32\_PERFEVTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 18.6.2.1). IA32\_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 18-1. Starting with Intel



Core 2 processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 18.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits can be found at: <https://perfmon-events.intel.com/>. The UMASK field may contain sub-fields identical to those listed in Table 18-71, Table 18-72, Table 18-73, and Table 18-74. One or more of these sub-fields may apply to specific events on an event-by-event basis.

In addition, the UMASK field may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 18-75.

**Table 18-75. Bus Snoop Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 11	HITM response
Bit 10	Reserved
Bit 9	HIT response
Bit 8	CLEAN response

There are also non-architectural events that support qualification of different types of snoop operation. The corresponding bit field for snoop type qualification are listed in Table 18-76.

**Table 18-76. Snoop Type Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 9:8	Description
Bit 9	CMP2I snoops
Bit 8	CMP2S snoops

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.

#### NOTE

Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

### 18.6.2.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural Perfmon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. See Table 18-2 for details of the PMC addresses and what these events count.

Programming the fixed-function performance counters does not involve any of the IA32\_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR IA32\_FIXED\_CTR\_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 18-41. Two sub-fields are defined for each control. See Figure 18-41; bit fields are:

- **Enable field (low 2 bits in each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

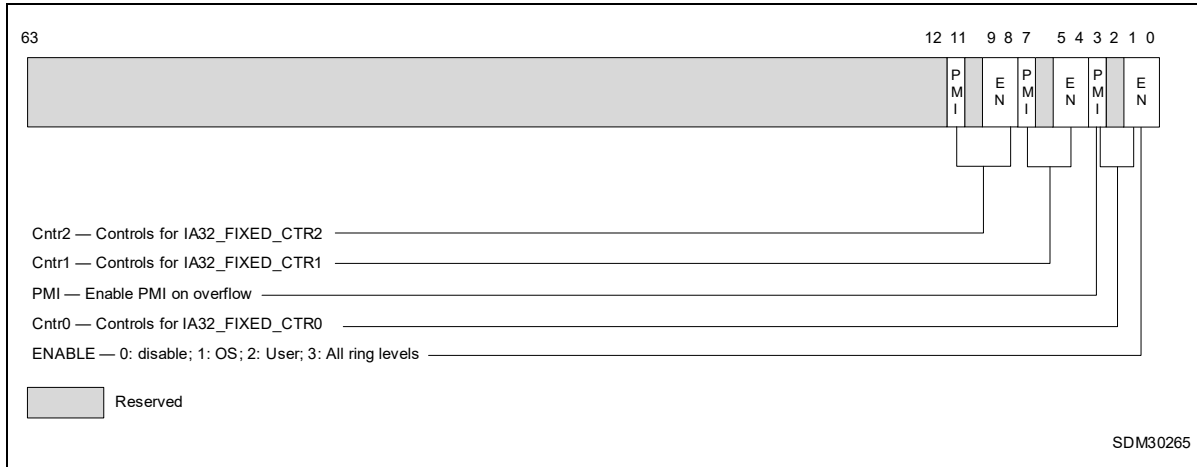


Figure 18-41. Layout of IA32\_FIXED\_CTR\_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

### 18.6.2.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e. enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSR:

- MSR\_PERF\_GLOBAL\_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (IA32\_FIXED\_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR\_PERF\_GLOBAL\_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (IA32\_FIXED\_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32\_FIXED\_CTRx) or general-purpose PMCs via a single WRMSR.

MSR\_PERF\_GLOBAL\_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 18-42). Each enable bit in MSR\_PERF\_GLOBAL\_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or IA32\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

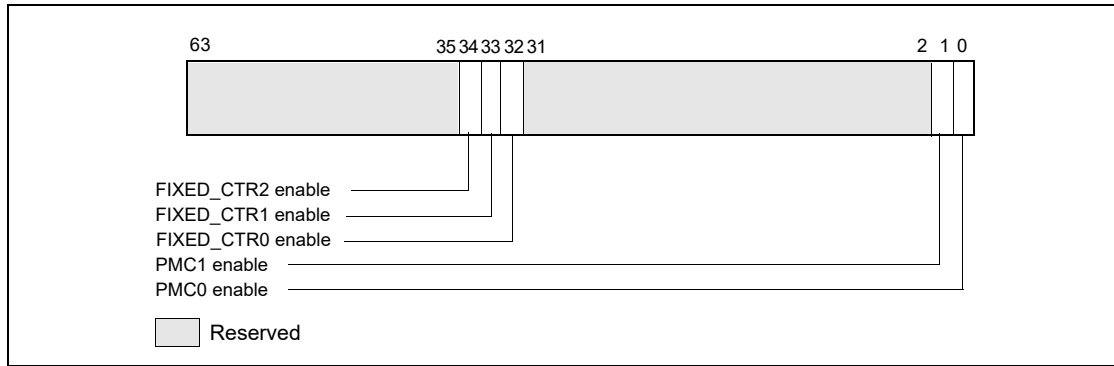


Figure 18-42. Layout of MSR\_PERF\_GLOBAL\_CTRL MSR

MSR\_PERF\_GLOBAL\_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR\_PERF\_GLOBAL\_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR\_PERF\_GLOBAL\_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 18-43). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

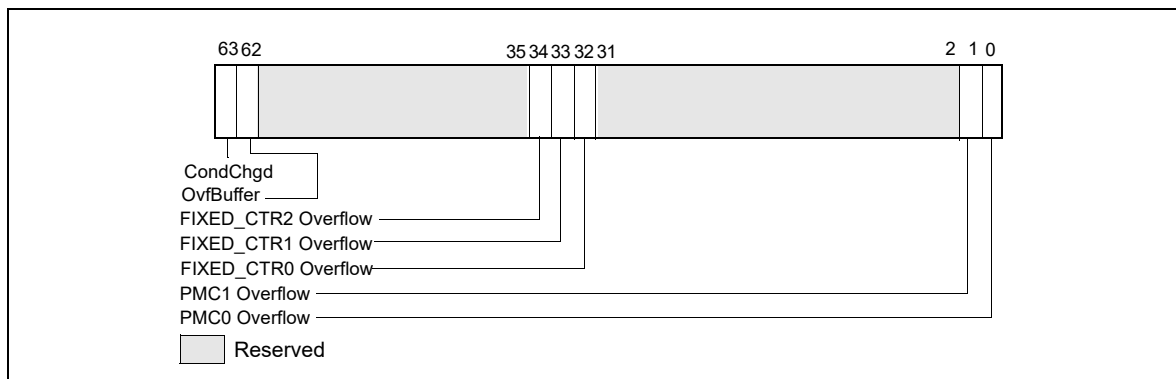


Figure 18-43. Layout of MSR\_PERF\_GLOBAL\_STATUS MSR

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR\_PERF\_GLOBAL\_STATUS.

MSR\_PERF\_GLOBAL\_OVF\_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-44). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

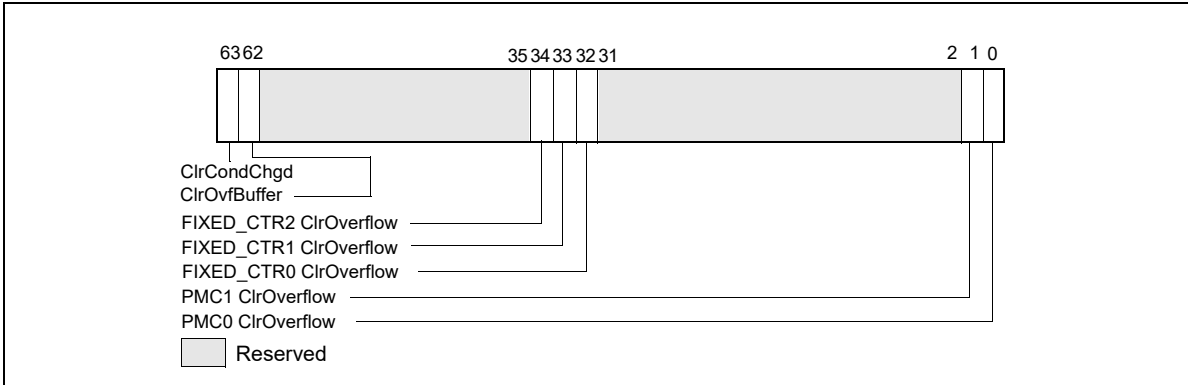


Figure 18-44. Layout of MSR\_PERF\_GLOBAL\_OVF\_CTRL MSR

### 18.6.2.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst<sup>®</sup> microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture does not require special MSR programming control (see Section 18.6.3.6, "At-Retirement Counting"), but is limited to IA32\_PMC0. See Table 18-77 for a list of events available to processors based on Intel Core microarchitecture.

Table 18-77. At-Retirement Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

### 18.6.2.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.6.2.4.2 and Section 17.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 18-78. The procedure for detecting availability of PEBS is the same as described in Section 18.6.3.8.1.

**Table 18-78. PEBS Performance Events for Intel Core Microarchitecture**

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

#### 18.6.2.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32\_PMC0 only. Use the following procedure to set up the processor and IA32\_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 17-8 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32\_PEBS\_ENABLE MSR.
3. Set up the IA32\_PMC0 performance counter and IA32\_PERFEVTSEL0 for an event listed in Table 18-78.

#### 18.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32\_PERF\_CAPABILITIES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version id equals 2 or higher. The bit fields of IA32\_PERF\_CAPABILITIES are defined in Table 2-2 of Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*. The relevant bit fields that governs PEBS are:

- PEBSTrap [bit 6]: When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- PEBSSaveArchRegs [bit 7]: When set, PEBS will save architectural register and state information according to the encoded value of the PEBSTrapFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1.
- PEBSTrapFormat [bits 11:8]: Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (See Section 18.6.3.8).
  - 0001B: PEBS record includes additional information of IA32\_PERF\_GLOBAL\_STATUS and load latency data. (See Section 18.3.1.1.1).
  - 0010B: PEBS record includes additional information of IA32\_PERF\_GLOBAL\_STATUS, load latency data, and TSX tuning information. (See Section 18.3.6.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32\_PERF\_GLOBAL\_STATUS at offset 90H. (See Section 18.3.8.1.1).
  - 0100B: PEBS record contents are defined by elections in MSR\_PEBS\_DATA\_CFG. (See Section 18.9.2.3).

### 18.6.2.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.1, “64 Bit Format of the DS Save Area,” for guidelines when writing the DS ISR.

The service routine can query MSR\_PERF\_GLOBAL\_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR\_PERF\_GLOBAL\_OVF\_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 18-79.

**Table 18-79. Requirements to Program PEBS**

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS.	<ul style="list-style-type: none"> <li>IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set.</li> <li>IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.</li> </ul>	
Ensure counters are in disabled.	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: <ul style="list-style-type: none"> <li>Clear all counters if “Counter Freeze on PMI” is not enabled.</li> <li>If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled.</li> </ul> Counters MUST be stopped before writing. <sup>1</sup>	Optional
Disable PEBS.	Clear ENABLE PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write “sample-after” values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	<ul style="list-style-type: none"> <li>Set local enable bit 22 - 1.</li> <li>Do NOT set local counter PMI/INT bit, bit 20 - 0.</li> <li>Event programmed must be PEBS capable.</li> </ul>	<ul style="list-style-type: none"> <li>Set appropriate OVF_PMI bits - 1.</li> <li>Only CCCR for MSR_IQ_COUNTER4 support PEBS.</li> </ul>
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

**NOTES:**

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

### 18.6.2.4.4 Re-configuring PEBS Facilities

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32\_DS\_AREA).

### 18.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMS instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMS instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32\_MISC\_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32\_DS\_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR\_PEBS\_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 18-80 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events can be found at: <https://perfmon-events.intel.com/>.

**Table 18-80. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)**

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCR0	360H	MSR_BSU_ESCRO	7	3A0H
					MSR_FSB_ESCRO	6	3A2H
					MSR_MOB_ESCRO	2	3AAH
					MSR_PMH_ESCRO	4	3ACH
					MSR_BPU_ESCRO	0	3B2H
					MSR_IS_ESCRO	1	3B4H
					MSR_ITLB_ESCRO	3	3B6H
					MSR_IX_ESCRO	5	3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCRO	7	3A0H
					MSR_FSB_ESCRO	6	3A2H
					MSR_MOB_ESCRO	2	3AAH
					MSR_PMH_ESCRO	4	3ACH
					MSR_BPU_ESCRO	0	3B2H
					MSR_IS_ESCRO	1	3B4H
					MSR_ITLB_ESCRO	3	3B6H
					MSR_IX_ESCRO	5	3C8H

**Table 18-80. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)**

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAA_T_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAA_T_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAA_T_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAA_T_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0 <sup>1</sup> MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH



**Table 18-80. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)**

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0	4	3B8H
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0 <sup>1</sup>	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1 <sup>1</sup>	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH
					MSR_IQ_COUNTER3	15	30FH
MSR_CRU_ESCR3	5	3CDH					
MSR_CRU_ESCR5	6	3E1H					
MSR_IQ_ESCR1 <sup>1</sup>	0	3BBH					
MSR_RAT_ESCR1	2	3BDH					
MSR_ALF_ESCR1	1	3CBH					
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H			
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0 <sup>1</sup>	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1 <sup>1</sup>	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH

**NOTES:**

1. MSR\_IQ\_ESCR0 and MSR\_IQ\_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

The at-retirement counting mechanism includes facilities for tagging  $\mu$ ops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting** — A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.

- Interrupt-based event sampling** — A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted. When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.
- Processor event-based sampling (PEBS)** — In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

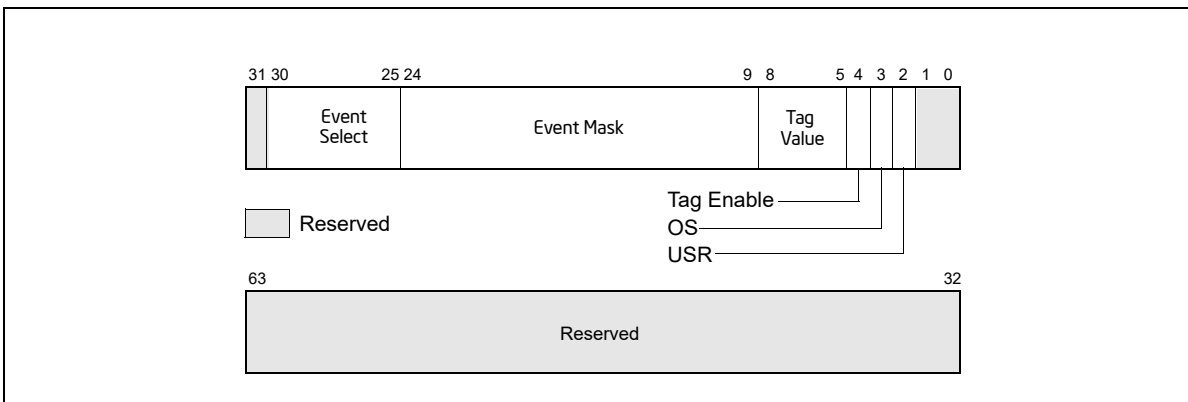
The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

### 18.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-80) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 18-80) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 18-45 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- USR flag, bit 2** — When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
- OS flag, bit 3** — When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)



**Figure 18-45. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without Intel HT Technology Support**

- Tag enable, bit 4** — When set, enables tagging of  $\mu$ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, "At-Retirement Counting."
- Tag value field, bits 5 through 8** — Selects a tag value to associate with a  $\mu$ op to assist in at-retirement event counting.
- Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and MSR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor MSR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 18-80 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

### 18.6.3.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR's (see Table 18-80). The counter pairs are partitioned into four groups:

- The BPU group, includes two performance counter pairs:
  - MSR\_BPU\_COUNTER0 and MSR\_BPU\_COUNTER1.
  - MSR\_BPU\_COUNTER2 and MSR\_BPU\_COUNTER3.
- The MS group, includes two performance counter pairs:
  - MSR\_MS\_COUNTER0 and MSR\_MS\_COUNTER1.
  - MSR\_MS\_COUNTER2 and MSR\_MS\_COUNTER3.
- The FLAME group, includes two performance counter pairs:
  - MSR\_FLAME\_COUNTER0 and MSR\_FLAME\_COUNTER1.
  - MSR\_FLAME\_COUNTER2 and MSR\_FLAME\_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR\_IQ\_COUNTER0 and MSR\_IQ\_COUNTER1.
  - MSR\_IQ\_COUNTER2 and MSR\_IQ\_COUNTER3.
  - MSR\_IQ\_COUNTER4 and MSR\_IQ\_COUNTER5.

The MSR\_IQ\_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR\_BPU\_COUNTER0 can start MSR\_BPU\_COUNTER2 and vice versa, and MSR\_BPU\_COUNTER1 can start MSR\_BPU\_COUNTER3 and vice versa (see Section 18.6.3.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 18-46). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or, if ECX[31] is set to 1, the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits. In such cases, counter bits 31:0 are written to EAX, while 0 is written to EDX.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

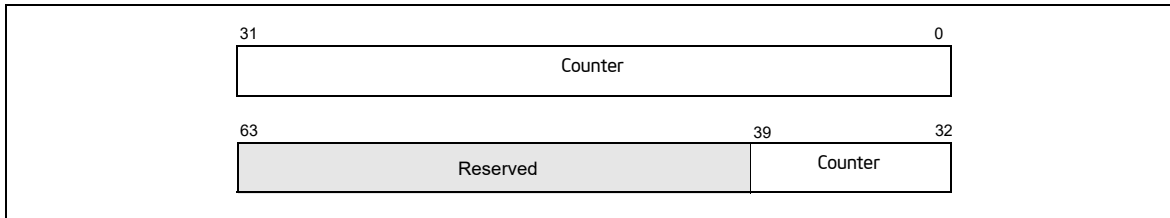


Figure 18-46. Performance Counter (Pentium 4 and Intel Xeon Processors)

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

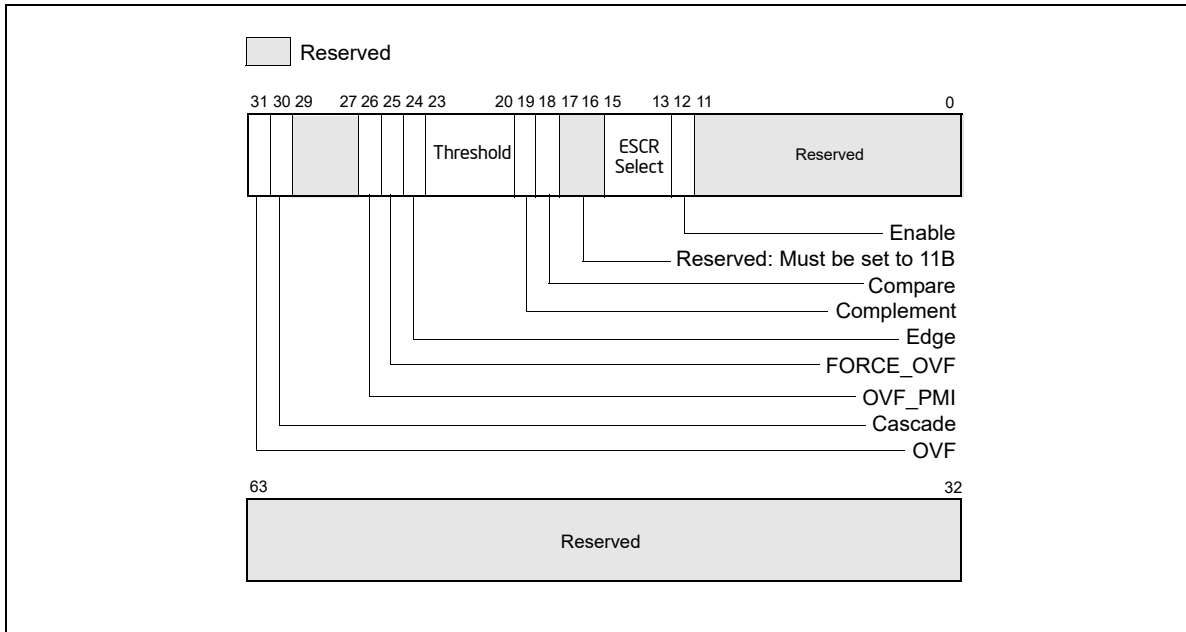
Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

### 18.6.3.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 18-80). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 18-47 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.6.3.5.2, "Filtering Events"). The complement flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.6.3.5.2, "Filtering Events").
- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.



**Figure 18-47. Counter Configuration Control Register (CCCR)**

- **FORCE\_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF\_PMI flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.
- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.6.3.2, “Performance Counters,” for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.
2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.
3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.
4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.
5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one “stage” forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCCRs are discussed in greater detail in Section 18.6.3.5, “Programming the Performance Counters for Non-Retirement Events.”

### 18.6.3.4 Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 17.4.5, “Branch Trace Store (BTS),” and Section 18.6.3.8, “Processor Event-Based Sampling (PEBS),” for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 17.4.9, “BTS and DS Save Area.”

### 18.6.3.5 Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event.
3. Match the CCCR Select value and ESCR name to a value listed in Table 18-80; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which they are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

#### 18.6.3.5.1 Selecting Events to Count

There is a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event, setup information is provided. Table 18-81 gives an example of one of the events.

**Table 18-81. Event Example**

Event Name	Event Parameters	Parameter Value	Description
branch_retired			Counts the retirement of a branch. Specify one or more mask bits to select any combination of branch taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 15-3 for the addresses of the ESCR MSRs.
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 15-3.
	ESCR Event Select	06H	ESCR[31:25]
	ESCR Event Mask	Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch Not-taken Predicted Branch Not-taken Mispredicted Branch Taken Predicted Branch Taken Mispredicted
	CCCR Select	05H	CCCR[15:13]

Table 18-81. Event Example (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

Event Parameters are described below.

- **ESCR restrictions** — Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.
- **Counter numbers per ESCR** — Lists which performance counters are associated with each ESCR. Table 18-80 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.
- **ESCR event select** — Gives the value to be placed in the event select field of the ESCR to select the event.
- **ESCR event mask** — Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.
- **CCCR select** — Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 18-80.
- **Event specific notes** — Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.
- **Can support PEBS** — Indicates if PEBS is supported for the event (only supplied for at-retirement events).
- **Requires additional MSR for tagging** — Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events).

#### NOTE

The performance-monitoring events found at <https://perfmon-events.intel.com/> are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

An event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.
3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 18-80.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.
6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.

**NOTE**

Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 18.6.3.5.2, "Filtering Events."

**18.6.3.5.2 Filtering Events**

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 2 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counter's threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

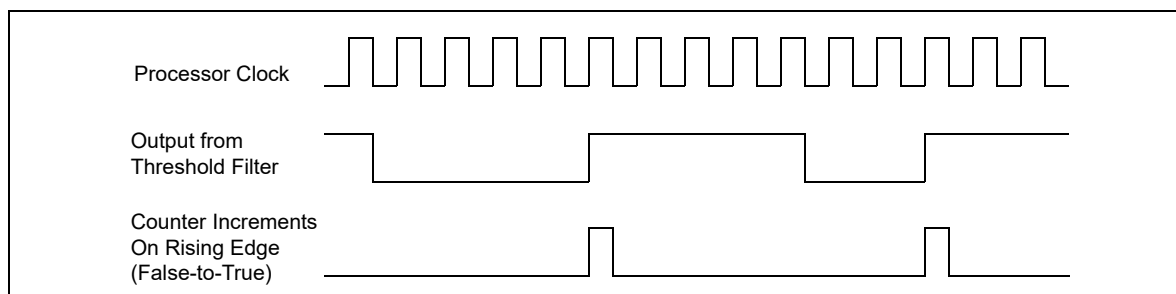
If the compare flag is set, then a "greater than" or a "less than or equal to" comparison of the input value vs. a threshold value can be made. The complement flag selects "less than or equal to" (flag set) or "greater than" (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, then any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a "rising edge" event; that is, a false-to-true transition. Figure 18-48 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 18.6.3.5.1, "Selecting Events to Count."

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
  - Set the compare flag.
  - Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
  - Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 18.6.3.5.3, "Starting Event Counting."



**Figure 18-48. Effects of Edge Filtering**



### 18.6.3.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter's CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 18.6.3.5.5, "Halting Event Counting").

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.2, "Filtering Events."

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 18.6.3.5.4, "Reading a Performance Counter's Count."

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 18.6.3.5.6, "Cascading Counters").

### 18.6.3.5.4 Reading a Performance Counter's Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 18.6.3.2, "Performance Counters." These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 18-80 used as an operand.

This setup procedure is continued in the next section, Section 18.6.3.5.5, "Halting Event Counting."

### 18.6.3.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.4, "Reading a Performance Counter's Count."

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter's CCCR MSR or clear the OVF flag in the alternate counter's CCCR MSR.

### 18.6.3.5.6 Cascading Counters

As described in Section 18.6.3.2, "Performance Counters," eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 18-80). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR\_IQ\_CCCR0 through MSR\_IQ\_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 18-47 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR\_MO B\_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It's possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14

cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

**Example 18-1. Counting Events**

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 18.6.3.5.1, "Selecting Events to Count." Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 18.6.3.5.8, "Generating an Interrupt on Overflow."

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

**18.6.3.5.7 EXTENDED CASCADING**

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily\_DisplayModel 0F\_02, 0F\_03, 0F\_04, 0F\_06. This feature uses bit 11 in CCCRs associated with the IQ block. See Table 18-82.

**Table 18-82. CCR Names and Bit Positions**

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1 2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INT00	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INT03	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INT04	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INT05	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily\_DisplayModel signature of 0F\_02. For processors with CPUID DisplayFamily\_DisplayModel signature of 0F\_00 and 0F\_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINT0y bits is used.

**Example 18-2. Scenario for Extended Cascading**

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR\_CRU\_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR\_IQ\_CCCR0. This enables CASCNT4INTO0 and OVF\_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR\_CRU\_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR\_IQ\_CCCR4 (set ENABLE bit, but not OVF\_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

#### 18.6.3.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR\_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 18.6.3.5.6, "Cascading Counters." Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE\_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

#### 18.6.3.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no\_event, which generally has a select value of 0).

### 18.6.3.6 At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called "at-retirement counting."

There are predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire** — In at-retirement event descriptions, the term "bogus" refers to instructions or  $\mu$ ops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or  $\mu$ ops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and  $\mu$ ops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, `Instruction_Retired` and `Uops_Retired`) can count instructions or  $\mu$ ops that are retired based on the characterization of bogus" versus non-bogus.
- **Tagging** — Tagging is a means of marking  $\mu$ ops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per  $\mu$ op and a direct count of the event would not provide an indication of how many  $\mu$ ops encountered that event. The tagging mechanisms allow a  $\mu$ op to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per  $\mu$ op, rather than once per event. For example, a  $\mu$ op may encounter a cache miss more than once during its life time, but a "Miss Retired" metric (that counts the number of retired  $\mu$ ops that encountered a cache miss) will increment only once for that  $\mu$ op. A "Miss Retired" metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the *Intel Pentium 4 Processor Optimization Reference Manual* (see Section 1.4, "Related Literature").
- **Replay** — To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules  $\mu$ ops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied,  $\mu$ ops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of  $\mu$ ops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist** — When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of  $\mu$ ops before they begin and are costly.

#### 18.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and  $\mu$ ops that encountered a specified event. For a subset of the at-retirement events, a  $\mu$ op may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count  $\mu$ ops tagged with that mechanism:

- **Front-end tagging** — This mechanism pertains to the tagging of  $\mu$ ops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the `Front_end_event` event.
- **Execution tagging** — This mechanism pertains to the tagging of  $\mu$ ops that encountered execution events (for example, instruction types) and are counted with the `Execution_Event` event.

- **Replay tagging** — This mechanism pertains to tagging of  $\mu$ ops whose retirement is replayed (for example, a cache miss) and are counted with the `Replay_event` event. Branch mispredictions are also tagged with this mechanism.
- **No tags** — This mechanism does not use tags. It uses the `Instr_retired` and the `Uops_retired` events.

Each tagging mechanism is independent from all others; that is, a  $\mu$ op that has been tagged using one mechanism will not be detected with another mechanism's tagged- $\mu$ op detector. For example, if  $\mu$ ops are tagged using the front-end tagging mechanisms, the `Replay_event` will not count those as tagged  $\mu$ ops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of  $\mu$ ops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of  $\mu$ ops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

There are performance monitoring events that support at-retirement counting: specifically the `Front_end_event`, `Execution_event`, `Replay_event`, `Inst_retired` and `Uops_retired` events. The following sections describe the tagging mechanisms for using these events to tag  $\mu$ op and count tagged  $\mu$ ops.

#### 18.6.3.6.2 Tagging Mechanism for `Front_end_event`

The `Front_end_event` counts  $\mu$ ops that have been tagged as encountering any of the following events:

- **$\mu$ op decode events** — Tagging  $\mu$ ops for  $\mu$ op decode events requires specifying bits in the `ESCR` associated with the performance-monitoring event, `Uop_type`.
- **Trace cache events** — Tagging  $\mu$ ops for trace cache events may require specifying certain bits in the `MSR_TC_PRECISE_EVENT` MSR.

The MSRs that are supported by the front-end tagging mechanism must be set and one or both of the `NBOGUS` and `BOGUS` bits in the `Front_end_event` event mask must be set to count events. None of the events currently supported requires the use of the `MSR_TC_PRECISE_EVENT` MSR.

#### 18.6.3.6.3 Tagging Mechanism For `Execution_event`

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* `ESCR` is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* `ESCR` is used to detect  $\mu$ ops that have been tagged with that tag value identifier using `Execution_event` for the event selection.

The upstream `ESCR` that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular  $\mu$ op. The value selected for the tag value should coincide with the event mask selected in the downstream `ESCR`. For example, if a tag value of 1 is set, then the event mask of `NBOGUS0` should be enabled, correspondingly in the downstream `ESCR`. The downstream `ESCR` detects and counts tagged  $\mu$ ops. The normal (not tag value) mask bits in the downstream `ESCR` specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. The tag enable and tag value bits are irrelevant for the downstream `ESCR` used to select the `Execution_event`.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 18.6.3.8.3, "Setting Up the PEBS Buffer.") It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream `ESCR` or multiple mask bits in the downstream `ESCR`. For example, use a tag value of 3H in the upstream `ESCR` and use `NBOGUS0/NBOGUS1` in the downstream `ESCR` event mask.

#### 18.6.3.7 Tagging Mechanism for `Replay_event`

The replay mechanism enables tagging of  $\mu$ ops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of  $\mu$ op that may experience the replay in the `MSR_PEBS_MATRIX_VERT` MSR and selecting the type of event in the `MSR_PEBS_ENABLE` MSR. Replay tagging must also be enabled with the `UOP_Tag` flag (bit 24) in the `MSR_PEBS_ENABLE` MSR.

The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 17.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA\_32\_PEBS\_ENABLE\_MSR. Each of these metrics requires that the Replay\_Event be used to count the tagged  $\mu$ ops.

### 18.6.3.8 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 17.4.9, “BTS and DS Save Area”). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution\_event, Front\_end\_event, and Replay\_event. Also, PEBS can only be carried out using the one performance counter, the MSR\_IQ\_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32\_PMC0 and IA32\_PERFVTSEL0 MSRs (See Section 18.6.2.4).

#### 18.6.3.8.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS\_UNAVAILABLE flag in the IA32\_MISC\_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR\_PEBS\_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR\_PEBS\_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32\_DS\_AREA MSR can be programmed to point to the DS save area.

#### 18.6.3.8.2 Setting Up the DS Save Area

Section 17.4.9.2, “Setting Up the DS Save Area,” describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

#### 18.6.3.8.3 Setting Up the PEBS Buffer

Only the MSR\_IQ\_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 17-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR\_PEBS\_ENABLE MSR.
3. Set up the MSR\_IQ\_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS.



#### 18.6.3.8.4 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.5, “Writing the DS Interrupt Service Routine,” for guidelines for writing the DS ISR.

#### 18.6.3.8.5 Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT.

The DS mechanism is available in real address mode.

#### 18.6.3.9 Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

### 18.6.4 Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 18.6.3. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

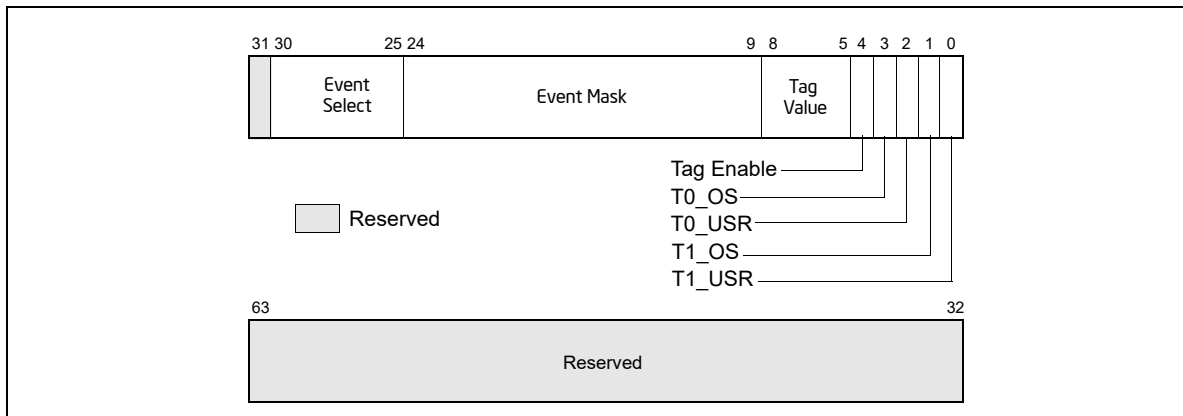
The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR\_PEBS\_ENABLE, MSR\_PEBS\_MATRIX\_VERT, and MSR\_TC\_PRECISE\_EVENT.

#### 18.6.4.1 ESCR MSRs

Figure 18-49 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1\_USR flag, bit 0** — When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.



**Figure 18-49. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology**

- **T1\_OS flag, bit 1** — When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1\_OS and T1\_USR flags are set, thread 1 events are counted at all privilege levels.)
- **T0\_USR flag, bit 2** — When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.
- **T0\_OS flag, bit 3** — When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0\_OS and T0\_USR flags are set, thread 0 events are counted at all privilege levels.)
- **Tag enable, bit 4** — When set, enables tagging of  $\mu$ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, “At-Retirement Counting.”
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a  $\mu$ op to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0\_OS and T0\_USR flags and the T1\_OS and T1\_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 8.4.5, “Identifying Logical Processors in an MP System,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 18.6.4.4, “Performance Monitoring Events”). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

### 18.6.4.2 CCCR MSRs

Figure 18-50 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field, bits 16 and 17** — Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:
  - **00** — None. Count only when neither logical processor is active.



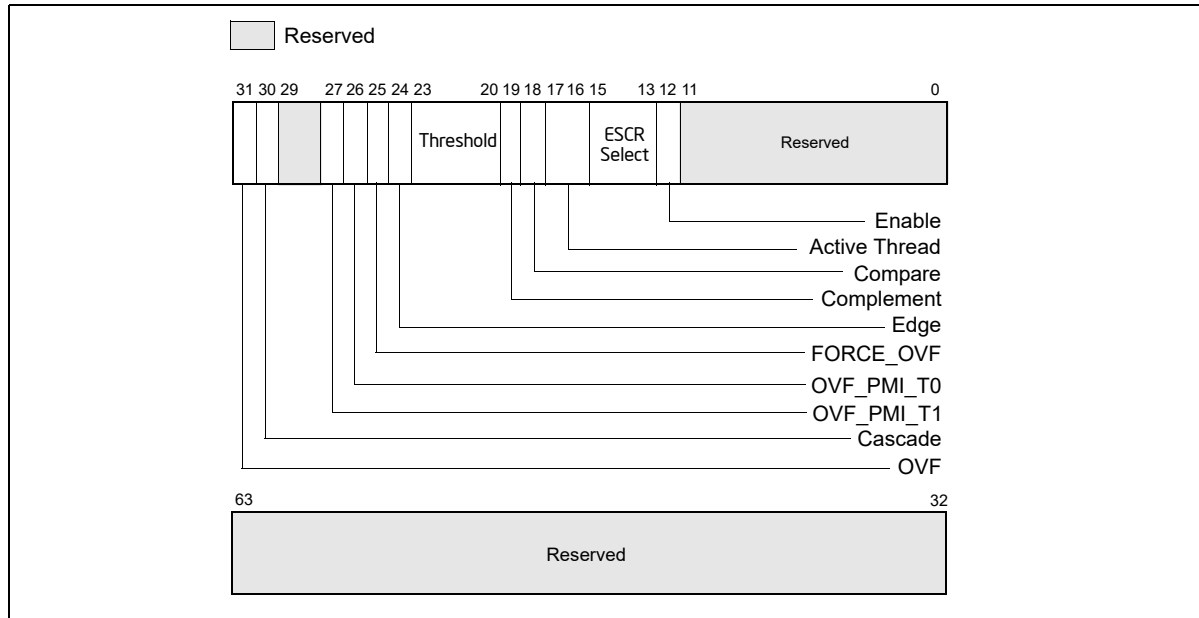
**01** — Single. Count only when one logical processor is active (either 0 or 1).

**10** — Both. Count only when both logical processors are active.

**11** — Any. Count when either logical processor is active.

A halted logical processor or a logical processor in the “wait for SIPI” state is considered inactive.

- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.



**Figure 18-50. Counter Configuration Control Register (CCCR)**

- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.6.3.5.2, “Filtering Events”). The compare flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.6.3.5.2, “Filtering Events”).
- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE\_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF\_PMI\_T0 flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF\_PMI\_T1 flag, bit 27** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.
- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.6.3.2, “Performance Counters,” for further details); when clear, disables cascading of counters.

- **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

### 18.6.4.3 IA32\_PEBS\_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR\_PEBS\_ENABLE MSR: bit 25 (ENABLE\_PEBS\_MY\_THR) and 26 (ENABLE\_PEBS\_OTH\_THR) respectively. These bits do not explicitly identify a specific logical processor by logic processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running (“my thread”) or for the other logical processor in the physical package on which the agent is not running (“other thread”).

PEBS is supported for only a subset of the at-retirement events: Execution\_event, Front\_end\_event, and Replay\_event. Also, PEBS can be carried out only with two performance counters: MSR\_IQ\_CCCR4 (MSR address 370H) for logical processor 0 and MSR\_IQ\_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE\_PEBS\_MY\_THR and ENABLE\_PEBS\_OTH\_THR bits in the MSR\_PEBS\_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

### 18.6.4.4 Performance Monitoring Events

When Intel Hyper-Threading Technology is active, many performance monitoring events can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS)** — The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI)** — The event cannot be qualified as being associated with a specific logical processor.

If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 18-83) depends only on the setting of the T0\_USR and T0\_OS flags in the ESCR being used to set up the event counter. The T1\_USR and T1\_OS flags have no effect on the count.

**Table 18-83. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events**

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while T1 in USR	Counts while T1 in OS or USR	Counts while T1 in OS
T0_OS/T0_USR = 01	Counts while T0 in USR	Counts while T0 in USR or T1 in USR	Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 11	Counts while T0 in OS or USR	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in OS
T0_OS/T0_USR = 10	Counts T0 in OS	Counts T0 in OS or T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI, the effect of selectively specifying T0\_USR, T0\_OS, T1\_USR, T1\_OS bits is shown in Table 18-84.

**Table 18-84. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events**

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 01	Counts while (a) T0 in USR or (b) T1 in USR	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 11	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 0	Counts while (a) T0 in OS or (b) T1 in OS	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS

### 18.6.4.5 Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

#### 18.6.4.5.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCRs to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the `global_power_events` and specify the `RUNNING` sub-event mask and the desired `T0_OS/T0_USR/T1_OS/T1_USR` bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

#### 18.6.4.5.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than "no\_event"; the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to "1".
4. Set the threshold to "15" and the complement to "1" in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g. by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an `HLT` instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical processors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

The Non-sleep Clockticks mechanism uses a filtering mechanism in CCCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

### 18.6.5 Performance Monitoring and Dual-Core Technology

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but two logical processors in the same core share performance monitoring resources (see Section 18.6.4, "Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture").

### 18.6.6 Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 18.1 and Section 18.6.4) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 18-51.

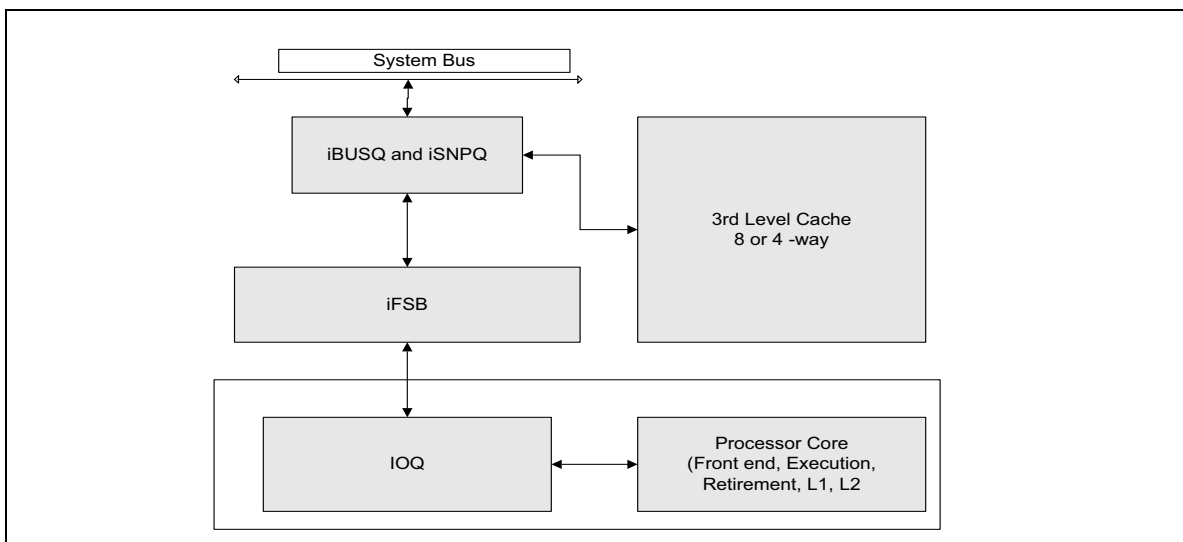


Figure 18-51. Block Diagram of 64-bit Intel Xeon Processor MP with 8-MByte L3

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPKC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- IBUSQ event** — This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR\_IFSB\_IBUSQ0 and MSR\_IFSB\_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 18-52.

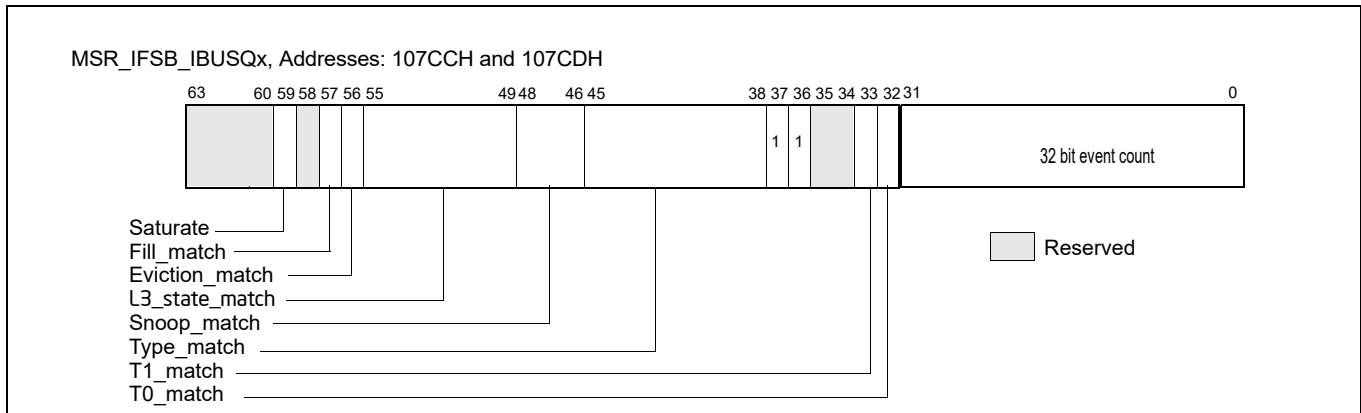


Figure 18-52. MSR\_IFSB\_IBUSQx, Addresses: 107CCH and 107CDH

- ISNPQ event** — This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR\_IFSB\_ISNPQ0 and MSR\_IFSB\_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 18-53.

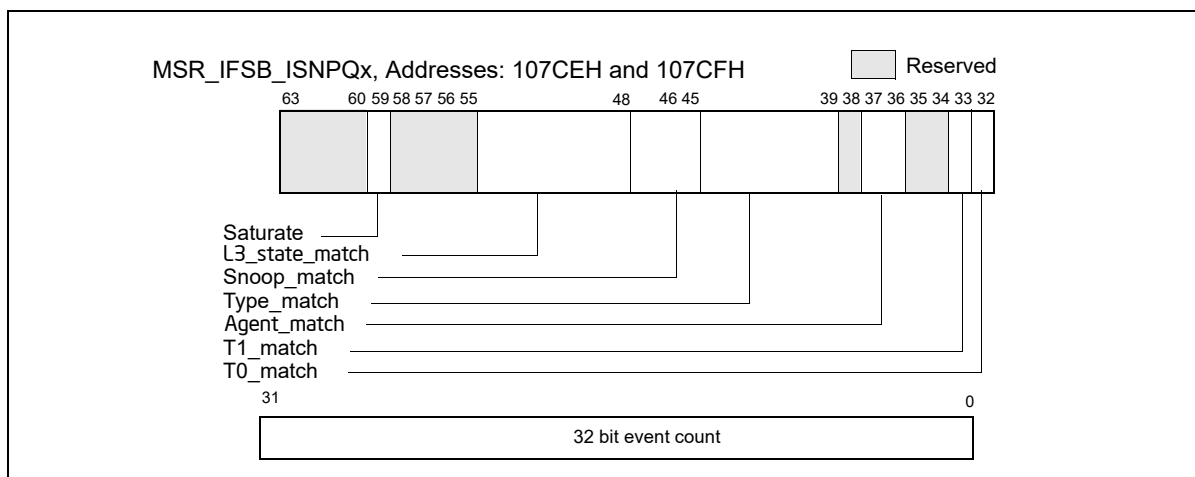


Figure 18-53. MSR\_IFSB\_ISNPQx, Addresses: 107CEH and 107CFH

- EFSB event** — This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR\_EFSB\_DRDY0 and MSR\_EFSB\_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 18-54.

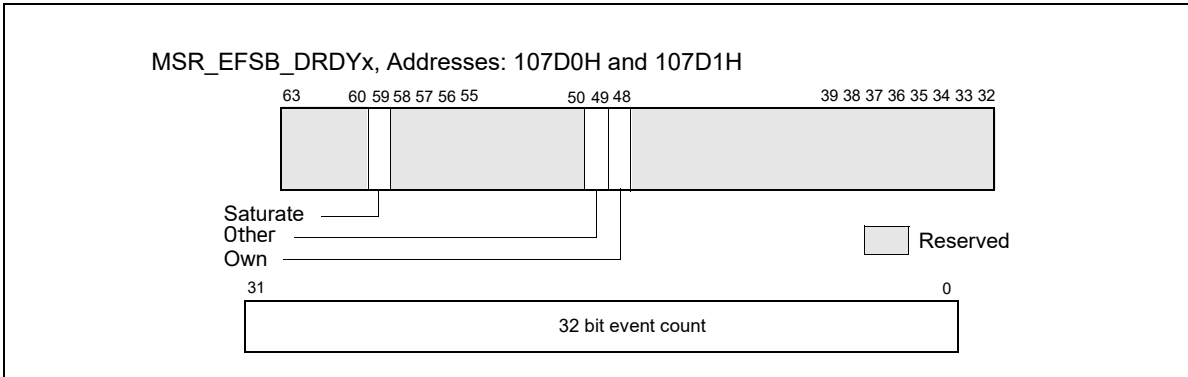


Figure 18-54. MSR\_EFSB\_DRDYx, Addresses: 107D0H and 107D1H

- IBUSQ Latency event** — This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR\_IFSB\_CTRL6[bit 26] to 1; the count freezes after software sets MSR\_IFSB\_CTRL6[bit 26] to 0. MSR\_IFSB\_CNTR7 acts as a 64-bit event counter for this event. See Figure 18-55.

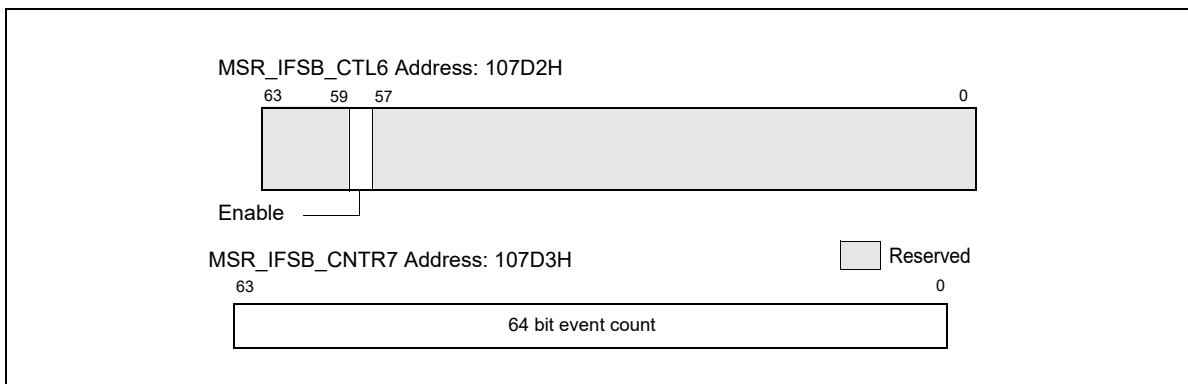


Figure 18-55. MSR\_IFSB\_CTL6, Address: 107D2H;  
 MSR\_IFSB\_CNTR7, Address: 107D3H

### 18.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems

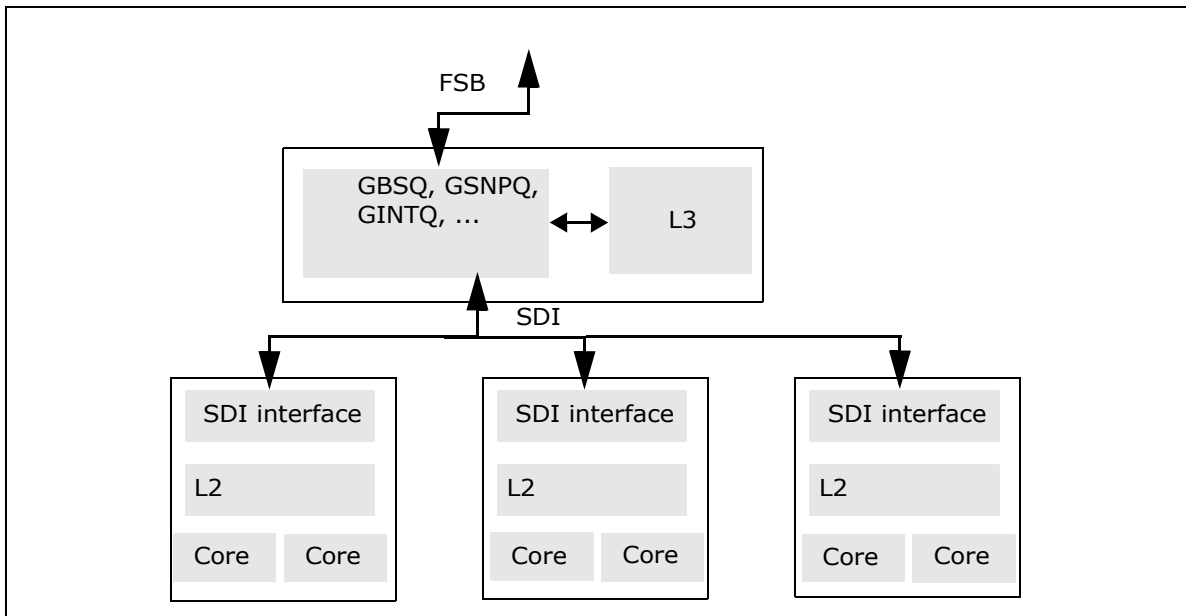
The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily\_DisplayModel value of 06\_1DH (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through addi-

tional control logic. See Figure 18-56 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 18-56 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.



**Figure 18-56. Block Diagram of Intel Xeon Processor 7400 Series**

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 18.1 and Section 18.6.4) apply to Intel Xeon processor 7100 series. The MSR used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily\_DisplayModel signature 06\_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ\_allocation and IOQ\_active\_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

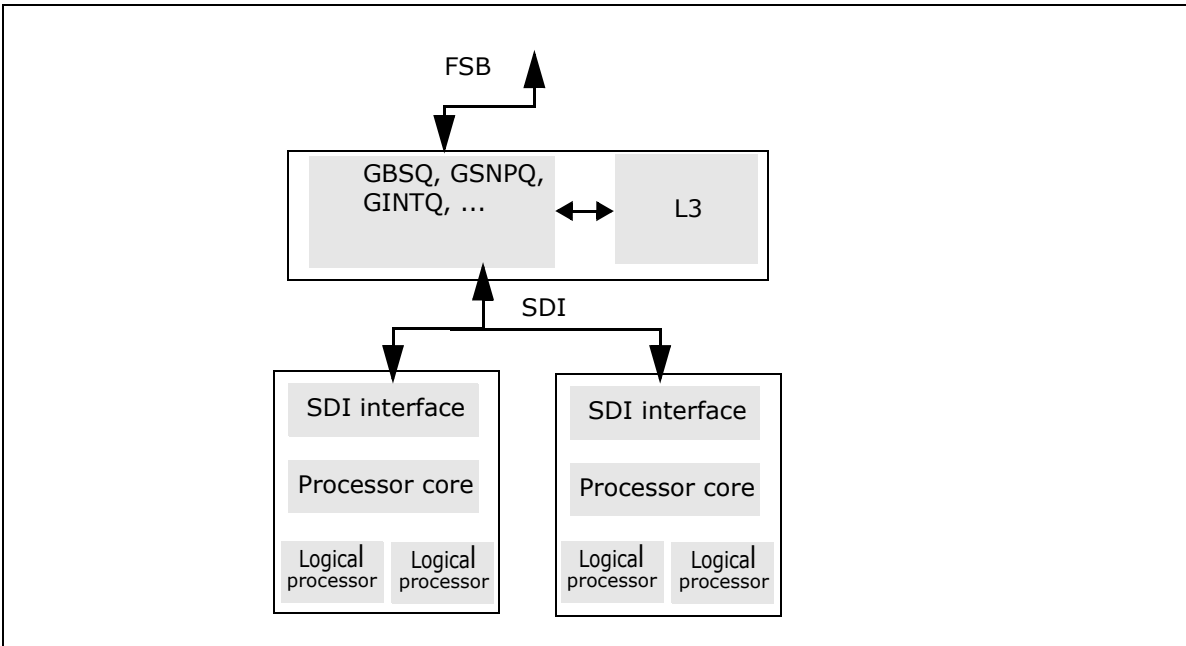


Figure 18-57. Block Diagram of Intel Xeon Processor 7100 Series

### 18.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR\_EMON\_L3\_GL\_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR\_EMON\_L3\_CTR\_CTL0 and MSR\_EMON\_L3\_CTR\_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR\_EMON\_L3\_CTR\_CTL2 and MSR\_EMON\_L3\_CTR\_CTL3) are dedicated to counting GSNPQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Four MSRs (MSR\_EMON\_L3\_CTR\_CTL4, MSR\_EMON\_L3\_CTR\_CTL5, MSR\_EMON\_L3\_CTR\_CTL6, and MSR\_EMON\_L3\_CTR\_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.



### 18.6.7.2 GBSQ Event Interface

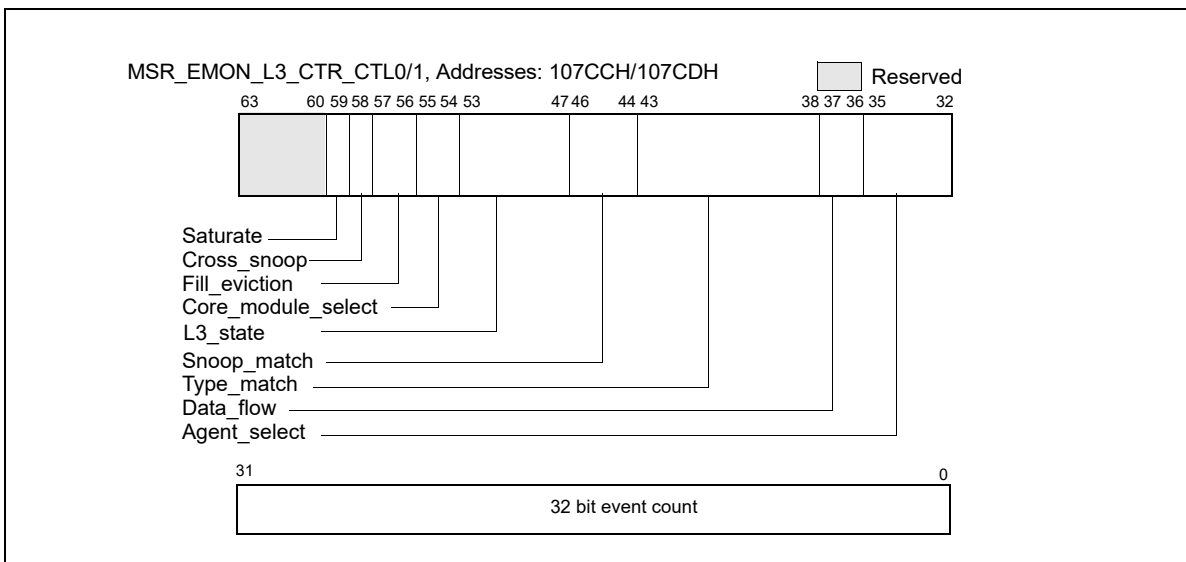
The layout of MSR\_EMON\_L3\_CTR\_CTL0 and MSR\_EMON\_L3\_CTR\_CTL1 is given in Figure 18-58. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- Agent\_Select (bits 35:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.

For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits corresponds to two logical processors in the first processor core, the upper two bits corresponds to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.

For Intel Xeon processor 7400 series, each bit of [34:32] specifies the SDI logic of a dual-core module as the originator of the transaction. A value of 0111B in bits [35:32] specifies transaction from any processor core.



**Figure 18-58. MSR\_EMON\_L3\_CTR\_CTL0/1, Addresses: 107CCH/107CDH**

- Data\_Flow (bits 37:36): Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.
- Type\_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include all transaction types.
- Snoop\_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L3\_State (bits 53:47): Each bit specifies an L2 coherency state.
- Core\_Module\_Select (bits 55:54): The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.

For Intel Xeon processor 7100 series,

- 00B: Match transactions from any core in the physical package
- 01B: Match transactions from this core only
- 10B: Match transactions from the other core in the physical package
- 11B: Match transaction from both cores in the physical package

For Intel Xeon processor 7400 series,

- 00B: Match transactions from any dual-core module in the physical package
- 01B: Match transactions from this dual-core module only
- 10B: Match transactions from either one of the other two dual-core modules in the physical package

- 11B: Match transaction from more than one dual-core modules in the physical package
- Fill\_Eviction (bits 57:56): The valid encodings are
  - 00B: Match any transactions
  - 01B: Match transactions that fill L3
  - 10B: Match transactions that fill L3 without an eviction
  - 11B: Match transaction fill L3 with an eviction
- Cross\_Snoop (bit 58): The encodings are
  - 0B: Match any transactions
  - 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

### 18.6.7.3 GSNPQ Event Interface

The layout of MSR\_EMON\_L3\_CTR\_CTL2 and MSR\_EMON\_L3\_CTR\_CTL3 is given in Figure 18-59. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent\_Select (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
- For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.

For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.

- Type\_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.
- Snoop\_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L2\_State (bits 53:47): Each bit specifies an L3 coherency state.
- Core\_Module\_Select (bits 56:54): Bit 56 enables Core\_Module\_Select matching. If bit 56 is clear, Core\_Module\_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.

For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are

- 00B: Match transactions from only one core (irrespective which core) in the physical package
- 01B: Match transactions from this core and not the other core
- 10B: Match transactions from the other core in the physical package, but not this core
- 11B: Match transaction from both cores in the physical package

For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are

- 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
- 01B: Match transactions from one or more dual-core modules.
- 10B: Match transactions from two or more dual-core modules.
- 11B: Match transaction from all three dual-core modules in the physical package.

- Block\_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.

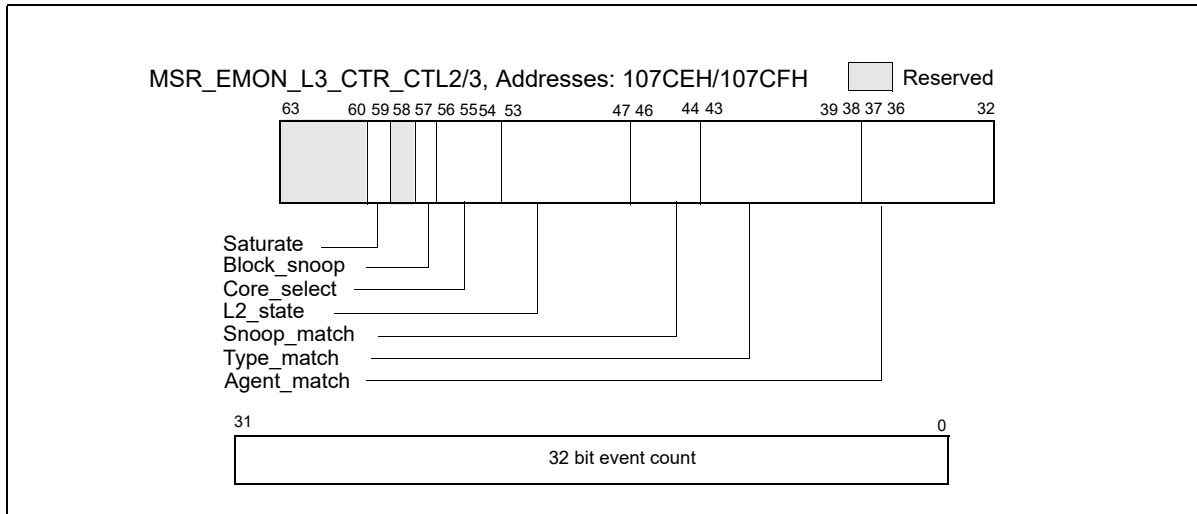


Figure 18-59. MSR\_EMON\_L3\_CTR\_CTL2/3, Addresses: 107CEH/107CFH

### 18.6.7.4 FSB Event Interface

The layout of MSR\_EMON\_L3\_CTR\_CTL4 through MSR\_EMON\_L3\_CTR\_CTL7 is given in Figure 18-60. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:

- Bit 58: must set to 1.
- FSB\_Submask (bits 57:32): Specifies FSB-specific sub-event mask.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

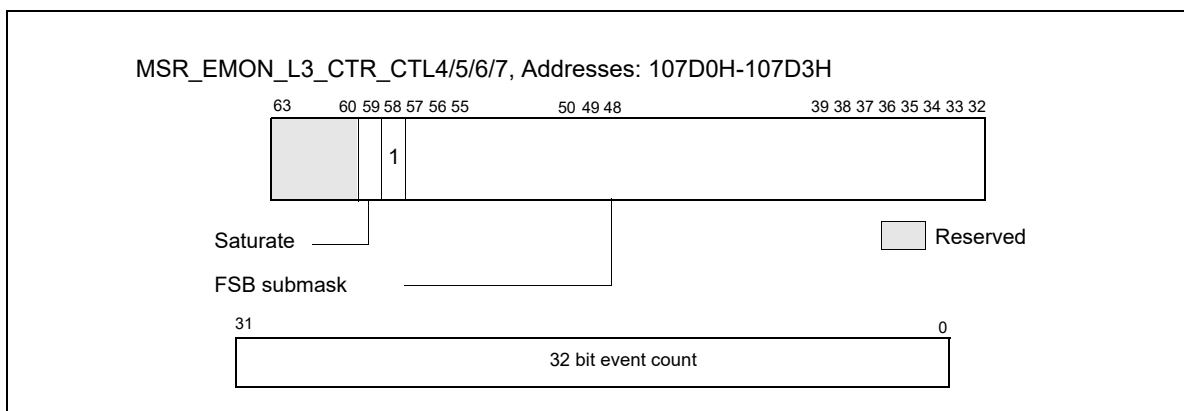


Figure 18-60. MSR\_EMON\_L3\_CTR\_CTL4/5/6/7, Addresses: 107D0H-107D3H

#### 18.6.7.4.1 FSB Sub-Event Mask Interface

- FSB\_type (bit 37:32): Specifies different FSB transaction types originated from this physical package.
- FSB\_L\_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package.
- FSB\_L\_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package.

- FSB\_L\_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package.
- FSB\_L\_defer (bit 41): Count DEFER responses to this processor's transactions.
- FSB\_L\_retry (bit 42): Count RETRY responses to this processor's transactions.
- FSB\_L\_snoop\_stall (bit 43): Count snoop stalls to this processor's transactions.
- FSB\_DBSY (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY).
- FSB\_DRDY (bit 45): Count DRDY assertions by this processor.
- FSB\_BNR (bit 46): Count BNR assertions by this processor.
- FSB\_IOQ\_empty (bit 47): Counts each bus clocks when the IOQ is empty.
- FSB\_IOQ\_full (bit 48): Counts each bus clocks when the IOQ is full.
- FSB\_IOQ\_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ.
- FSB\_WW\_data (bit 50): Counts back-to-back write transaction's data phase.
- FSB\_WW\_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
- FSB\_WR\_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
- FSB\_RW\_issue (bit 53): Counts back-to-back read-write transaction request pairs issued by this processor.
- FSB\_other\_DBSY (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY).
- FSB\_other\_DRDY (bit 55): Count DRDY assertions by another agent.
- FSB\_other\_snoop\_stall (bit 56): Count snoop stalls on the FSB due to another agent.
- FSB\_other\_BNR (bit 57): Count BNR assertions from another agent.

### 18.6.7.5 Common Event Control Interface

The MSR\_EMON\_L3\_GL\_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:

- GL\_freeze\_cmd (bit 0): Freeze the event counters specified by the GL\_event\_select field.
- GL\_unfreeze\_cmd (bit 1): Unfreeze the event counters specified by the GL\_event\_select field.
- GL\_reset\_cmd (bit 2): Clear the event count field of the event counters specified by the GL\_event\_select field. The event select field is not affected.
- GL\_event\_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR\_EMON\_L3\_CTR\_CTL0, bit 23 corresponds to MSR\_EMON\_L3\_CTR\_CTL7.
- GL\_event\_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR\_EMON\_L3\_CTR\_CTL0, bit 55 corresponds to MSR\_EMON\_L3\_CTR\_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 18-58 for example) is set, the event logic forces the value FFFF\_FFFFH into the event count field instead of incrementing it.

### 18.6.8 Performance Monitoring (P6 Family Processor)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

**NOTE**

The performance-monitoring events found at <https://perfmon-events.intel.com/> are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSRs: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

**NOTE**

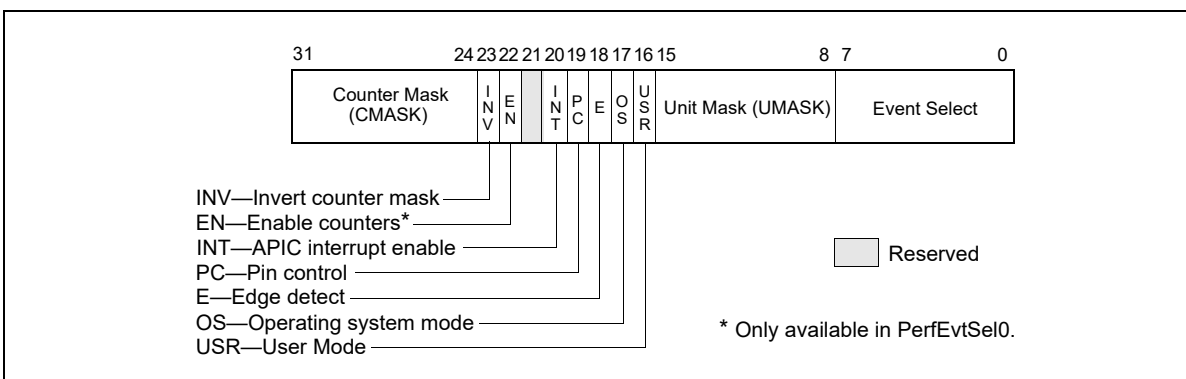
The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed for P6 family processors are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

**18.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs**

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 18-61 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7)** — Selects the event logic unit to detect certain microarchitectural conditions.
- **Unit mask (UMASK) field (bits 8 through 15)** — Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states.



**Figure 18-61. PerfEvtSel0 and PerfEvtSel1 MSRs**

- **USR (user mode) flag (bit 16)** — Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17)** — Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19)** — When set, the processor toggles the PM<sub>i</sub> pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM<sub>i</sub> pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20)** — When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22)** — This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31)** — When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

### 18.6.8.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

### 18.6.8.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

### 18.6.8.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking.
- Initialize and start counters.
- Stop counters.
- Read the event counters.
- Read the time-stamp counter.

The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 18.6.8.3, “Starting and Stopping the Performance-Monitoring Counters”).

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

### 18.6.8.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

## 18.6.9 Performance Monitoring (Pentium Processors)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0.

Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

### NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed for Pentium processors are model-specific for the Pentium processor.

The performance-monitoring events found at <https://perfmon-events.intel.com/> are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

### 18.6.9.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 18-62). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21)** — Selects (by entering an event code in the field) up to two events to be monitored.

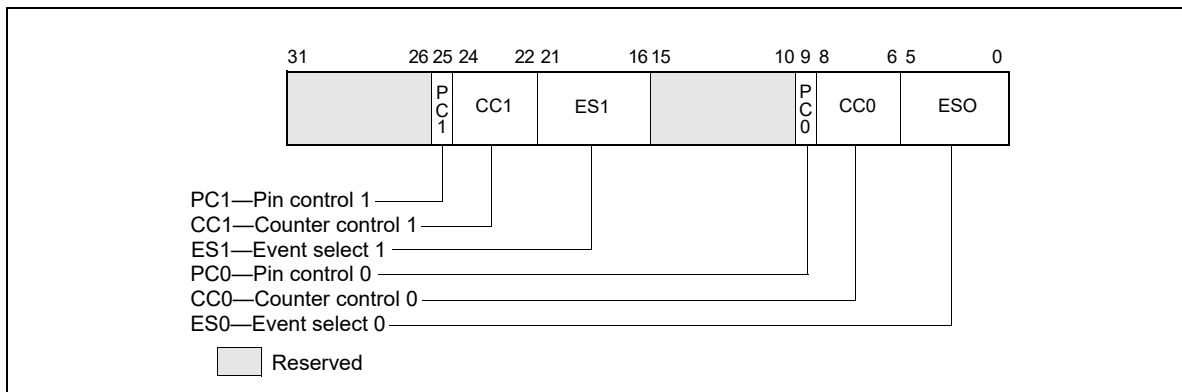


Figure 18-62. CESR MSR (Pentium Processor Only)

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24)** — Controls the operation of the counter. Control codes are as follows:

- 000 — Count nothing (counter disabled).
- 001 — Count the selected event while CPL is 0, 1, or 2.
- 010 — Count the selected event while CPL is 3.
- 011 — Count the selected event regardless of CPL.
- 100 — Count nothing (counter disabled).
- 101 — Count clocks (duration) while CPL is 0, 1, or 2.
- 110 — Count clocks (duration) while CPL is 3.
- 111 — Count clocks (duration) regardless of CPL.

The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25)** — Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the overflow or incremented condition. The external signaling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

### 18.6.9.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an “occurrence event” is being counted, the associated pin is asserted (high) each time the event occurs. When a “duration event” is being counted, the associated PM pin is asserted for the



entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single external clock may correspond to multiple internal clocks.

A “count up to” function may be provided when the event pin is programmed to signal an overflow of the counter. Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific value less than  $2^{40} - 1$ . After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

### 18.6.9.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

## 18.7 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor’s bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions.
- Enhanced Intel SpeedStep Technology transitions (P-state transitions).

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 18.7.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock** — The frequency of this clock is the frequency of the processor when the processor is not in turbo mode, and not being throttled via Intel SpeedStep.
- **Maximum Clock** — This is the maximum frequency of the processor when turbo mode is at the highest point.
- **Bus Clock** — These clockticks increment at a fixed frequency and help coordinate the bus on some systems.

- **Core Crystal Clock** — This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.
- **Non-halted Clockticks** — Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep Clockticks** — Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp Counter** — See Section 17.17, “Time-Stamp Counter”.
- **Reference Clockticks** — TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 17.17, “Time-Stamp Counter,” for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI** — Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI** — Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

### 18.7.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and umask 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

### 18.7.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 14, “Power and Thermal Management”), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32\_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32\_FIXED\_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32\_MPERF, and IA32\_FIXED\_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

## 18.7.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-8 “Information Returned by CPUID Instruction”), the nominal TSC frequency can be determined by using the following equation:

$$\text{Nominal TSC frequency} = (\text{CPUID.15H.ECX}[31:0] * \text{CPUID.15H.EBX}[31:0]) \div \text{CPUID.15H.EAX}[31:0]$$

For Intel processors in which CPUID.15H.EBX[31:0]  $\div$  CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 18-85 can be used to look up the nominal core crystal clock frequency.

**Table 18-85. Nominal Core Crystal Clock Frequency**

Processor Families/Processor Number Series <sup>1</sup>	Nominal Core Crystal Clock Frequency
Intel® Xeon® Processor Scalable Family with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors and Intel® Xeon® W Processor Family.	24 MHz
Next Generation Intel® Atom™ processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).	19.2 MHz

### NOTES:

- For any processor in which CPUID.15H is enumerated and MSR\_PLATFORM\_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

### 18.7.3.1 For Intel® Processors Based on Microarchitecture Code Name Sandy Bridge, Ivy Bridge, Haswell and Broadwell

The scalable bus frequency is encoded in the bit field MSR\_PLATFORM\_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

### 18.7.3.2 For Intel® Processors Based on Microarchitecture Code Name Nehalem

The scalable bus frequency is encoded in the bit field MSR\_PLATFORM\_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

### 18.7.3.3 For Intel® Atom™ Processors Based on the Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The scalable bus frequency is encoded in the bit field MSR\_PLATFORM\_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR\_FSB\_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR\_FSB\_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

### 18.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR\_FSB\_FREQ[2:0] at (0CDH), see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR\_PLATFORM\_ID[12:8]. It corresponds to the Processor Base frequency.
- If XE operation is enabled, the maximum resolved bus ratio is given in MSR\_PERF\_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR\_PERF\_STATUS[31] is set, XE operation is enabled. The MSR\_PERF\_STATUS[31] field is read-only.

## 18.8 IA32\_PERF\_CAPABILITIES MSR ENUMERATION

The layout of IA32\_PERF\_CAPABILITIES MSR is shown in Figure 18-63; it provides enumeration of a variety of interfaces:

- IA32\_PERF\_CAPABILITIES.LBR\_FMT[bits 5:0]: encodes the LBR format, details are described in Section 17.4.8.1.
- IA32\_PERF\_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist; see Section 18.6.2.4.2.
- IA32\_PERF\_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers; see Section 18.6.2.4.2.
- IA32\_PERF\_CAPABILITIES.PEBS\_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records; see Section 18.6.2.4.2.
- IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[12]: Indicates IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM is supported if 1, see Section 18.8.1.
- IA32\_PERF\_CAPABILITIES.FULL\_WRITE[13]: Indicates the processor supports IA32\_A\_PMCx interface for updating bits 32 and above of IA32\_PMCx; see Section 18.2.6.
- IA32\_PERF\_CAPABILITIES.PEBS\_BASELINE [bit 14]: If set, the following is true:
  - The IA32\_PEBS\_ENABLE MSR (address 3F1H) exists and all architecturally enumerated fixed and general-purpose counters have corresponding bits in IA32\_PEBS\_ENABLE that enable generation of PEBS records. The general-purpose counter bits start at bit IA32\_PEBS\_ENABLE[0], and the fixed counter bits start at bit IA32\_PEBS\_ENABLE[32].
  - The format of the PEBS record is enumerated by IA32\_PERF\_CAPABILITIES.PEBS\_FMT; see Section 18.6.2.4.2.
  - Extended PEBS is supported. All counters support the PEBS facility, and all events (both precise and non-precise) can generate PEBS records when PEBS is enabled for that counter. Note that not all events may be available on all counters.
  - Adaptive PEBS is supported. The PEBS\_DATA\_CFG MSR (address 3F2H) and adaptive record enable bits (IA32\_PERFEVTSELx.Adaptive\_Record and IA32\_FIXED\_CTR\_CTRL.FCx\_Adaptive\_Record) are supported. The definition of the PEBS\_DATA\_CFG MSR, including which bits are supported and how they affect the record, is enumerated by IA32\_PERF\_CAPABILITIES.PEBS\_FMT; see Section 18.9.2.3.
- IA32\_PERF\_CAPABILITIES.PERF\_METRICS\_AVAILABLE[15]: If set, indicates that the architecture provides built in support for TMA L1 metrics through the PERF\_METRICS MSR, see Section 18.3.9.3.
- IA32\_PERF\_CAPABILITIES.PEBS\_OUTPUT\_PT\_AVAIL[16]: If set on parts that enumerate support for Intel PT (CPUID.0x7.0.EBX[25]=1), setting IA32\_PEBS\_ENABLE.PEBS\_OUTPUT to 01B will result in PEBS output being written into the Intel PT trace stream. See Section 18.5.5.2.

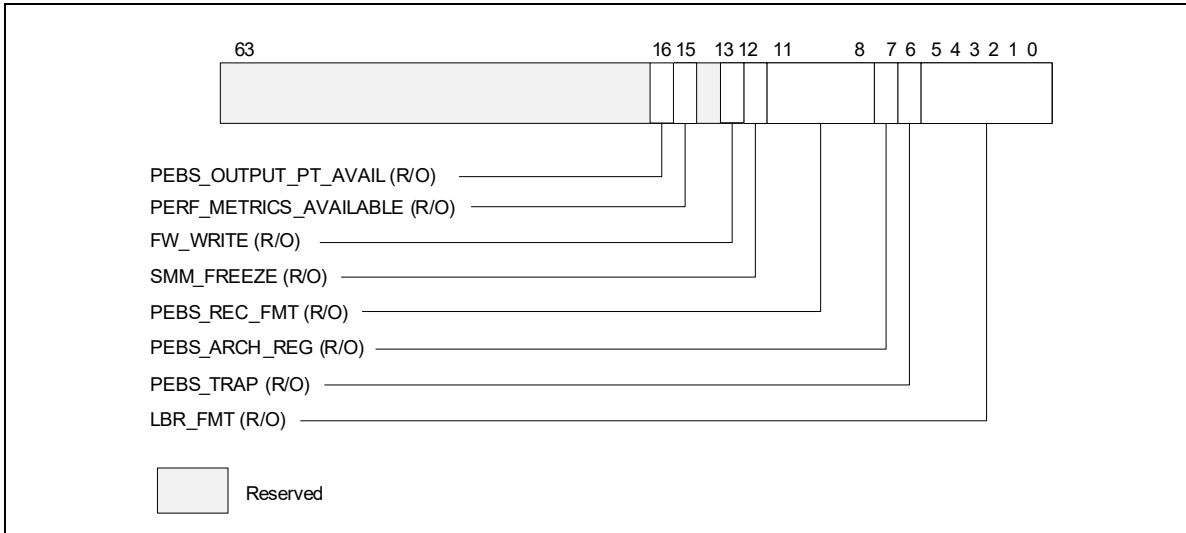


Figure 18-63. Layout of IA32\_PERF\_CAPABILITIES MSR

### 18.8.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 17.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)”) are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32\_PERF\_CAPABILITIES MSR. If IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE\_WHILE\_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32\_PERF\_GLOBAL\_CTRL, save a copy of the content of IA32\_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32\_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32\_PERF\_GLOBAL\_CTRL will be set to 1, the saved copy of IA32\_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM[bit 14] to 1 only supported as indicated by IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] reporting 1.

## 18.9 PEBS FACILITY

### 18.9.1 Extended PEBS

- The Extended PEBS feature supports Processor Event Based Sampling (PEBS) on all counters, both fixed function and general purpose; and all performance monitoring events, both precise and non-precise. PEBS can be enabled for the general purpose counters using PEBS\_EN\_PMCi bits of IA32\_PEBS\_ENABLE (i = 0, 1,...n). PEBS can be enabled for 'i' fixed function counters using the PEBS\_EN\_FIXEDi bits of IA32\_PEBS\_ENABLE (i = 0, 1, ...m).

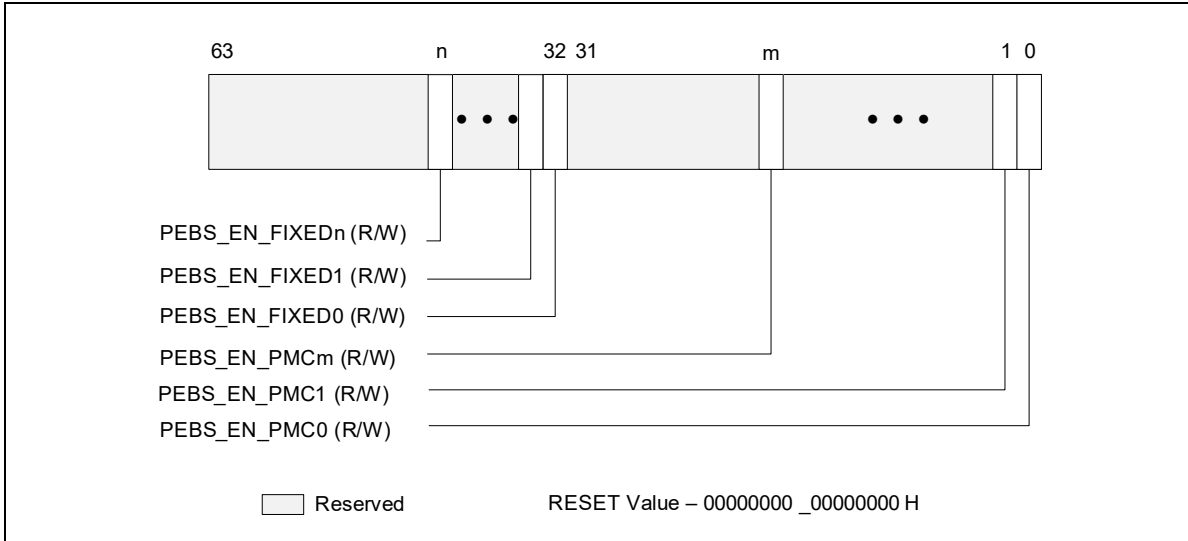


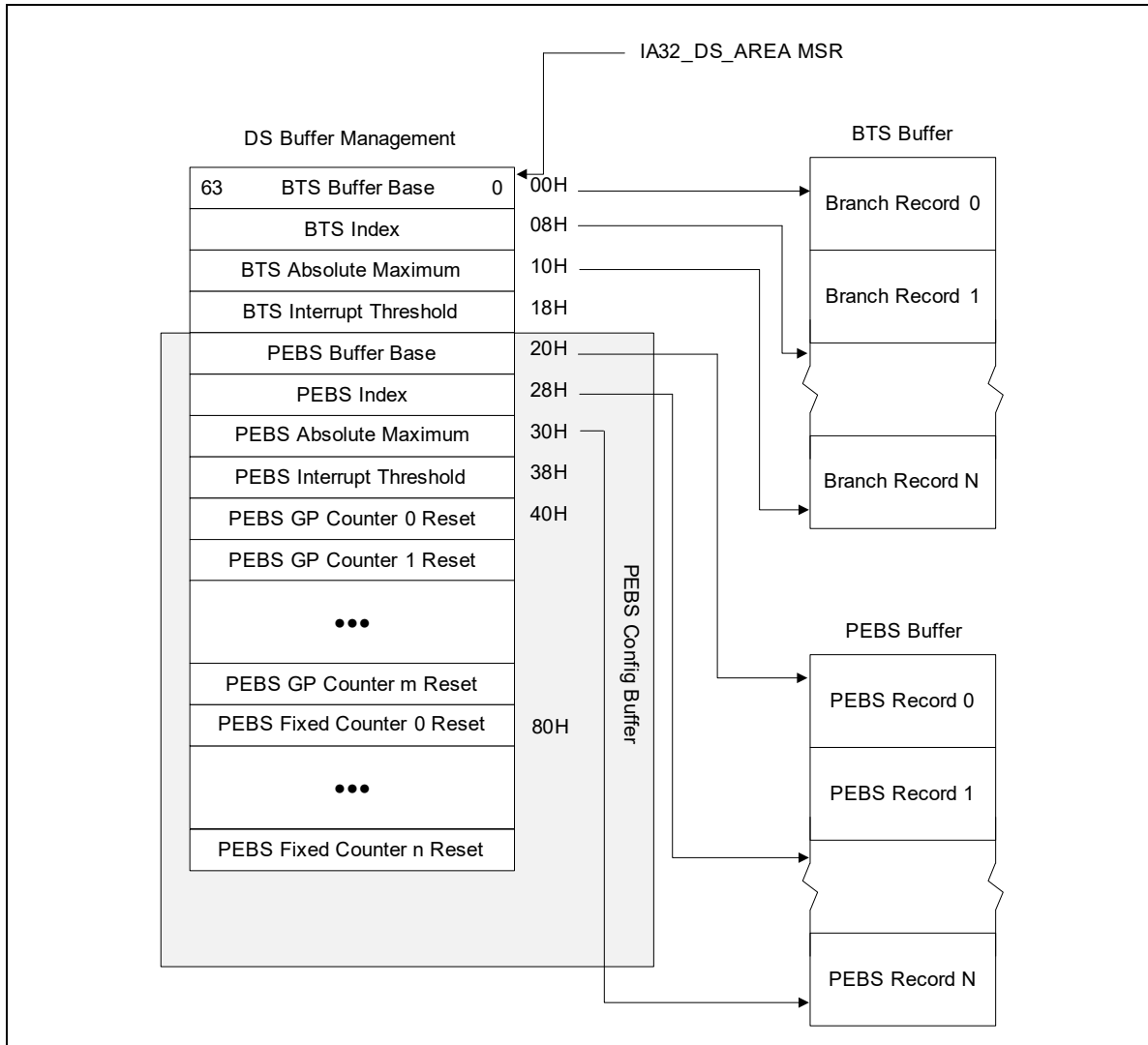
Figure 18-64. Layout of IA32\_PEBS\_ENABLE MSR

A PEBS record due to a precise event will be generated after an instruction that causes the event when the counter has already overflowed. A PEBS record due to a non-precise event will occur at the next opportunity after the counter has overflowed, including immediately after an overflow is set by an MSR write.

Currently, IA32\_FIXED\_CTR0 counts instructions retired and is a precise event. IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2 ... IA32\_FIXED\_CTRm count as non-precise events.

The Applicable Counter field in the Basic Info Group of the PEBS record indicates which counters caused the PEBS record to be generated. It is in the same format as the enable bits for each counter in IA32\_PEBS\_ENABLE. As an example, an Applicable Counter field with bits 2 and 32 set would indicate that both general purpose counter 2 and fixed function counter 0 generated the PEBS record.

- To properly use PEBS for the additional counters, software will need to set up the counter reset values in PEBS portion of the DS\_BUFFER\_MANAGEMENT\_AREA data structure that is indicated by the IA32\_DS\_AREA register. The layout of the DS\_BUFFER\_MANAGEMENT\_AREA is shown in Figure 18-65. When a counter generates a PEBS records, the appropriate counter reset values will be loaded into that counter. In the above example where general purpose counter 2 and fixed function counter 0 generated the PEBS record, general purpose counter 2 would be reloaded with the value contained in PEBS GP Counter 2 Reset (offset 50H) and fixed function counter 0 would be reloaded with the value contained in PEBS Fixed Counter 0 Reset (offset 80H).



**Figure 18-65. PEBS Programming Environment**

Extended PEBS support debuts on Intel® Atom processors based on the Goldmont Plus microarchitecture and future Intel® Core™ processors based on the Ice Lake microarchitecture.

## 18.9.2 Adaptive PEBS

The PEBS facility has been enhanced to collect the following CPU state in addition to GPRs, EventingIP, TSC and memory access related information collected by legacy PEBS:

- XMM registers
- LBR records (TO/FROM/INFO)

The PEBS record is restructured where fields are grouped into Basic group, Memory group, GPR group, XMM group and LBR group. A new register MSR\_PEBS\_DATA\_CFG provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.

By default, the PEBS record will only contain the Basic group. Optionally, each counter can be configured to generate a PEBS records with the groups specified in MSR\_PEBS\_DATA\_CFG.

Details and examples for the Adaptive PEBS capability follow below.

### 18.9.2.1 Adaptive\_Record Counter Control

- IA32\_PERFEVTSELx.Adaptive\_Record[34]: If this bit is set and IA32\_PEBS\_ENABLE.PEBS\_EN\_PMCx is set for the corresponding GP counter, an overflow of PMCx results in generation of an adaptive PEBS record with state information based on the selections made in MSR\_PEBS\_DATA\_CFG. If this bit is not set, a basic record is generated.

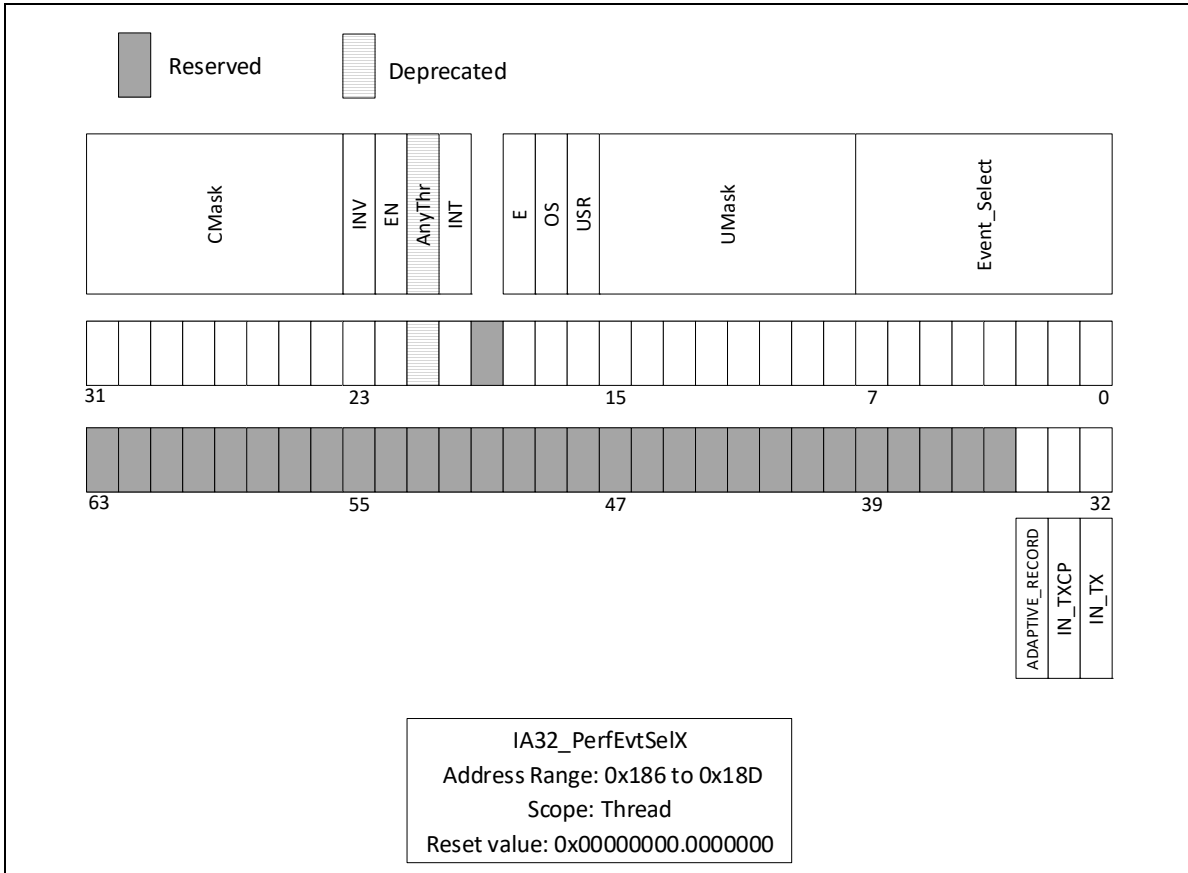


Figure 18-66. Layout of IA32\_PerfEvtSelX MSR Supporting Adaptive PEBS

- IA32\_FIXED\_CTR\_CTRL.FCx\_Adaptive\_Record: If this bit is set and IA32\_PEBS\_ENABLE.PEBS\_EN\_FIXEDx is set for the corresponding Fixed counter, an overflow of FixedCtrx results in generation of an adaptive PEBS record with state information based on the selections made in MSR\_PEBS\_DATA\_CFG. If this bit is not set, a basic record is generated.



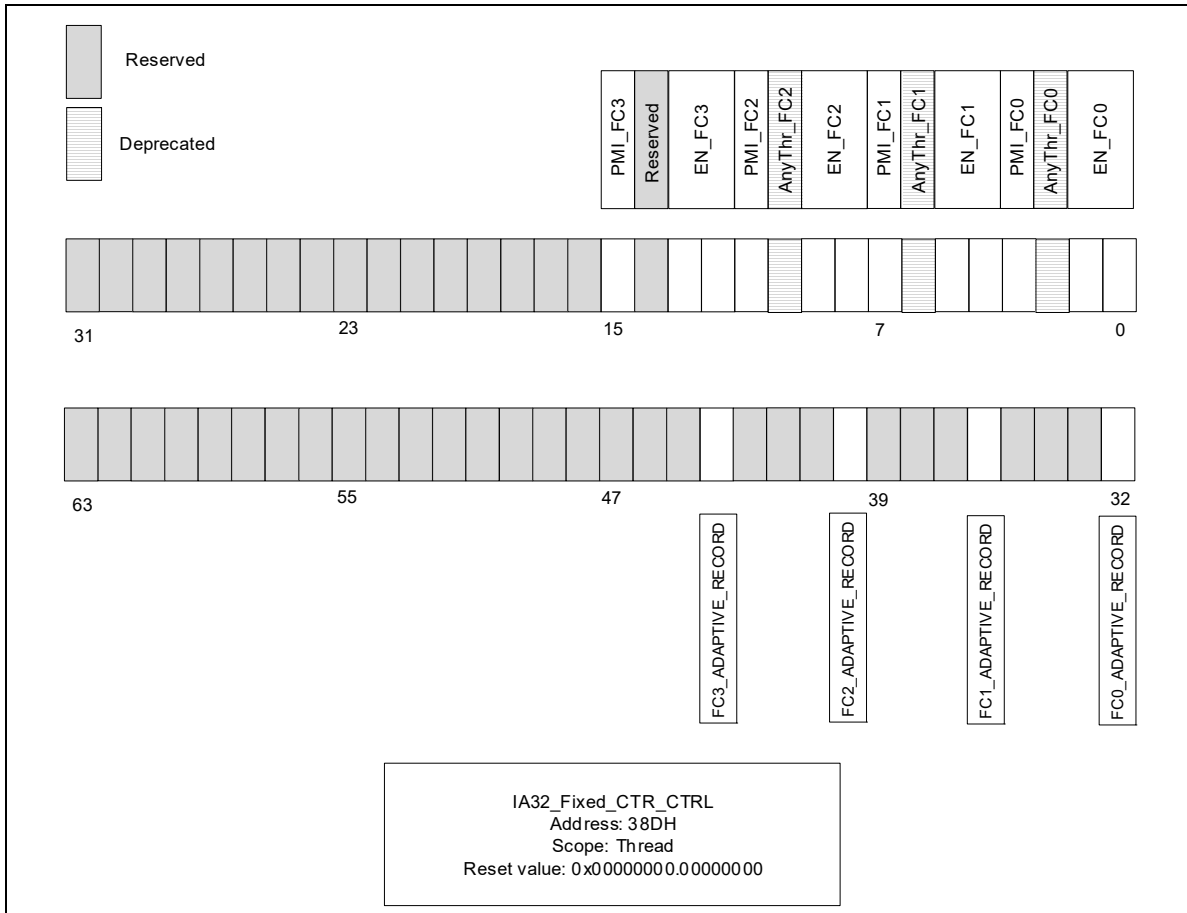


Figure 18-67. Layout of IA32\_FIXED\_CTR\_CTRL MSR Supporting Adaptive PEBS

### 18.9.2.2 PEBS Record Format

The data fields in the PEBS record are aggregated into five groups which are described in the sub-sections below. Processors that support Adaptive PEBS implement a new MSR called MSR\_PEBS\_DATA\_CFG which allows software to select the data groups to be captured. The data groups are not placed at fixed locations in the PEBS record, but are positioned immediately after one another, thus making the record format/size variable based on the groups selected.

#### 18.9.2.2.1 Basic Info

The Basic group contains essential information for software to parse a record along with several critical fields. It is always collected.

Table 18-86. Basic Info Group

Field Name	Bit Width	Description
Record Format	[47:0]	This field indicates which data groups are included in the record. The field is zero if none of the counters that triggered the current PEBS record have their Adaptive_Record bit set. Otherwise it contains the value of MSR_PEBS_DATA_CFG.
	[63:48]	This field provides the size of the current record in bytes. Selected groups are packed back-to-back in the record without gaps or padding for unselected groups.

**Table 18-86. Basic Info Group (Contd.)**

Instruction Pointer	[63:0]	This field reports the Eventing Instruction Pointer (EventingIP) of the retired instruction that triggered the PEBS record generation. Note that this field is different than R/EIP which records the instruction pointer of the next instruction to be executed after record generation. The legacy R/EIP field has been removed.
Applicable Counters	[63:0]	The Applicable Counters field indicates which counters triggered the generation of the PEBS record, linking the record to specific events. This allows software to correlate the PEBS record entry properly with the instruction that caused the event, even when multiple counters are configured to generate PEBS records and multiple bits are set in the field.
TSC	[63:0]	This field provides the time stamp counter value when the PEBS record was generated.

**18.9.2.2.2 Memory Access Info**

This group contains the legacy PEBS memory-related fields; see Section 18.3.1.1.2.

**Table 18-87. Memory Access Info Group**

Field Name	Bit Width	Description
Memory Access Address	[63:0]	This field contains the linear address of the source of the load, or linear address of the destination (target) of the store. This value is written as a 64-bit address in canonical form.
Memory Auxiliary Info	[63:0]	When MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains an encoded value indicating the memory hierarchy source which satisfied the load. These encodings are detailed in Table 18-4 and Table 18-13. If the PEBS assist was triggered for a store uop, this field will contain information indicating the status of the store, as detailed in Table 18-14.
Memory Access Latency	[63:0]	When MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains the latency to service the load in core clock cycles.
TSX Auxiliary Info	[31:0]	This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
	[63:31]	This field contains the abort details. Refer to Section 18.3.6.5.1.

**18.9.2.2.3 GPRs**

This group is captured when the GPR bit is enabled in MSR\_PEBS\_DATA\_CFG. GPRs are always 64 bits wide. If they are selected for non 64-bit mode, the upper 32-bit of the legacy RAX - RDI and all contents of R8-15 GPRs will be filled with 0s. In 64bit mode, the full 64 bit value of each register is written.

The order differs from legacy. The table below shows the order of the GPRs in Ice Lake microarchitecture.

**Table 18-88. GPRs in Ice Lake Microarchitecture**

Field Name	Bit Width
RFLAGS	[63:0]
RIP	[63:0]
RAX	[63:0]
RCX*	[63:0]

**Table 18-88. GPRs in Ice Lake Microarchitecture (Contd.)**

RDX*	[63:0]
RBX*	[63:0]
RSP*	[63:0]
RBP*	[63:0]
RSI*	[63:0]
RDI*	[63:0]
R8	[63:0]
...	...
R15	[63:0]

The machine state reported in the PEBS record is the committed machine state immediately after the instruction that triggers PEBS completes.

For instance, consider the following instruction sequence:

MOV eax, [eax]; triggers PEBS record generation

NOP

If the mov instruction triggers PEBS record generation, the EventingIP field in the PEBS record will report the address of the mov, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation. And the value of RIP will contain the linear address of the nop.

#### 18.9.2.2.4 XMMs

This group is captured when the XMM bit is enabled in MSR\_PEBS\_DATA\_CFG and SSE is enabled. If SSE is not enabled, the fields will contain zeroes. XMM8-XMM15 will also contain zeroes if not in 64-bit mode.

**Table 18-89. XMMs**

Field Name	Bit Width
XMM0	[127:0]
...	...
XMM15	[127:0]

#### 18.9.2.2.5 LBRs

To capture LBR data in the PEBS record, the LBR bit in MSR\_PEBS\_DATA\_CFG must be enabled. The number of LBR entries included in the record can be configured in the LBR\_entries field of MSR\_PEBS\_DATA\_CFG.

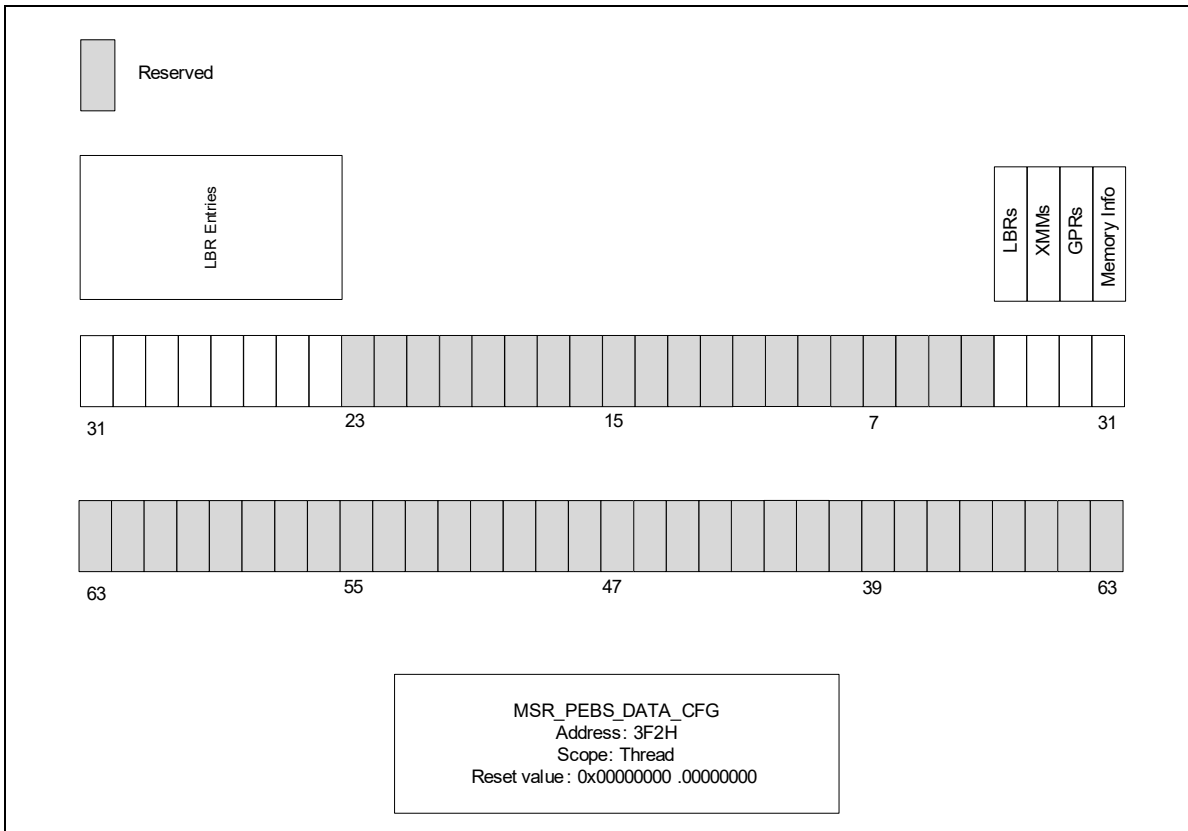
**Table 18-90. LBRs**

Field Name	Bit Width	Description
LBR[].FROM	[63:0]	Branch from address.
LBR[].TO	[63:0]	Branch to address.
LBR[].INFO	[63:0]	Other LBR information, like timing. This field is described in more detail in Section 17.12.1, "MSR_LBR_INFO_x MSR".

LBR entries are recorded into the record starting at LBR[TOS] and proceeding to LBR[TOS-1] and following. Note that LBR index is modulo the number of LBRs supporting on the processor.

**18.9.2.3 MSR\_PEBS\_DATA\_CFG**

Bits in MSR\_PEBS\_DATA\_CFG can be set to include data field blocks/groups into adaptive records. The Basic Info group is always included in the record. Additionally, the number of LBR entries included in the record is configurable.



**Figure 18-68. MSR\_PEBS\_DATA\_CFG**

**Table 18-91. MSR\_PEBS\_CFG Programming<sup>1</sup>**

Bit	Bit Index	Access	Description
Memory Info	0	R/W	Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.
GPRs	1	R/W	Setting this bit will capture the contents of the General Purpose registers in the PEBS record.
XMMs	2	R/W	Setting this bit will capture the contents of the XMM registers in the PEBS record.
LBRs	3	R/W	Setting this bit will capture LBR TO, FROM and INFO in the PEBS record.
Reserved <sup>2</sup>	23:4	NA	Reserved
LBR Entries	31:24	R/W	Set the field to the desired number of entries minus 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, it is recommended to select an even number of LBR entries (programmed into LBR_entries as an odd number).

**NOTES:**

1. A write to the MSR will be ignored when IA32\_MISC\_ENABLE.PERFMON\_AVAILABLE is zero (default).
2. Writing to the reserved bits will cause a GP fault.

**18.9.2.4 PEBS Record Examples**

The following example shows the layout of the PEBS record when all data groups are selected (all valid bits in MSR\_PEBS\_DATA\_CFG are set) and maximum number of LBRs are selected. There are no gaps in the PEBS record when a subset of the groups are selected, thus keeping the layout compact. Implementations that do not support some features will have to pad zeroes in the corresponding fields.

**Table 18-92. PEBS Record Example 1**

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	
0x20	Memory Info	Memory Access Address	DLA
0x28		Memory Auxiliary Info	DATA_SRC
0x30		Memory Access Latency	Load Latency
0x38		TSX Auxiliary Info	HLE Information

**Table 18-92. PEBS Record Example 1**

0x40	GPRs	RFLAGS	
0x48		RIP	
0x50		RAX	
...		...	
0x88		RDI	
0x90		R8	
...		...	
0xC8		R15	
0xD0	XMMs	XMM0	New
...		...	
0x1C0		XMM15	
0x1D0	LBRs	LBR[TOS].FROM	New
0x1D8		LBR[TOS].TO	
0x1E0		LBR[TOS].INFO	
...		...	
0x4B8		LBR[TOS + 1].FROM	
0x4C0		LBR[TOS + 1].TO	
0x4C8		LBR[TOS + 1].INFO	

The following example shows the layout of the PEBS record when Basic, GPR, and LBR group with 3 LBR entries are selected.

**Table 18-93. PEBS Record Example 2**

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	

Table 18-93. PEBS Record Example 2

0x20	GPRs	RFLAGS	
0x28		RIP	
0x30		RAX	
...		...	
0x68		RDI	
0x70		R8	
...		...	
0xA8		R15	
0xB0	LBRs	LBR[TOS].FROM	New
0xB8		LBR[TOS].TO	
0xC0		LBR[TOS].INFO	
...		...	
0xE0		LBR[TOS + 1].FROM	
0xE8		LBR[TOS + 1].TO	
0xF0		LBR[TOS + 1].INFO	

### 18.9.3 Precise Distribution of Instructions Retired (PDIR) Facility

Precise Distribution of Instructions Retired Facility is available via PEBS on some microarchitectures. Refer to Section 18.3.4.4.4. Counters that support PDIR also vary. See the processor specific sections for availability.

### 18.9.4 Reduced Skid PEBS

Processors based on Goldmont Plus microarchitecture support the Reduced Skid PEBS feature described in Section 18.5.3.1.2 on the IA32\_PMC0 counter. Although Extended PEBS adds support for generating PEBS records for precise events on additional general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

### 18.9.5 EPT-Friendly PEBS

The 3rd generation Intel Xeon Scalable Family of processors based on Ice Lake microarchitecture (and later processors) support VMX guest use of PEBS when the DS Area (including the PEBS Buffer and DS Management Area) is allocated from a paged pool of EPT pages. In such a configuration PEBS DS Area accesses may result in VM exits (e.g., EPT violations due to "lazy" EPT page-table entry propagation), and in such cases the PEBS record will not be lost but instead will "skid" to after the subsequent VM Entry back to the guest. For precise events the guest will observe that the record skid by one event occurrence, while for non-precise events the record will skid by one instruction.





## 22. Updates to Chapter 22, Volume 3C

Change bars and green text show changes to Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Changes to this chapter: Removed reference to chapter 31 which has been removed from the manual. The removed chapter was previously titled Chapter 31, "Virtual-Machine Monitor Programming Considerations".

### 22.1 OVERVIEW

This chapter describes the basics of virtual machine architecture and an overview of the virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environments.

Information about VMX instructions is provided in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*. Other aspects of VMX and system programming considerations are described in chapters of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

### 22.2 VIRTUAL MACHINE ARCHITECTURE

Virtual-machine extensions define processor-level support for virtual machines on IA-32 processors. Two principal classes of software are supported:

- **Virtual-machine monitors (VMM)** — A VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents guest software (see next paragraph) with an abstraction of a virtual processor and allows it to execute directly on a logical processor. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O.
- **Guest software** — Each virtual machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software. Each operates independently of other virtual machines and uses on the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The software stack acts as if it were running on a platform with no VMM. Software executing in a virtual machine must operate with reduced privilege so that the VMM can retain control of platform resources.

### 22.3 INTRODUCTION TO VMX OPERATION

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation. In general, a VMM will run in VMX root operation and guest software will run in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits.

Processor behavior in VMX root operation is very much as it is outside VMX operation. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited (see Section 22.8).

Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

There is no software-visible bit whose setting indicates whether a logical processor is in VMX non-root operation. This fact may allow a VMM to prevent guest software from determining that it is running in a virtual machine.

Because VMX operation places restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.

## 22.4 LIFE CYCLE OF VMM SOFTWARE

Figure 22-1 illustrates the life cycle of a VMM and its guest software as well as the interactions between them. The following items summarize that life cycle:

- Software enters VMX operation by executing a VMXON instruction.
- Using VM entries, a VMM can then enter guests into virtual machines (one at a time). The VMM effects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.
- VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the virtual machine using a VM entry.
- Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction.

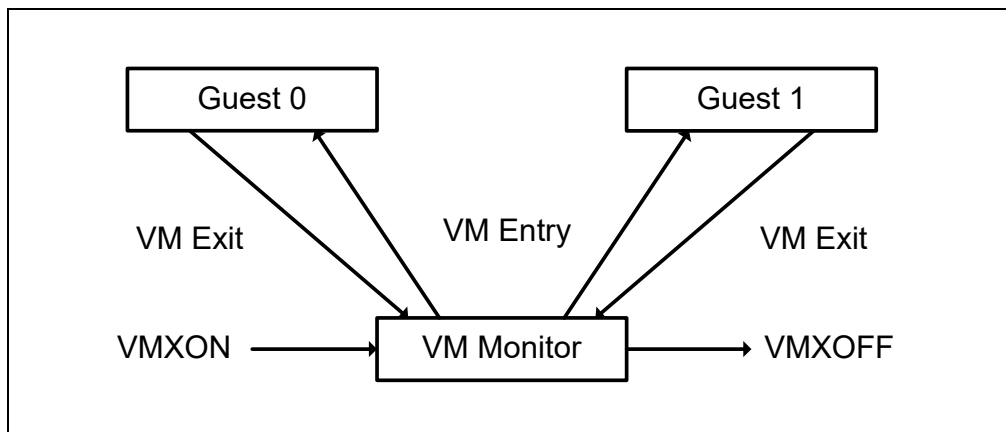


Figure 22-1. Interaction of a Virtual-Machine Monitor and Guests

## 22.5 VIRTUAL-MACHINE CONTROL STRUCTURE

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual-machine control structure (VMCS).

Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions.

A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor.

## 22.6 DISCOVERING SUPPORT FOR VMX

Before system software enters into VMX operation, it must discover the presence of VMX support in the processor. System software can determine whether a processor supports VMX operation using CPUID. If  $\text{CPUID.1:ECX.VMX}[\text{bit } 5] = 1$ , then VMX operation is supported. See Chapter 3, "Instruction Set Reference, A-L" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

The VMX architecture is designed to be extensible so that future processors in VMX operation can support additional features not present in first-generation implementations of the VMX architecture. The availability of extensible VMX features is reported to software using a set of VMX capability MSRs (see Appendix A, "VMX Capability Reporting Facility").

## 22.7 ENABLING AND ENTERING VMX OPERATION

Before system software can enter VMX operation, it enables VMX by setting `CR4.VMXE[bit 13] = 1`. VMX operation is then entered by executing the `VMXON` instruction. `VMXON` causes an invalid-opcode exception (`#UD`) if executed with `CR4.VMXE = 0`. Once in VMX operation, it is not possible to clear `CR4.VMXE` (see Section 22.8). System software leaves VMX operation by executing the `VMXOFF` instruction. `CR4.VMXE` can be cleared outside of VMX operation after executing of `VMXOFF`.

`VMXON` is also controlled by the `IA32_FEATURE_CONTROL` MSR (MSR address 3AH). This MSR is cleared to zero when a logical processor is reset. The relevant bits of the MSR are:

- **Bit 0 is the lock bit.** If this bit is clear, `VMXON` causes a general-protection exception. If the lock bit is set, `WRMSR` to this MSR causes a general-protection exception; the MSR cannot be modified until a power-up reset condition. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX. To enable VMX support in a platform, BIOS must set bit 1, bit 2, or both (see below), as well as the lock bit.
- **Bit 1 enables VMXON in SMX operation.** If this bit is clear, execution of `VMXON` in SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support both VMX operation (see Section 22.6) and SMX operation (see Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*) cause general-protection exceptions.
- **Bit 2 enables VMXON outside SMX operation.** If this bit is clear, execution of `VMXON` outside SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support VMX operation (see Section 22.6) cause general-protection exceptions.

### NOTE

A logical processor is in SMX operation if `GETSEC[SEXIT]` has not been executed since the last execution of `GETSEC[SENDER]`. A logical processor is outside SMX operation if `GETSEC[SENDER]` has not been executed or if `GETSEC[SEXIT]` was executed after the last execution of `GETSEC[SENDER]`. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*.

Before executing `VMXON`, software should allocate a naturally aligned 4-KByte region of memory that a logical processor may use to support VMX operation.<sup>1</sup> This region is called the **VMXON region**. The address of the VMXON region (the VMXON pointer) is provided in an operand to `VMXON`. Section 23.11.5, “VMXON Region,” details how software should initialize and access the VMXON region.

## 22.8 RESTRICTIONS ON VMX OPERATION

VMX operation places restrictions on processor operation. These are detailed below:

- In VMX operation, processors may fix certain bits in `CR0` and `CR4` to specific values and not support other values. `VMXON` fails if any of these bits contains an unsupported value (see “VMXON—Enter VMX Operation” in Chapter 29). Any attempt to set one of these bits to an unsupported value while in VMX operation (including VMX root operation) using any of the `CLTS`, `LMSW`, or `MOV CR` instructions causes a general-protection exception. VM entry or VM exit cannot set any of these bits to an unsupported value. Software should consult the VMX capability MSRs `IA32_VMX_CR0_FIXED0` and `IA32_VMX_CR0_FIXED1` to determine how bits in `CR0` are fixed (see Appendix A.7). For `CR4`, software should consult the VMX capability MSRs `IA32_VMX_CR4_FIXED0` and `IA32_VMX_CR4_FIXED1` (see Appendix A.8).

### NOTES

The first processors to support VMX operation require that the following bits be 1 in VMX operation: `CR0.PE`, `CR0.NE`, `CR0.PG`, and `CR4.VMXE`. The restrictions on `CR0.PE` and `CR0.PG` imply that VMX

1. Future processors may require that a different amount of memory be reserved. If so, this fact is reported to software using the VMX capability-reporting mechanism.

operation is supported only in paged protected mode (including IA-32e mode). Therefore, guest software cannot be run in unpagged protected mode or in real-address mode.

Later processors support a VM-execution control called “unrestricted guest” (see Section 23.6.2). If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation (even if the capability MSR IA32\_VMX\_CR0\_FIXED0 reports otherwise).<sup>1</sup> Such processors allow guest software to run in unpagged protected mode or in real-address mode.

- VMXON fails if a logical processor is in A20M mode (see “VMXON—Enter VMX Operation” in Chapter 29). Once the processor is in VMX operation, A20M interrupts are blocked. Thus, it is impossible to be in A20M mode in VMX operation.
- The INIT signal is blocked whenever a logical processor is in VMX root operation. It is not blocked in VMX non-root operation. Instead, INITs cause VM exits (see Section 24.2, “Other Causes of VM Exits”).
- Intel<sup>®</sup> Processor Trace (Intel PT) can be used in VMX operation only if IA32\_VMX\_MISC[14] is read as 1 (see Appendix A.6). On processors that support Intel PT but which do not allow it to be used in VMX operation, execution of VMXON clears IA32\_RTIT\_CTL.TraceEn (see “VMXON—Enter VMX Operation” in Chapter 29); any attempt to write IA32\_RTIT\_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.

---

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “unrestricted guest” VM-execution control were 0. See Section 23.6.2.

### 23. Updates to Chapter 27, Volume 3C

Change bars and green text show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Changes to this chapter: Removed reference to chapter 32 which has been removed from the manual. The removed chapter was previously titled Chapter 32, "Virtualization of System Resources".

# CHAPTER 27

## VMX SUPPORT FOR ADDRESS TRANSLATION

---

The architecture for VMX operation includes two features that support address translation: virtual-processor identifiers (VPIDs) and the extended page-table mechanism (EPT). VPIDs are a mechanism for managing translations of linear addresses. EPT defines a layer of address translation that augments the translation of linear addresses.

Section 27.1 details the architecture of VPIDs. Section 27.2 provides the details of EPT. Section 27.3 explains how a logical processor may cache information from the paging structures, how it may use that cached information, and how software can managed the cached information.

### 27.1 VIRTUAL PROCESSOR IDENTIFIERS (VPIDS)

The original architecture for VMX operation required VMX transitions to flush the TLBs and paging-structure caches. This ensured that translations cached for the old linear-address space would not be used after the transition.

Virtual-processor identifiers (**VPIDs**) introduce to VMX operation a facility by which a logical processor may cache information for multiple linear-address spaces. When VPIDs are used, VMX transitions may retain cached information and the logical processor switches to a different linear-address space.

Section 27.3 details the mechanisms by which a logical processor manages information cached for multiple address spaces. A logical processor may tag some cached information with a 16-bit VPID. This section specifies how the current VPID is determined at any point in time:

- The current VPID is 0000H in the following situations:
  - Outside VMX operation. (This includes operation in system-management mode under the default treatment of SMIs and SMM with VMX operation; see Section 30.14.)
  - In VMX root operation.
  - In VMX non-root operation when the “enable VPID” VM-execution control is 0.
- If the logical processor is in VMX non-root operation and the “enable VPID” VM-execution control is 1, the current VPID is the value of the VPID VM-execution control field in the VMCS. (VM entry ensures that this value is never 0000H; see Section 25.2.1.1.)

VPIDs and PCIDs (see Section 4.10.1) can be used concurrently. When this is done, the processor associates cached information with both a VPID and a PCID. Such information is used only if the current VPID and PCID **both** match those associated with the cached information.

### 27.2 THE EXTENDED PAGE TABLE MECHANISM (EPT)

The extended page-table mechanism (**EPT**) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain addresses that would normally be treated as physical addresses (and used to access memory) are instead treated as **guest-physical addresses**. Guest-physical addresses are translated by traversing a set of **EPT paging structures** to produce physical addresses that are used to access memory.

- Section 27.2.1 gives an overview of EPT.
- Section 27.2.2 describes operation of EPT-based address translation.
- Section 27.2.3 discusses VM exits that may be caused by EPT.
- Section 27.2.7 describes interactions between EPT and memory typing.

#### 27.2.1 EPT Overview

EPT is used when the “enable EPT” VM-execution control is 1.<sup>1</sup> It translates the guest-physical addresses used in VMX non-root operation and those used by VM entry for event injection.

The translation from guest-physical addresses to physical addresses is determined by a set of **EPT paging structures**. The EPT paging structures are similar to those used to translate linear addresses while the processor is in IA-32e mode. Section 27.2.2 gives the details of the EPT paging structures.

If  $CR0.PG = 1$ , linear addresses are translated through paging structures referenced through control register CR3. While the “enable EPT” VM-execution control is 1, these are called **guest paging structures**. There are no guest paging structures if  $CR0.PG = 0$ .<sup>1</sup>

When the “enable EPT” VM-execution control is 1, the identity of **guest-physical addresses** depends on the value of  $CR0.PG$ :

- If  $CR0.PG = 0$ , each linear address is treated as a guest-physical address.
- If  $CR0.PG = 1$ , guest-physical addresses are those derived from the contents of control register CR3 and the guest paging structures. (This includes the values of the PDPTes, which logical processors store in internal, non-architectural registers.) The latter includes (in page-table entries and in other paging-structure entries for which bit 7—PS—is 1) the addresses to which linear addresses are translated by the guest paging structures.

If  $CR0.PG = 1$ , the translation of a linear address to a physical address requires multiple translations of guest-physical addresses using EPT. Assume, for example, that  $CR4.PAE = CR4.PSE = 0$ . The translation of a 32-bit linear address then operates as follows:

- Bits 31:22 of the linear address select an entry in the guest page directory located at the guest-physical address in CR3. The guest-physical address of the guest page-directory entry (PDE) is translated through EPT to determine the guest PDE’s physical address.
- Bits 21:12 of the linear address select an entry in the guest page table located at the guest-physical address in the guest PDE. The guest-physical address of the guest page-table entry (PTE) is translated through EPT to determine the guest PTE’s physical address.
- Bits 11:0 of the linear address is the offset in the page frame located at the guest-physical address in the guest PTE. The guest-physical address determined by this offset is translated through EPT to determine the physical address to which the original linear address translates.

In addition to translating a guest-physical address to a physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called **EPT violations** and cause VM exits. See Section 27.2.3.

A processor uses EPT to translate guest-physical addresses only when those addresses are used to access memory. This principle implies the following:

- The MOV to CR3 instruction loads CR3 with a guest-physical address. Whether that address is translated through EPT depends on whether PAE paging is being used.<sup>2</sup>
  - If PAE paging is not being used, the instruction does not use that address to access memory and does **not** cause it to be translated through EPT. (If  $CR0.PG = 1$ , the address will be translated through EPT on the next memory accessing using a linear address.)
  - If PAE paging is being used, the instruction loads the four (4) page-directory-pointer-table entries (PDPTes) from that address and it **does** cause the address to be translated through EPT.
- Section 4.4.1 identifies executions of MOV to CR0 and MOV to CR4 that load the PDPTes from the guest-physical address in CR3. Such executions cause that address to be translated through EPT.
- The PDPTes contain guest-physical addresses. The instructions that load the PDPTes (see above) do not use those addresses to access memory and do **not** cause them to be translated through EPT. The address in a PDPTE will be translated through EPT on the next memory accessing using a linear address that uses that PDPTE.

1. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, the logical processor operates as if the “enable EPT” VM-execution control were 0. See Section 23.6.2.

1. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that  $CR0.PG$  must be 1 in VMX operation,  $CR0.PG$  can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

2. A logical processor uses PAE paging if  $CR0.PG = 1$ ,  $CR4.PAE = 1$  and  $IA32_EFER.LMA = 0$ . See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.



## 27.2.2 EPT Translation Mechanism

The EPT translation mechanism uses only bits 47:0 of each guest-physical address.<sup>1</sup> It uses a page-walk length of 4, meaning that at most 4 EPT paging-structure entries are accessed to translate a guest-physical address.<sup>2</sup>

These 48 bits are partitioned by the logical processor to traverse the EPT paging structures:

- A 4-KByte naturally aligned EPT PML4 table is located at the physical address specified in bits 51:12 of the extended-page-table pointer (EPTP), a VM-execution control field (see Table 23-9 in Section 23.6.11). An EPT PML4 table comprises 512 64-bit entries (EPT PML4Es). An EPT PML4E is selected using the physical address defined as follows:
  - Bits 63:52 are all 0.
  - Bits 51:12 are from the EPTP.
  - Bits 11:3 are bits 47:39 of the guest-physical address.
  - Bits 2:0 are all 0.

Because an EPT PML4E is identified using bits 47:39 of the guest-physical address, it controls access to a 512-GByte region of the guest-physical-address space. The format of an EPT PML4E is given in Table 27-1.

**Table 27-1. Format of an EPT PML4 Entry (PML4E) that References an EPT Page-Directory-Pointer Table**

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 512-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 512-GByte region controlled by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 512-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 512-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 512-GByte region controlled by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 512-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page-directory-pointer table referenced by this entry <sup>1</sup>
51:N	Reserved (must be 0)
63:52	Ignored

1. No processors supporting the Intel 64 architecture support more than 48 physical-address bits. Thus, no such processor can produce a guest-physical address with more than 48 bits. An attempt to use such an address causes a page fault. An attempt to load CR3 with such an address causes a general-protection fault. If PAE paging is being used, an attempt to load CR3 that would load a PDPTe with such an address causes a general-protection fault.

2. Future processors may include support for other EPT page-walk lengths. Software should read the VMX capability MSR IA32\_VMX\_EPT\_VPID\_CAP (see Appendix A.10) to determine what EPT page-walk lengths are supported.

**NOTES:**

1. N is the physical-address width supported by the processor. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- A 4-KByte naturally aligned EPT page-directory-pointer table is located at the physical address specified in bits 51:12 of the EPT PML4E. An EPT page-directory-pointer table comprises 512 64-bit entries (EPT PDPTes). An EPT PDPTE is selected using the physical address defined as follows:
  - Bits 63:52 are all 0.
  - Bits 51:12 are from the EPT PML4E.
  - Bits 11:3 are bits 38:30 of the guest-physical address.
  - Bits 2:0 are all 0.

Because an EPT PDPTE is identified using bits 47:30 of the guest-physical address, it controls access to a 1-GByte region of the guest-physical-address space. Use of the EPT PDPTE depends on the value of bit 7 in that entry:<sup>1</sup>

- If bit 7 of the EPT PDPTE is 1, the EPT PDPTE maps a 1-GByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:30 are from the EPT PDPTE.
  - Bits 29:0 are from the original guest-physical address.

The format of an EPT PDPTE that maps a 1-GByte page is given in Table 27-2.

- If bit 7 of the EPT PDPTE is 0, a 4-KByte naturally aligned EPT page directory is located at the physical address specified in bits 51:12 of the EPT PDPTE. The format of an EPT PDPTE that references an EPT page directory is given in Table 27-3.

---

1. Not all processors allow bit 7 of an EPT PDPTE to be set to 1. Software should read the VMX capability MSR IA32\_VMX\_EPT\_VPID\_CAP (see Appendix A.10) to determine whether this is allowed.

**Table 27-2. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page**

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 1-GByte page referenced by this entry
1	Write access; indicates whether writes are allowed to the 1-GByte page referenced by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-GByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-GByte page controlled by this entry
5:3	EPT memory type for this 1-GByte page (see Section 27.2.7)
6	Ignore PAT memory type for this 1-GByte page (see Section 27.2.7)
7	Must be 1 (otherwise, this entry references an EPT page directory)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-GByte page referenced by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 1-GByte page referenced by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-GByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
29:12	Reserved (must be 0)
(N-1):30	Physical address of the 1-GByte page referenced by this entry <sup>1</sup>
51:N	Reserved (must be 0)
59:52	Ignored
60	Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 1-GByte page mapped by this entry (see Section 27.2.3.2). Ignored if bit 7 of EPTP is 0
62:61	Ignored
63	Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 24.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

**NOTES:**

1. N is the physical-address width supported by the logical processor.

**Table 27-3. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that References an EPT Page Directory**

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 1-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 1-GByte region controlled by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-GByte region controlled by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page directory referenced by this entry <sup>1</sup>
51:N	Reserved (must be 0)
63:52	Ignored

**NOTES:**

1. N is the physical-address width supported by the logical processor.

An EPT page-directory comprises 512 64-bit entries (PDEs). An EPT PDE is selected using the physical address defined as follows:

- Bits 63:52 are all 0.
- Bits 51:12 are from the EPT PDPTE.
- Bits 11:3 are bits 29:21 of the guest-physical address.
- Bits 2:0 are all 0.

Because an EPT PDE is identified using bits 47:21 of the guest-physical address, it controls access to a 2-MByte region of the guest-physical-address space. Use of the EPT PDE depends on the value of bit 7 in that entry:

- If bit 7 of the EPT PDE is 1, the EPT PDE maps a 2-MByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:21 are from the EPT PDE.
  - Bits 20:0 are from the original guest-physical address.

The format of an EPT PDE that maps a 2-MByte page is given in Table 27-4.

- If bit 7 of the EPT PDE is 0, a 4-KByte naturally aligned EPT page table is located at the physical address specified in bits 51:12 of the EPT PDE. The format of an EPT PDE that references an EPT page table is given in Table 27-5.

An EPT page table comprises 512 64-bit entries (PTEs). An EPT PTE is selected using a physical address defined as follows:

- Bits 63:52 are all 0.

**Table 27-4. Format of an EPT Page-Directory Entry (PDE) that Maps a 2-MByte Page**

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 2-MByte page referenced by this entry
1	Write access; indicates whether writes are allowed to the 2-MByte page referenced by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte page controlled by this entry
5:3	EPT memory type for this 2-MByte page (see Section 27.2.7)
6	Ignore PAT memory type for this 2-MByte page (see Section 27.2.7)
7	Must be 1 (otherwise, this entry references an EPT page table)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
20:12	Reserved (must be 0)
(N-1):21	Physical address of the 2-MByte page referenced by this entry <sup>1</sup>
51:N	Reserved (must be 0)
59:52	Ignored
60	Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 2-MByte page mapped by this entry (see Section 27.2.3.2). Ignored if bit 7 of EPTP is 0
62:61	Ignored
63	Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 24.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

**NOTES:**

1. N is the physical-address width supported by the logical processor.

- Bits 51:12 are from the EPT PDE.
- Bits 11:3 are bits 20:12 of the guest-physical address.
- Bits 2:0 are all 0.
- Because an EPT PTE is identified using bits 47:12 of the guest-physical address, every EPT PTE maps a 4-KByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:12 are from the EPT PTE.

- Bits 11:0 are from the original guest-physical address.

The format of an EPT PTE is given in Table 27-6.

An EPT paging-structure entry is **present** if any of bits 2:0 is 1; otherwise, the entry is **not present**. The processor ignores bits 62:3 and uses the entry neither to reference another EPT paging-structure entry nor to produce a physical address. A reference using a guest-physical address whose translation encounters an EPT paging-structure that is not present causes an EPT violation (see Section 27.2.3.2). (If the “EPT-violation #VE” VM-execution control is 1, the EPT violation is convertible to a virtualization exception only if bit 63 is 0; see Section 24.5.7.1. If the “EPT-violation #VE” VM-execution control is 0, this bit is ignored.)

**Table 27-5. Format of an EPT Page-Directory Entry (PDE) that References an EPT Page Table**

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 2-MByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 2-MByte region controlled by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte region controlled by this entry
6:3	Reserved (must be 0)
7	Must be 0 (otherwise, this entry maps a 2-MByte page)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte region controlled by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page table referenced by this entry <sup>1</sup>
51:N	Reserved (must be 0)
63:52	Ignored

**NOTES:**

1. N is the physical-address width supported by the logical processor.

**NOTE**

If the “mode-based execute control for EPT” VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 **or bit 10** is 1. If bits 2:0 are all 0 but bit 10 is 1, the entry is used normally to reference another EPT paging-structure entry or to produce a physical address.

The discussion above describes how the EPT paging structures reference each other and how the logical processor traverses those structures when translating a guest-physical address. It does not cover all details of the translation process. Additional details are provided as follows:

- Situations in which the translation process may lead to VM exits (sometimes before the process completes) are described in Section 27.2.3.
- Interactions between the EPT translation mechanism and memory typing are described in Section 27.2.7.

Figure 27-1 gives a summary of the formats of the EPTP and the EPT paging-structure entries. For the EPT paging structure entries, it identifies separately the format of entries that map pages, those that reference other EPT

paging structures, and those that do neither because they are not present; bits 2:0 and bit 7 are highlighted because they determine how a paging-structure entry is used. (Figure 27-1 does not comprehend the fact that, if the “mode-based execute control for EPT” VM-execution control is 1, an entry is present if any of bits 2:0 or bit 10 is 1.)

### 27.2.3 EPT-Induced VM Exits

Accesses using guest-physical addresses may cause VM exits due to EPT misconfigurations, EPT violations, and page-modification log-full events. An **EPT misconfiguration** occurs when, in the course of translating a guest-physical address, the logical processor encounters an EPT paging-structure entry that contains an unsupported value (see Section 27.2.3.1). An **EPT violation** occurs when there is no EPT misconfiguration but the EPT paging-structure entries disallow an access using the guest-physical address (see Section 27.2.3.2). A **page-modification log-full event** occurs when the logical processor determines a need to create a page-modification log entry and the current log is full (see Section 27.2.6).

These events occur only due to an attempt to access memory with a guest-physical address. Loading CR3 with a guest-physical address with the MOV to CR3 instruction can cause neither an EPT configuration nor an EPT violation until that address is used to access a paging structure.<sup>1</sup>

If the “EPT-violation #VE” VM-execution control is 1, certain EPT violations may cause virtualization exceptions instead of VM exits. See Section 24.5.7.1.

#### 27.2.3.1 EPT Misconfigurations

An EPT misconfiguration occurs if translation of a guest-physical address encounters an EPT paging-structure that meets any of the following conditions:

- Bit 0 of the entry is clear (indicating that data reads are not allowed) and bit 1 is set (indicating that data writes are allowed).
- Either of the following if the processor does not support execute-only translations:
  - Bit 0 of the entry is clear (indicating that data reads are not allowed) and bit 2 is set (indicating that instruction fetches are allowed).<sup>2</sup>
  - The “mode-based execute control for EPT” VM-execution control is 1, bit 0 of the entry is clear (indicating that data reads are not allowed), and bit 10 is set (indicating that instruction fetches are allowed from user-mode linear addresses).

Software should read the VMX capability MSR IA32\_VMX\_EPT\_VPID\_CAP to determine whether execute-only translations are supported (see Appendix A.10).

- The entry is present (see Section 27.2.2) and one of the following holds:
  - A reserved bit is set. This includes the setting of a bit in the range 51:12 that is beyond the logical processor’s physical-address width.<sup>3</sup> See Section 27.2.2 for details of which bits are reserved in which EPT paging-structure entries.
  - The entry is the last one used to translate a guest physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE) and the value of bits 5:3 (EPT memory type) is 2, 3, or 7 (these values are reserved).

EPT misconfigurations result when an EPT paging-structure entry is configured with settings reserved for future functionality. Software developers should be aware that such settings may be used in the future and that an EPT paging-structure entry that causes an EPT misconfiguration on one processor might not do so in the future.

1. If the logical processor is using PAE paging—because CR0.PG = CR4.PAE = 1 and IA32\_EFER.LMA = 0—the MOV to CR3 instruction loads the PDPTes from memory using the guest-physical address being loaded into CR3. In this case, therefore, the MOV to CR3 instruction may cause an EPT misconfiguration, an EPT violation, or a page-modification log-full event.

2. If the “mode-based execute control for EPT” VM-execution control is 1, setting bit 2 indicates that instruction fetches are allowed from supervisor-mode linear addresses.

3. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

**Table 27-6. Format of an EPT Page-Table Entry that Maps a 4-KByte Page**

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 4-KByte page referenced by this entry
1	Write access; indicates whether writes are allowed to the 4-KByte page referenced by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 4-KByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 4-KByte page controlled by this entry
5:3	EPT memory type for this 4-KByte page (see Section 27.2.7)
6	Ignore PAT memory type for this 4-KByte page (see Section 27.2.7)
7	Ignored
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 27.2.5). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 4-KByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of the 4-KByte page referenced by this entry <sup>1</sup>
51:N	Reserved (must be 0)
59:52	Ignored
60	Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 4-KByte page mapped by this entry (see Section 27.2.3.2). Ignored if bit 7 of EPTP is 0
61	Sub-page write permissions. If the “sub-page write permissions for EPT” VM-execution control is 1, writes to individual 128-byte regions of the 4-KByte page referenced by this entry may be allowed even if the page would normally not be writable (see Section 27.2.4). If “sub-page write permissions for EPT” VM-execution control is 0, this bit is ignored.
62	Ignored
63	Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 24.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

**NOTES:**

1. N is the physical-address width supported by the logical processor.

**27.2.3.2 EPT Violations**

An EPT violation may occur during an access using a guest-physical address whose translation does not cause an EPT misconfiguration. An EPT violation occurs in any of the following situations:

- Translation of the guest-physical address encounters an EPT paging-structure entry that is not present (see Section 27.2.2).





- The access is a data write and, for any byte to be written, bit 1 (write access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte. Writes by the logical processor to guest paging structures to update accessed and dirty flags are considered to be data writes.
 

If bit 6 of the EPT pointer (EPTP) is 1 (enabling accessed and dirty flags for EPT), processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations. Thus, if bit 1 is clear in any of the EPT paging-structure entries used to translate the guest-physical address of a guest paging-structure entry, an attempt to use that entry to translate a linear address causes an EPT violation.

(This does not apply to loads of the PDPTTE registers by the MOV to CR instruction for PAE paging; see Section 4.4.1. Those loads of guest PDPTTEs are treated as reads and do not cause EPT violations due to a guest-physical address not being writable.)

If the “sub-page write permissions for EPT” VM-execution control is 1, data writes to a guest-physical address that would cause an EPT violation (as indicated above) do not do so in certain situations. If the guest-physical address is mapped using a 4-KByte page and bit 61 (sub-page write permissions) of the EPT PTE used to map the page is 1, writes to certain 128-byte sub-pages may be allowed. See Section 27.2.4 for details.
- The access is an instruction fetch and the EPT paging structures prevent execute access to any of the bytes being fetched. Whether this occurs depends upon the setting of the “mode-based execute control for EPT” VM-execution control:
  - If the control is 0, an instruction fetch from a byte is prevented if bit 2 (execute access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.
  - If the control is 1, an instruction fetch from a byte is prevented in either of the following cases:
    - Paging maps the linear address of the byte as a supervisor-mode address and bit 2 (execute access for supervisor-mode linear addresses) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.
 

Paging maps a linear address as a supervisor-mode address if the U/S flag (bit 2) is 0 in at least one of the paging-structure entries controlling the translation of the linear address.
    - Paging maps the linear address of the byte as a user-mode address and bit 10 (execute access for user-mode linear addresses) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.
 

Paging maps a linear address as a user-mode address if the U/S flag is 1 in all of the paging-structure entries controlling the translation of the linear address. If paging is disabled (CR0.PG = 0), every linear address is a user-mode address.
- If supervisor shadow-stack control is enabled (by setting bit 7 of EPTP), the access is a supervisor shadow-stack access, and the EPT paging-structure entries used to translate the guest-physical address of the access disallow supervisor shadow-stack accesses. Such an access is disallowed if any of the following hold:
  - Bit 0 (read access) is clear in any EPT paging-structure entry used to translate the guest-physical address of the access.
  - Bit 1 (write access) is clear in any EPT paging-structure entry that references an EPT paging structure in the translation of the guest-physical address. (Clearing bit 1 in the EPT paging-structure entry that maps the page of the guest-physical address does not disallow shadow-stack reads and writes.)
  - Bit 60 (supervisor-shadow stack access) is clear in the EPT paging-structure entry that maps the page of the guest-physical address.

Supervisor shadow-stack control and the supervisor-shadow stack access bits in EPT paging-structure entries do not affect other accesses (including user shadow-stack accesses).

### 27.2.3.3 Prioritization of EPT Misconfigurations and EPT Violations

The translation of a linear address to a physical address requires one or more translations of guest-physical addresses using EPT (see Section 27.2.1). This section specifies the relative priority of EPT-induced VM exits with respect to each other and to other events that may be encountered when accessing memory using a linear address.

For an access to a guest-physical address, determination of whether an EPT misconfiguration or an EPT violation occurs is based on an iterative process:<sup>1</sup>

1. An EPT paging-structure entry is read (initially, this is an EPT PML4 entry):
  - a. If the entry is not present (see Section 27.2.2), an EPT violation occurs.
  - b. If the entry is present but its contents are not configured properly (see Section 27.2.3.1), an EPT misconfiguration occurs.
  - c. If the entry is present and its contents are configured properly, operation depends on whether the entry references another EPT paging structure (whether it is an EPT PDE with bit 7 set to 1 or an EPT PTE):
    - i) If the entry does reference another EPT paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
    - ii) Otherwise, the entry is used to produce the ultimate physical address (the translation of the original guest-physical address); step 2 is executed.
2. Once the ultimate physical address is determined, the privileges determined by the EPT paging-structure entries are evaluated:
  - a. If the access to the guest-physical address is not allowed by these privileges (see Section 27.2.3.2), an EPT violation occurs.
  - b. If the access to the guest-physical address is allowed by these privileges, memory is accessed using the ultimate physical address.

If  $CR0.PG = 1$ , the translation of a linear address is also an iterative process, with the processor first accessing an entry in the guest paging structure referenced by the guest-physical address in CR3 (or, if PAE paging is in use, the guest-physical address in the appropriate PDPTTE register), then accessing an entry in another guest paging structure referenced by the guest-physical address in the first guest paging-structure entry, etc. Each guest-physical address is itself translated using EPT and may cause an EPT-induced VM exit. The following items detail how page faults and EPT-induced VM exits are recognized during this iterative process:

1. An attempt is made to access a guest paging-structure entry with a guest-physical address (initially, the address in CR3 or PDPTTE register).
  - a. If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.
  - b. If the access does not cause an EPT-induced VM exit, bit 0 (the present flag) of the entry is consulted:
    - i) If the present flag is 0 or any reserved bit is set, a page fault occurs.
    - ii) If the present flag is 1, no reserved bit is set, operation depends on whether the entry references another guest paging structure (whether it is a guest PDE with  $PS = 1$  or a guest PTE):
      - If the entry does reference another guest paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
      - Otherwise, the entry is used to produce the ultimate guest-physical address (the translation of the original linear address); step 2 is executed.
2. Once the ultimate guest-physical address is determined, the privileges determined by the guest paging-structure entries are evaluated:
  - a. If the access to the linear address is not allowed by these privileges (e.g., it was a write to a read-only page), a page fault occurs.
  - b. If the access to the linear address is allowed by these privileges, an attempt is made to access memory at the ultimate guest-physical address:
    - i) If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.
    - ii) If the access does not cause an EPT-induced VM exit, memory is accessed using the ultimate physical address (the translation, using EPT, of the ultimate guest-physical address).

If  $CR0.PG = 0$ , a linear address is treated as a guest-physical address and is translated using EPT (see above). This process, if it completes without an EPT violation or EPT misconfiguration, produces a physical address and deter-

---

1. This is a simplification of the more detailed description given in Section 27.2.2.

mines the privileges allowed by the EPT paging-structure entries. If these privileges do not allow the access to the physical address (see Section 27.2.3.2), an EPT violation occurs. Otherwise, memory is accessed using the physical address.

## 27.2.4 Sub-Page Write Permissions

Section 27.2.3.2 explained how EPT enforces the access rights for guest-physical addresses using EPT violations. Since these access rights are determined using the EPT paging-structure entries that are used to translate a guest-physical address, their granularity is limited to that which is used to map pages (1-GByte, 2-MByte, or 4-KByte).

The **sub-page write-permission** feature allows the control of write accesses to guest-physical addresses to be controlled at finer granularity. Sub-page write permissions allow write accesses to be controlled at the granularity of naturally aligned 128-byte **sub-pages**. Specifically, the feature allows writes to selected sub-pages of 4-KByte page that would otherwise not be writable.

Sub-page write permissions are enabled setting the “sub-page write permissions for EPT” VM-execution control to 1. The remainder of this section describes changes to processor operation with this control setting.

Section 27.2.4.1 identifies the data accesses that are eligible for sub-page write permissions. Section 27.2.4.2 explains how the processor determines whether to allow such an access.

### 27.2.4.1 Write Accesses That Are Eligible for Sub-Page Write Permissions

A guest-physical address is eligible for sub-page write permissions if writes to it would be disallowed following Section 27.2.3.2: bit 1 (write access) is clear in any of the EPT paging-structure entries used to translate the guest-physical address. Guest-physical addresses to which writes would be disallowed for other reasons (e.g., the translation encounters an EPT paging-structure entry that is not present) are not eligible for sub-page write permissions.

In addition, a guest-physical address is eligible for sub-page write permissions only if it is mapped using a 4-KByte page and bit 61 (sub-page write permissions) of the EPT PTE used to map the page is 1. (Guest-physical addresses mapped with larger pages are not eligible for sub-page write permissions.)

For some memory accesses, the processor ignores bit 61 in an EPT PTE used to map a 4-KByte page and does not apply sub-page write permissions to the access. (In such a case, the access causes an EPT violation when indicated by the conditions given in Section 27.2.3.2.) Sub-page write permissions never apply to the following accesses:

- A write access performed within a transactional region.
- A write access by an enclave to an address within the enclave's ELRANGE. (Sub-page write permissions may apply to write accesses by an enclaves to addresses outside its ELRANGE.)
- A write access to the enclave page cache (EPC) by an Intel SGX instruction.
- A write access to a guest paging structure to update an accessed or dirty flag.
- Processor accesses to guest paging-structure entries when accessed and dirty flags for EPT are enabled (such accesses are treated as writes with regard to EPT violations).

There are additional accesses to which sub-page write permissions might not be applied (behavior is model-specific). The following items enumerate examples:

- A write access that crosses two 4-KByte pages. In this case, sub-page permissions may be applied to neither or to only one of the pages. (There is no write to either page unless the write is allowed to both pages.)
- A write access by an instruction that performs multiple write accesses (sub-page write permissions are intended principally for basic instructions such as AND, MOV, OR, TEST, XCHG, and XOR).

If a guest-physical address is eligible for sub-page write permissions, the processor determines whether to allow write to the address using the process described in Section 27.2.4.2.

If a guest-physical address is eligible for sub-page write permissions and that address translates to an address on the APIC-access page (see Section 28.4), the processor may treat a write access to the address as if the “virtualize APIC accesses” VM-execution control were 0. For that reason, it is recommended that software not configure any guest-physical address that translates to an address on the APIC-access page to be eligible for sub-page write permissions.

### 27.2.4.2 Determining an Access's Sub-Page Write Permission

Sub-page write permissions control write accesses individually to each of the 32 128-byte sub-pages of a 4-KByte page. Bits 11:7 of guest-physical address identify the sub-page.

For each guest-physical address eligible for sub-page write permissions, there is a 64-bit **sub-page permission vector (SPP vector)**. All addresses on a 4-KByte page use the same SPP vector. If an address's sub-page number (bits 11:7 of the address) is *S*, writes to address are allowed if and only if bit 2*S* of the sub-page permission is set to 1. (The bits at odd positions in a SPP vector are not used and must be zero.)

Each page's SPP vector is located in memory. For a write to a guest-physical address eligible for sub-page write permissions, the processor uses the following process to locate the address's SPP vector:

1. The SPPTP (sub-page-permission-table pointer) VM-execution control field contains the physical address of the 4-KByte root SPP table (SSPL4 table). Bits 47:39 of the guest-physical address identify a 64-bit entry in that table, called an SPPL4E.
2. A 4-KByte SPPL3 table is located at the physical address in the selected SPPL4E. Bits 38:30 of the guest-physical address identify a 64-bit entry in that table, called an SPPL3E.
3. A 4-KByte SPPL2 table is located at the physical address in the selected SPPL3E. Bits 29:21 of the guest-physical address identify a 64-bit entry in that table, called an SPPL2E.
4. A 4-KByte SPP-vector table (SSPL1 table) is located at the physical address in the selected SPPL2E. Bits 20:12 of the guest-physical address identify the 64-bit SPP vector for the address. As noted earlier, bit 2*S* of the sub-page permission vector determines whether the address may be written, where *S* is the value of address bits 11:7.

(The memory type used to access these tables is reported in bits 53:50 of the IA32\_VMX\_BASIC MSR. See Appendix A.1.)

A write access to multiple 128-byte sub-pages on a single 4-KByte page is allowed only if the indicated bit in the page's SPP vector for each of those sub-pages. The following items apply to cases in which an access writes to two 4-KByte pages:

- If a write to either page would be disallowed according to Section 28.2.3.2, the access might be disallowed even if the guest-physical address of that page is eligible for sub-page write permissions. (This behavior is model-specific.)
- The access is allowed only if, for each page, either (1) a write to the page would be allowed following Section 28.2.3.2; or (2) both (a) the guest-physical address of that page is eligible for sub-page write permissions; and (b) the page's sub-page vector allows the write (as described above).

Bit 0 of each entry (SPPL4E, SPPL3E, or SPPL2E) is the entry's **valid** bit. If the process above accesses an entry in which this bit is 0, the process stops and the logical processor incurs an **SPP miss**.

In each entry (SPPL4E, SPPL3E, or SPPL2E), bits 11:1 are reserved, as are bits 63:N, where N is the processor's physical-address width. If the process above accesses an entry in which the valid bit is 1 and in which some reserved bit is set, the process stops and the logical processor incurs an **SPP misconfiguration**. Bits in an SPP vector in odd positions are also reserved; an SPP misconfiguration occurs also any of those bits are set in the final SPP vector.

SPP misses and SPP misconfigurations are called **SPP-related events** and cause VM exits.

### 27.2.5 Accessed and Dirty Flags for EPT

The Intel 64 architecture supports **accessed and dirty flags** in ordinary paging-structure entries (see Section 4.8). Some processors also support corresponding flags in EPT paging-structure entries. Software should read the VMX capability MSR IA32\_VMX\_EPT\_VPID\_CAP (see Appendix A.10) to determine whether the processor supports this feature.

Software can enable accessed and dirty flags for EPT using bit 6 of the extended-page-table pointer (EPTP), a VM-execution control field (see Table 23-9 in Section 23.6.11). If this bit is 1, the processor will set the accessed and dirty flags for EPT as described below. In addition, setting this flag causes processor accesses to guest paging-structure entries to be treated as writes (see below and Section 27.2.3.2).

For any EPT paging-structure entry that is used during guest-physical-address translation, bit 8 is the accessed flag. For a EPT paging-structure entry that maps a page (as opposed to referencing another EPT paging structure), bit 9 is the dirty flag.

Whenever the processor uses an EPT paging-structure entry as part of guest-physical-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a guest-physical address, the processor sets the dirty flag (if it is not already set) in the EPT paging-structure entry that identifies the final physical address for the guest-physical address (either an EPT PTE or an EPT paging-structure entry in which bit 7 is 1).

When accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes (see Section 27.2.3.2). Thus, such an access will cause the processor to set the dirty flag in the EPT paging-structure entry that identifies the final physical address of the guest paging-structure entry.

(This does not apply to loads of the PDPT registers for PAE paging by the MOV to CR instruction; see Section 4.4.1. Those loads of guest PDPTes are treated as reads and do not cause the processor to set the dirty flag in any EPT paging-structure entry.)

These flags are “sticky,” meaning that, once set, the processor does not clear them; only software can clear them.

A processor may cache information from the EPT paging-structure entries in TLBs and paging-structure caches (see Section 27.3). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the processor might not set the corresponding bit in memory on a subsequent access using an affected guest-physical address.

## 27.2.6 Page-Modification Logging

When accessed and dirty flags for EPT are enabled, software can track writes to guest-physical addresses using a feature called **page-modification logging**.

Software can enable page-modification logging by setting the “enable PML” VM-execution control (see Table 23-7 in Section 23.6.2). When this control is 1, the processor adds entries to the **page-modification log** as described below. The page-modification log is a 4-KByte region of memory located at the physical address in the PML address VM-execution control field. The page-modification log consists of 512 64-bit entries; the PML index VM-execution control field indicates the next entry to use.

Before allowing a guest-physical access, the processor may determine that it first needs to set an accessed or dirty flag for EPT (see Section 27.2.5). When this happens, the processor examines the PML index. If the PML index is not in the range 0–511, there is a **page-modification log-full event** and a VM exit occurs. In this case, the accessed or dirty flag is not set, and the guest-physical access that triggered the event does not occur.

If instead the PML index is in the range 0–511, the processor proceeds to update accessed or dirty flags for EPT as described in Section 27.2.5. If the processor updated a dirty flag for EPT (changing it from 0 to 1), it then operates as follows:

1. The guest-physical address of the access is written to the page-modification log. Specifically, the guest-physical address is written to physical address determined by adding 8 times the PML index to the PML address. Bits 11:0 of the value written are always 0 (the guest-physical address written is thus 4-KByte aligned).
2. The PML index is decremented by 1 (this may cause the value to transition from 0 to FFFFH).

Because the processor decrements the PML index with each log entry, the value may transition from 0 to FFFFH. At that point, no further logging will occur, as the processor will determine that the PML index is not in the range 0–511 and will generate a page-modification log-full event (see above).

## 27.2.7 EPT and Memory Typing

This section specifies how a logical processor determines the memory type use for a memory access while EPT is in use. (See Chapter 11, “Memory Cache Control” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for details of memory typing in the Intel 64 architecture.) Section 27.2.7.1 explains how the memory type is determined for accesses to the EPT paging structures. Section 27.2.7.2 explains how the memory type is determined for an access using a guest-physical address that is translated using EPT.



### 27.2.7.1 Memory Type Used for Accessing EPT Paging Structures

This section explains how the memory type is determined for accesses to the EPT paging structures. The determination is based first on the value of bit 30 (cache disable—CD) in control register CR0:

- If CR0.CD = 0, the memory type used for any such reference is the EPT paging-structure memory type, which is specified in bits 2:0 of the extended-page-table pointer (EPTP), a VM-execution control field (see Section 23.6.11). A value of 0 indicates the uncacheable type (UC), while a value of 6 indicates the write-back type (WB). Other values are reserved.
- If CR0.CD = 1, the memory type used for any such reference is uncacheable (UC).

The MTRRs have no effect on the memory type used for an access to an EPT paging structure.

### 27.2.7.2 Memory Type Used for Translated Guest-Physical Addresses

The **effective memory type** of a memory access using a guest-physical address (an access that is translated using EPT) is the memory type that is used to access memory. The effective memory type is based on the value of bit 30 (cache disable—CD) in control register CR0; the **last** EPT paging-structure entry used to translate the guest-physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE); and the PAT memory type (see below):

- The **PAT memory type** depends on the value of CR0.PG:
  - If CR0.PG = 0, the PAT memory type is WB (writeback).<sup>1</sup>
  - If CR0.PG = 1, the PAT memory type is the memory type selected from the IA32\_PAT MSR as specified in Section 11.12.3, “Selecting a Memory Type from the PAT”.<sup>2</sup>
- The **EPT memory type** is specified in bits 5:3 of the last EPT paging-structure entry: 0 = UC; 1 = WC; 4 = WT; 5 = WP; and 6 = WB. Other values are reserved and cause EPT misconfigurations (see Section 27.2.3).
- If CR0.CD = 0, the effective memory type depends upon the value of bit 6 of the last EPT paging-structure entry:
  - If the value is 0, the effective memory type is the combination of the EPT memory type and the PAT memory type specified in Table 11-7 in Section 11.5.2.2, using the EPT memory type in place of the MTRR memory type.
  - If the value is 1, the memory type used for the access is the EPT memory type. The PAT memory type is ignored.
- If CR0.CD = 1, the effective memory type is UC.

The MTRRs have no effect on the memory type used for an access to a guest-physical address.

## 27.3 CACHING TRANSLATION INFORMATION

Processors supporting Intel® 64 and IA-32 architectures may accelerate the address-translation process by caching on the processor data from the structures in memory that control that process. Such caching is discussed in Section 4.10, “Caching Translation Information” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. The current section describes how this caching interacts with the VMX architecture.

The VPID and EPT features of the architecture for VMX operation augment this caching architecture. EPT defines the guest-physical address space and defines translations to that address space (from the linear-address space)

- 
1. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
  2. Table 11-11 in Section 11.12.3, “Selecting a Memory Type from the PAT” illustrates how the PAT memory type is selected based on the values of the PAT, PCD, and PWT bits in a page-table entry (or page-directory entry with PS = 1). For accesses to a guest paging-structure entry X, the PAT memory type is selected from the table by using a value of 0 for the PAT bit with the values of PCD and PWT from the paging-structure entry Y that references X (or from CR3 if X is in the root paging structure). With PAE paging, the PAT memory type for accesses to the PDPTes is WB.

and from that address space (to the physical-address space). Both features control the ways in which a logical processor may create and use information cached from the paging structures.

Section 27.3.1 describes the different kinds of information that may be cached. Section 27.3.2 specifies when such information may be cached and how it may be used. Section 27.3.3 details how software can invalidate cached information.

## 27.3.1 Information That May Be Cached

Section 4.10, “Caching Translation Information” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* identifies two kinds of translation-related information that may be cached by a logical processor: **translations**, which are mappings from linear page numbers to physical page frames, and **paging-structure caches**, which map the upper bits of a linear page number to information from the paging-structure entries used to translate linear addresses matching those upper bits.

The same kinds of information may be cached when VPIDs and EPT are in use. A logical processor may cache and use such information based on its function. Information with different functionality is identified as follows:

- **Linear mappings.**<sup>1</sup> There are two kinds:
  - Linear translations. Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
  - Linear paging-structure-cache entries. Each of these is a mapping from the upper portion of a linear address to the physical address of the paging structure used to translate the corresponding region of the linear-address space, along with information about access privileges. For example, bits 47:39 of a linear address would map to the address of the relevant page-directory-pointer table.

Linear mappings do not contain information from any EPT paging structure.

- **Guest-physical mappings.**<sup>2</sup> There are two kinds:
  - Guest-physical translations. Each of these is a mapping from a guest-physical page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
  - Guest-physical paging-structure-cache entries. Each of these is a mapping from the upper portion of a guest-physical address to the physical address of the EPT paging structure used to translate the corresponding region of the guest-physical address space, along with information about access privileges.

The information in guest-physical mappings about access privileges and memory typing is derived from EPT paging structures.

- **Combined mappings.**<sup>3</sup> There are two kinds:
  - Combined translations. Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
  - Combined paging-structure-cache entries. Each of these is a mapping from the upper portion of a linear address to the physical address of the paging structure used to translate the corresponding region of the linear-address space, along with information about access privileges.

The information in combined mappings about access privileges and memory typing is derived from both guest paging structures and EPT paging structures.

Guest-physical mappings and combined mappings may also include SPP vectors and information about the data structures used to locate SPP vectors (see Section 27.2.4.2).

## 27.3.2 Creating and Using Cached Translation Information

The following items detail the creation of the mappings described in the previous section:<sup>4</sup>

- 
1. Earlier versions of this manual used the term “VPID-tagged” to identify linear mappings.
  2. Earlier versions of this manual used the term “EPTP-tagged” to identify guest-physical mappings.
  3. Earlier versions of this manual used the term “dual-tagged” to identify combined mappings.



- The following items describe the creation of mappings while EPT is not in use (including execution outside VMX non-root operation):
  - Linear mappings may be created. They are derived from the paging structures referenced (directly or indirectly) by the current value of CR3 and are associated with the current VPID and the current PCID.
  - No linear mappings are created with information derived from paging-structure entries that are not present (bit 0 is 0) or that set reserved bits. For example, if a PTE is not present, no linear mapping are created for any linear page number whose translation would use that PTE.
  - No guest-physical or combined mappings are created while EPT is not in use.
- The following items describe the creation of mappings while EPT is in use:
  - Guest-physical mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by bits 51:12 of the current EPTP. These 40 bits contain the address of the EPT-PML4-table. (the notation **EP4TA** refers to those 40 bits). Newly created guest-physical mappings are associated with the current EP4TA.
  - Combined mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by the current EP4TA. If CR0.PG = 1, they are also derived from the paging structures referenced (directly or indirectly) by the current value of CR3. They are associated with the current VPID, the current PCID, and the current EP4TA.<sup>1</sup> No combined paging-structure-cache entries are created if CR0.PG = 0.<sup>2</sup>
  - No guest-physical mappings or combined mappings are created with information derived from EPT paging-structure entries that are not present (see Section 27.2.2) or that are misconfigured (see Section 27.2.3.1).
  - No combined mappings are created with information derived from guest paging-structure entries that are not present or that set reserved bits.
  - No linear mappings are created while EPT is in use.

The following items detail the use of the various mappings:

- If EPT is not in use (e.g., when outside VMX non-root operation), a logical processor may use cached mappings as follows:
  - For accesses using linear addresses, it may use linear mappings associated with the current VPID and the current PCID. It may also use global TLB entries (linear mappings) associated with the current VPID and any PCID.
  - No guest-physical or combined mappings are used while EPT is not in use.
- If EPT is in use, a logical processor may use cached mappings as follows:
  - For accesses using linear addresses, it may use combined mappings associated with the current VPID, the current PCID, and the current EP4TA. It may also use global TLB entries (combined mappings) associated with the current VPID, the current EP4TA, and any PCID.
  - For accesses using guest-physical addresses, it may use guest-physical mappings associated with the current EP4TA.
  - No linear mappings are used while EPT is in use.

---

4. This section associated cached information with the current VPID and PCID. If PCIDs are not supported or are not being used (e.g., because CR4.PCIDE = 0), all the information is implicitly associated with PCID 000H; see Section 4.10.1, “Process-Context Identifiers (PCIDs),” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

1. At any given time, a logical processor may be caching combined mappings for a VPID and a PCID that are associated with different EP4TAs. Similarly, it may be caching combined mappings for an EP4TA that are associated with different VPIDs and PCIDs.

2. If the capability MSR IA32\_VMX\_CRO\_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

## 27.3.3 Invalidating Cached Translation Information

Software modifications of paging structures (including EPT paging structures and the data structures used to locate SPP vectors) may result in inconsistencies between those structures and the mappings cached by a logical processor. Certain operations invalidate information cached by a logical processor and can be used to eliminate such inconsistencies.

### 27.3.3.1 Operations that Invalidate Cached Mappings

The following operations invalidate cached mappings as indicated:

- Operations that architecturally invalidate entries in the TLBs or paging-structure caches independent of VMX operation (e.g., the INVLPG and INVPCID instructions) invalidate linear mappings and combined mappings.<sup>1</sup> They are required to do so only for the current VPID (but, for combined mappings, all EP4TAs). Linear mappings for the current VPID are invalidated even if EPT is in use.<sup>2</sup> Combined mappings for the current VPID are invalidated even if EPT is not in use.<sup>3</sup>
- An EPT violation invalidates any guest-physical mappings (associated with the current EP4TA) that would be used to translate the guest-physical address that caused the EPT violation. If that guest-physical address was the translation of a linear address, the EPT violation also invalidates any combined mappings for that linear address associated with the current PCID, the current VPID and the current EP4TA.
- If the “enable VPID” VM-execution control is 0, VM entries and VM exits invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs). Combined mappings for VPID 0000H are invalidated for all EP4TAs.
- Execution of the INVVPID instruction invalidates linear mappings and combined mappings. Invalidation is based on instruction operands, called the INVVPID type and the INVVPID descriptor. Four INVVPID types are currently defined:
  - **Individual-address.** If the INVVPID type is 0, the logical processor invalidates linear mappings and combined mappings associated with the VPID specified in the INVVPID descriptor and that would be used to translate the linear address specified in of the INVVPID descriptor. Linear mappings and combined mappings for that VPID and linear address are invalidated for all PCIDs and, for combined mappings, all EP4TAs. (The instruction may also invalidate mappings associated with other VPIDs and for other linear addresses.)
  - **Single-context.** If the INVVPID type is 1, the logical processor invalidates all linear mappings and combined mappings associated with the VPID specified in the INVVPID descriptor. Linear mappings and combined mappings for that VPID are invalidated for all PCIDs and, for combined mappings, all EP4TAs. (The instruction may also invalidate mappings associated with other VPIDs.)
  - **All-context.** If the INVVPID type is 2, the logical processor invalidates linear mappings and combined mappings associated with all VPIDs except VPID 0000H and with all PCIDs. (The instruction may also invalidate linear mappings with VPID 0000H.) Combined mappings are invalidated for all EP4TAs.
  - **Single-context-retaining-globals.** If the INVVPID type is 3, the logical processor invalidates linear mappings and combined mappings associated with the VPID specified in the INVVPID descriptor. Linear mappings and combined mappings for that VPID are invalidated for all PCIDs and, for combined mappings, all EP4TAs. The logical processor is **not** required to invalidate information that was used for **global** translations (although it may do so). See Section 4.10, “Caching Translation Information” for details regarding global translations. (The instruction may also invalidate mappings associated with other VPIDs.)

See Chapter 29 for details of the INVVPID instruction. See Section 27.3.3.3 for guidelines regarding use of this instruction.

- 
1. See Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for an enumeration of operations that architecturally invalidate entries in the TLBs and paging-structure caches independent of VMX operation.
  2. While no linear mappings are created while EPT is in use, a logical processor may retain, while EPT is in use, linear mappings (for the same VPID as the current one) there were created earlier, when EPT was not in use.
  3. While no combined mappings are created while EPT is not in use, a logical processor may retain, while EPT is in not use, combined mappings (for the same VPID as the current one) there were created earlier, when EPT was in use.

- Execution of the INVEPT instruction invalidates guest-physical mappings and combined mappings. Invalidation is based on instruction operands, called the INVEPT type and the INVEPT descriptor. Two INVEPT types are currently defined:
  - **Single-context.** If the INVEPT type is 1, the logical processor invalidates all guest-physical mappings and combined mappings associated with the EP4TA specified in the INVEPT descriptor. Combined mappings for that EP4TA are invalidated for all VPIDs and all PCIDs. (The instruction may invalidate mappings associated with other EP4TAs.)
  - **All-context.** If the INVEPT type is 2, the logical processor invalidates guest-physical mappings and combined mappings associated with all EP4TAs (and, for combined mappings, for all VPIDs and PCIDs).
- See Chapter 29 for details of the INVEPT instruction. See Section 27.3.3.4 for guidelines regarding use of this instruction.
- A power-up or a reset invalidates all linear mappings, guest-physical mappings, and combined mappings.

### 27.3.3.2 Operations that Need Not Invalidate Cached Mappings

The following items detail cases of operations that are not required to invalidate certain cached mappings:

- Operations that architecturally invalidate entries in the TLBs or paging-structure caches independent of VMX operation are not required to invalidate any guest-physical mappings.
- The INVVPID instruction is not required to invalidate any guest-physical mappings.
- The INVEPT instruction is not required to invalidate any linear mappings.
- VMX transitions are not required to invalidate any guest-physical mappings. If the “enable VPID” VM-execution control is 1, VMX transitions are not required to invalidate any linear mappings or combined mappings.
- The VMXOFF and VMXON instructions are not required to invalidate any linear mappings, guest-physical mappings, or combined mappings.

A logical processor may invalidate any cached mappings at any time. For this reason, the operations identified above may invalidate the indicated mappings despite the fact that doing so is not required.

### 27.3.3.3 Guidelines for Use of the INVVPID Instruction

■ The need for VMM software to use the INVVPID instruction depends on how that software is virtualizing memory.

If EPT is not in use, it is likely that the VMM is virtualizing the guest paging structures. Such a VMM may configure the VMCS so that all or some of the operations that invalidate entries in the TLBs and the paging-structure caches (e.g., the INVLPG instruction) cause VM exits. If VMM software is emulating these operations, it may be necessary to use the INVVPID instruction to ensure that the logical processor’s TLBs and the paging-structure caches are appropriately invalidated.

Requirements of when software should use the INVVPID instruction depend on the specific algorithm being used for page-table virtualization. The following items provide guidelines for software developers:

- Emulation of the INVLPG instruction may require execution of the INVVPID instruction as follows:
  - The INVVPID type is individual-address (0).
  - The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.
  - The linear address in the INVVPID descriptor is that of the operand of the INVLPG instruction being emulated.
- Some instructions invalidate all entries in the TLBs and paging-structure caches—except for global translations. An example is the MOV to CR3 instruction. (See Section 4.10, “Caching Translation Information” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for details regarding global translations.) Emulation of such an instruction may require execution of the INVVPID instruction as follows:
  - The INVVPID type is single-context-retaining-globals (3).
  - The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.

- Some instructions invalidate all entries in the TLBs and paging-structure caches—including for global translations. An example is the MOV to CR4 instruction if the value of value of bit 4 (page global enable—PGE) is changing. Emulation of such an instruction may require execution of the INVVPID instruction as follows:
  - The INVVPID type is single-context (1).
  - The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.

If EPT is not in use, the logical processor associates all mappings it creates with the current VPID, and it will use such mappings to translate linear addresses. For that reason, a VMM should not use the same VPID for different non-EPT guests that use different page tables. Doing so may result in one guest using translations that pertain to the other.

If EPT is in use, the instructions enumerated above might not be configured to cause VM exits and the VMM might not be emulating them. In that case, executions of the instructions by guest software properly invalidate the required entries in the TLBs and paging-structure caches (see Section 27.3.3.1); execution of the INVVPID instruction is not required.

If EPT is in use, the logical processor associates all mappings it creates with the value of bits 51:12 of current EPTP. If a VMM uses different EPTP values for different guests, it may use the same VPID for those guests. Doing so cannot result in one guest using translations that pertain to the other.

The following guidelines apply more generally and are appropriate even if EPT is in use:

- As detailed in Section 28.4.5, an access to the APIC-access page might not cause an APIC-access VM exit if software does not properly invalidate information that may be cached from the paging structures. If, at one time, the current VPID on a logical processor was a non-zero value X, it is recommended that software use the INVVPID instruction with the “single-context” INVVPID type and with VPID X in the INVVPID descriptor before a VM entry on the same logical processor that establishes VPID X and either (a) the “virtualize APIC accesses” VM-execution control was changed from 0 to 1; or (b) the value of the APIC-access address was changed.
- Software can use the INVVPID instruction with the “all-context” INVVPID type immediately after execution of the VMXON instruction or immediately prior to execution of the VMXOFF instruction. Either prevents potentially undesired retention of information cached from paging structures between separate uses of VMX operation.

### 27.3.3.4 Guidelines for Use of the INVEPT Instruction

The following items provide guidelines for use of the INVEPT instruction to invalidate information cached from the EPT paging structures.

- Software should use the INVEPT instruction with the “single-context” INVEPT type after making any of the following changes to an EPT paging-structure entry (the INVEPT descriptor should contain an EPTP value that references — directly or indirectly — the modified EPT paging structure):
  - Changing any of the privilege bits 2:0 from 1 to 0.<sup>1</sup>
  - Changing the physical address in bits 51:12.
  - Clearing bit 8 (the accessed flag) if accessed and dirty flags for EPT will be enabled.
  - For an EPT PDPTTE or an EPT PDE, changing bit 7 (which determines whether the entry maps a page).
  - For the **last** EPT paging-structure entry used to translate a guest-physical address (an EPT PDPTTE with bit 7 set to 1, an EPT PDE with bit 7 set to 1, or an EPT PTE), changing either bits 5:3 or bit 6. (These bits determine the effective memory type of accesses using that EPT paging-structure entry; see Section 27.2.7.)
  - For the **last** EPT paging-structure entry used to translate a guest-physical address (an EPT PDPTTE with bit 7 set to 1, an EPT PDE with bit 7 set to 1, or an EPT PTE), clearing bit 9 (the dirty flag) if accessed and dirty flags for EPT will be enabled.
- Software should use the INVEPT instruction with the “single-context” INVEPT type before a VM entry with an EPTP value X such that X[6] = 1 (accessed and dirty flags for EPT are enabled) if the logical processor had

---

1. If the “mode-based execute control for EPT” VM-execution control is 1, software should use the INVEPT instruction after changing privilege bit 10 from 1 to 0.

earlier been in VMX non-root operation with an EPTP value Y such that  $Y[6] = 0$  (accessed and dirty flags for EPT are not enabled) and  $Y[51:12] = X[51:12]$ .

- Software may use the INVEPT instruction after modifying a present EPT paging-structure entry (see Section 27.2.2) to change any of the privilege bits 2:0 from 0 to 1.<sup>1</sup> Failure to do so may cause an EPT violation that would not otherwise occur. Because an EPT violation invalidates any mappings that would be used by the access that caused the EPT violation (see Section 27.3.3.1), an EPT violation will not recur if the original access is performed again, even if the INVEPT instruction is not executed.
- Because a logical processor does not cache any information derived from EPT paging-structure entries that are not present (see Section 27.2.2) or misconfigured (see Section 27.2.3.1), it is not necessary to execute INVEPT following modification of an EPT paging-structure entry that had been not present or misconfigured.
- As detailed in Section 28.4.5, an access to the APIC-access page might not cause an APIC-access VM exit if software does not properly invalidate information that may be cached from the EPT paging structures. If EPT was in use on a logical processor at one time with EPTP X, it is recommended that software use the INVEPT instruction with the “single-context” INVEPT type and with EPTP X in the INVEPT descriptor before a VM entry on the same logical processor that enables EPT with EPTP X and either (a) the “virtualize APIC accesses” VM-execution control was changed from 0 to 1; or (b) the value of the APIC-access address was changed.
- Software can use the INVEPT instruction with the “all-context” INVEPT type immediately after execution of the VMXON instruction or immediately prior to execution of the VMXOFF instruction. Either prevents potentially undesired retention of information cached from EPT paging structures between separate uses of VMX operation.

In a system containing more than one logical processor, software must account for the fact that information from an EPT paging-structure entry may be cached on logical processors other than the one that modifies that entry. The process of propagating the changes to a paging-structure entry is commonly referred to as “TLB shutdown.” A discussion of TLB shutdown appears in Section 4.10.5, “Propagation of Paging-Structure Changes to Multiple Processors,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

---

1. If the “mode-based execute control for EPT” VM-execution control is 1, software may use the INVEPT instruction after modifying a present EPT paging-structure entry to change privilege bit 10 from 0 to 1.



## 24. Updates to Chapter 30, Volume 3C

Change bars and green text show changes to Chapter 30 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Changes to this chapter: Update to table 30-3, "SMRAM State Save Map for Intel 64 Architecture".

This chapter describes aspects of IA-64 and IA-32 architecture used in system management mode (SMM).

SMM provides an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code. It was introduced into the IA-32 architecture in the Intel386 SL processor (a mobile specialized version of the Intel386 processor). It is also available in the Pentium M, Pentium 4, Intel Xeon, P6 family, and Pentium and Intel486 processors (beginning with the enhanced versions of the Intel486 SL and Intel486 processors).

### 30.1 SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

When SMM is invoked through a system management interrupt (SMI), the processor saves the current state of the processor (the processor's context), then switches to a separate operating environment defined by a new address space. The system management software executive (SMI handler) starts execution in that environment, and the critical code and data of the SMI handler reside in a physical memory region (SMRAM) within that address space. While in SMM, the processor executes SMI handler code to perform operations such as powering down unused disk drives or monitors, executing proprietary code, or placing the whole system in a suspended state. When the SMI handler has completed its operations, it executes a resume (RSM) instruction. This instruction causes the processor to reload the saved context of the processor, switch back to protected or real mode, and resume executing the interrupted application or operating-system program or task.

The following SMM mechanisms make it transparent to applications programs and operating systems:

- The only way to enter SMM is by means of an SMI.
- The processor executes SMM code in a separate address space that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.
- All interrupts normally handled by the operating system are disabled upon entry into SMM.
- The RSM instruction can be executed only in SMM.

Section 30.3 describes transitions into and out of SMM. The execution environment after entering SMM is in real-address mode with paging disabled ( $CR0.PE = CR0.PG = 0$ ). In this initial execution environment, the SMI handler can address up to 4 GBytes of memory and can execute all I/O and system instructions. Section 30.5 describes in detail the initial SMM execution environment for an SMI handler and operation within that environment. The SMI handler may subsequently switch to other operating modes while remaining in SMM.

#### NOTES

Software developers should be aware that, even if a logical processor was using the physical-address extension (PAE) mechanism (introduced in the P6 family processors) or was in IA-32e mode before an SMI, this will not be the case after the SMI is delivered. This is because delivery of an SMI disables paging (see Table 30-4). (This does not apply if the dual-monitor treatment of SMIs and SMM is active; see Section 30.15.)



### 30.1.1 System Management Mode and VMX Operation

Traditionally, SMM services system management interrupts and then resumes program execution (back to the software stack consisting of executive and application software; see Section 30.2 through Section 30.13).

A virtual machine monitor (VMM) using VMX can act as a host to multiple virtual machines and each virtual machine can support its own software stack of executive and application software. On processors that support VMX, virtual-machine extensions may use system-management interrupts (SMIs) and system-management mode (SMM) in one of two ways:

- **Default treatment.** System firmware handles SMIs. The processor saves architectural states and critical states relevant to VMX operation upon entering SMM. When the firmware completes servicing SMIs, it uses RSM to resume VMX operation.
- **Dual-monitor treatment.** Two VM monitors collaborate to control the servicing of SMIs: one VMM operates outside of SMM to provide basic virtualization in support for guests; the other VMM operates inside SMM (while in VMX operation) to support system-management functions. The former is referred to as **executive monitor**, the latter **SMM-transfer monitor (STM)**.<sup>1</sup>

The default treatment is described in Section 30.14, “Default Treatment of SMIs and SMM with VMX Operation and SMX Operation”. Dual-monitor treatment of SMM is described in Section 30.15, “Dual-Monitor Treatment of SMIs and SMM”.

## 30.2 SYSTEM MANAGEMENT INTERRUPT (SMI)

The only way to enter SMM is by signaling an SMI through the SMI# pin on the processor or through an SMI message received through the APIC bus. The SMI is a nonmaskable external interrupt that operates independently from the processor’s interrupt- and exception-handling mechanism and the local APIC. The SMI takes precedence over an NMI and a maskable interrupt. SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

### NOTES

In the Pentium 4, Intel Xeon, and P6 family processors, when a processor that is designated as an application processor during an MP initialization sequence is waiting for a startup IPI (SIPI), it is in a mode where SMIs are masked. However if a SMI is received while an application processor is in the wait for SIPI mode, the SMI will be pended. The processor then responds on receipt of a SIPI by immediately servicing the pended SMI and going into SMM before handling the SIPI.

An SMI may be blocked for one instruction following execution of STI, MOV to SS, or POP into SS.

## 30.3 SWITCHING BETWEEN SMM AND THE OTHER PROCESSOR OPERATING MODES

Figure 2-3 shows how the processor moves between SMM and the other processor operating modes (protected, real-address, and virtual-8086). Signaling an SMI while the processor is in real-address, protected, or virtual-8086 modes always causes the processor to switch to SMM. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

### 30.3.1 Entering SMM

The processor always handles an SMI on an architecturally defined “interruptible” point in program execution (which is commonly at an IA-32 architecture instruction boundary). When the processor receives an SMI, it waits for all instructions to retire and for all stores to complete. The processor then saves its current context in SMRAM (see Section 30.4), enters SMM, and begins to execute the SMI handler.

1. The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether it is supported.

Upon entering SMM, the processor signals external hardware that SMI handling has begun. The signaling mechanism used is implementation dependent. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is asserted each time a bus transaction is generated while the processor is in SMM. For the Pentium and Intel486 processors, the SMIACK# pin is asserted.

An SMI has a greater priority than debug exceptions and external interrupts. Thus, if an NMI, maskable hardware interrupt, or a debug exception occurs at an instruction boundary along with an SMI, only the SMI is handled. Subsequent SMI requests are not acknowledged while the processor is in SMM. The first SMI interrupt request that occurs while the processor is in SMM (that is, after SMM has been acknowledged to external hardware) is latched and serviced when the processor exits SMM with the RSM instruction. The processor will latch only one SMI while in SMM.

See Section 30.5 for a detailed description of the execution environment when in SMM.

### 30.3.2 Exiting From SMM

The only way to exit SMM is to execute the RSM instruction. The RSM instruction is only available to the SMI handler; if the processor is not in SMM, attempts to execute the RSM instruction result in an invalid-opcode exception (#UD) being generated.

The RSM instruction restores the processor's context by loading the state save image from SMRAM back into the processor's registers. The processor then returns an SMIACK transaction on the system bus and returns program control back to the interrupted program.

#### NOTE

On processors that support the shadow-stack feature, RSM loads the SSP register from the state save image in SMRAM (see Table 30-3). The value is made canonical by sign-extension before loading it into SSP.

Upon successful completion of the RSM instruction, the processor signals external hardware that SMM has been exited. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is no longer generated on bus cycles. For the Pentium and Intel486 processors, the SMIACK# pin is deserted.

If the processor detects invalid state information saved in the SMRAM, it enters the shutdown state and generates a special bus cycle to indicate it has entered shutdown state. Shutdown happens only in the following situations:

- A reserved bit in control register CR4 is set to 1 on a write to CR4. This error should not happen unless SMI handler code modifies reserved areas of the SMRAM saved state map (see Section 30.4.1). CR4 is saved in the state map in a reserved location and cannot be read or modified in its saved state.
- An illegal combination of bits is written to control register CR0, in particular PG set to 1 and PE set to 0, or NW set to 1 and CD set to 0.
- CR4.PCIDE would be set to 1 and IA32\_EFER.LMA to 0.
- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.
- CR4.CET would be set to 1 and CR0.WP to 0.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined. On Pentium 4 and later processors, shutdown will inhibit INTR and A20M but will not change any of the other inhibits. On these processors, NMIs will be inhibited if no action is taken in the SMI handler to uninhibit them (see Section 30.8).

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 30.10). Also, the SMBASE address can be changed on a return from SMM (see Section 30.11).

## 30.4 SMRAM

Upon entering SMM, the processor switches to a new address space. Because paging is disabled upon entering SMM, this initial address space maps all memory accesses to the low 4 GBytes of the processor's physical address space. The SMI handler's critical code and data reside in a memory region referred to as system-management RAM (SMRAM). The processor uses a pre-defined region within SMRAM to save the processor's pre-SMI context. SMRAM can also be used to store system management information (such as the system configuration and specific information about powered-down devices) and OEM-specific information.

The default SMRAM size is 64 KBytes beginning at a base physical address in physical memory called the SMBASE (see Figure 30-1). The SMBASE default value following a hardware reset is 30000H. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 8000H]. It stores the processor's state in the area from [SMBASE + FE00H] to [SMBASE + FFFFH]. See Section 30.4.1 for a description of the mapping of the state save area.

The system logic is minimally required to decode the physical address range for the SMRAM from [SMBASE + 8000H] to [SMBASE + FFFFH]. A larger area can be decoded if needed. The size of this SMRAM can be between 32 KBytes and 4 GBytes.

The location of the SMRAM can be changed by changing the SMBASE value (see Section 30.11). It should be noted that all processors in a multiple-processor system are initialized with the same SMBASE value (30000H). Initialization software must sequentially place each processor in SMM and change its SMBASE so that it does not overlap those of other processors.

The actual physical location of the SMRAM can be in system memory or in a separate RAM memory. The processor generates an SMI acknowledge transaction (P6 family processors) or asserts the SMIACK# pin (Pentium and Intel486 processors) when the processor receives an SMI (see Section 30.3.1).

System logic can use the SMI acknowledge transaction or the assertion of the SMIACK# pin to decode accesses to the SMRAM and redirect them (if desired) to specific SMRAM memory. If a separate RAM memory is used for SMRAM, system logic should provide a programmable method of mapping the SMRAM into system memory space when the processor is not in SMM. This mechanism will enable start-up procedures to initialize the SMRAM space (that is, load the SMI handler) before executing the SMI handler during SMM.

### 30.4.1 SMRAM State Save Map

When an IA-32 processor that does not support Intel 64 architecture initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area begins at [SMBASE + 8000H + 7FFFH] and extends down to [SMBASE + 8000H + 7E00H]. Table 30-1 shows the state save map. The offset in column 1 is relative to the SMBASE value plus 8000H. Reserved spaces should not be used by software.

Some of the registers in the SMRAM state save area (marked YES in column 3) may be read and changed by the SMI handler, with the changed values restored to the processor registers by the RSM instruction. Some register images are read-only, and must not be modified (modifying these registers will result in unpredictable behavior). An SMI handler should not rely on any values stored in an area that is marked as reserved.

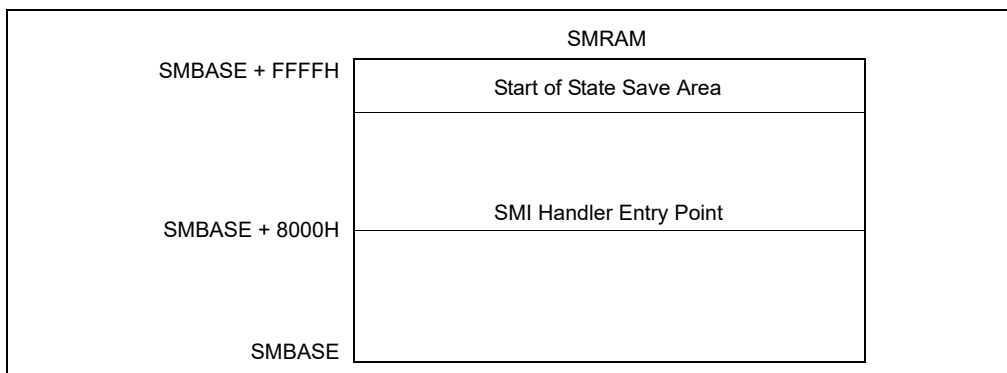


Figure 30-1. SMRAM Usage

Table 30-1. SMRAM State Save Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FFCH	CR0	No
7FF8H	CR3	No
7FF4H	EFLAGS	Yes
7FF0H	EIP	Yes
7FECH	EDI	Yes
7FE8H	ESI	Yes
7FE4H	EBP	Yes
7FE0H	ESP	Yes
7FDCH	EBX	Yes
7FD8H	EDX	Yes
7FD4H	ECX	Yes
7FD0H	EAX	Yes
7FCCH	DR6	No
7FC8H	DR7	No
7FC4H	TR <sup>1</sup>	No
7FC0H	Reserved	No
7FBCH	GS <sup>1</sup>	No
7FB8H	FS <sup>1</sup>	No
7FB4H	DS <sup>1</sup>	No
7FB0H	SS <sup>1</sup>	No
7FACH	CS <sup>1</sup>	No
7FA8H	ES <sup>1</sup>	No
7FA4H	I/O State Field, see Section 30.7	No
7FA0H	I/O Memory Address Field, see Section 30.7	No
7F9FH-7F03H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes
7EF7H - 7E00H	Reserved	No

**NOTE:**

1. The two most significant bytes are reserved.

The following registers are saved (but not readable) and restored upon exiting SMM:

- Control register CR4. (This register is cleared to all 0s when entering SMM).
- The hidden segment descriptor information stored in segment registers CS, DS, ES, FS, GS, and SS.

If an SMI request is issued for the purpose of powering down the processor, the values of all reserved locations in the SMM state save must be saved to nonvolatile memory.

The following state is not automatically saved and restored following an SMI and the RSM instruction, respectively:

- Debug registers DR0 through DR3.
- The x87 FPU registers.
- The MTRRs.
- Control register CR2.
- The model-specific registers (for the P6 family and Pentium processors) or test registers TR3 through TR7 (for the Pentium and Intel486 processors).
- The state of the trap controller.
- The machine-check architecture registers.
- The APIC internal interrupt state (ISR, IRR, etc.).
- The microcode update state.

If an SMI is used to power down the processor, a power-on reset will be required before returning to SMM, which will reset much of this state back to its default values. So an SMI handler that is going to trigger power down should first read these registers listed above directly, and save them (along with the rest of RAM) to nonvolatile storage. After the power-on reset, the continuation of the SMI handler should restore these values, along with the rest of the system's state. Anytime the SMI handler changes these registers in the processor, it must also save and restore them.

### NOTES

A small subset of the MSRs (such as, the time-stamp counter and performance-monitoring counters) are not arbitrarily writable and therefore cannot be saved and restored. SMM-based power-down and restoration should only be performed with operating systems that do not use or rely on the values of these registers.

Operating system developers should be aware of this fact and ensure that their operating-system assisted power-down and restoration software is immune to unexpected changes in these register values.

#### 30.4.1.1 SMRAM State Save Map and Intel 64 Architecture

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area on an Intel 64 processor at [SMBASE + 8000H + 7FFFH] and extends to [SMBASE + 8000H + 7C00H].

Support for Intel 64 architecture is reported by CPUID.80000001:EDX[29] = 1. The layout of the SMRAM state save map is shown in Table 30-3.

Additionally, the SMRAM state save map shown in Table 30-3 also applies to processors with the following CPUID signatures listed in Table 30-2, irrespective of the value in CPUID.80000001:EDX[29].

**Table 30-2. Processor Signatures and 64-bit SMRAM State Save Map Format**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processor Q9xxx, Intel Core 2 Duo processors E8000, T9000,
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad, Intel Core 2 Extreme, Intel Core 2 Duo processors, Intel Pentium dual-core processors
06_1CH	45 nm Intel® Atom™ processors

Table 30-3. SMRAM State Save Map for Intel 64 Architecture

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FF8H	CR0	No
7FF0H	CR3	No
7FE8H	RFLAGS	Yes
7FE0H	IA32_EFER	Yes
7FD8H	RIP	Yes
7FD0H	DR6	No
7FC8H	DR7	No
7FC4H	TR SEL <sup>1</sup>	No
7FC0H	LDTR SEL <sup>1</sup>	No
7FBCH	GS SEL <sup>1</sup>	No
7FB8H	FS SEL <sup>1</sup>	No
7FB4H	DS SEL <sup>1</sup>	No
7FB0H	SS SEL <sup>1</sup>	No
7FACH	CS SEL <sup>1</sup>	No
7FA8H	ES SEL <sup>1</sup>	No
7FA4H	IO_MISC	No
7F9CH	IO_MEM_ADDR	No
7F94H	RDI	Yes
7F8CH	RSI	Yes
7F84H	RBP	Yes
7F7CH	RSP	Yes
7F74H	RBX	Yes
7F6CH	RDX	Yes
7F64H	RCX	Yes
7F5CH	RAX	Yes
7F54H	R8	Yes
7F4CH	R9	Yes
7F44H	R10	Yes
7F3CH	R11	Yes
7F34H	R12	Yes
7F2CH	R13	Yes
7F24H	R14	Yes
7F1CH	R15	Yes
7F1BH-7F04H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes

Table 30-3. SMRAM State Save Map for Intel 64 Architecture (Contd.)

Offset (Added to SMBASE + 8000H)	Register	Writable?
7EF7H - 7EE4H	Reserved	No
7EE0H	Setting of "enable EPT" VM-execution control	No
7ED8H	Value of EPTP VM-execution control field	No
7ED7H - 7ECC0H	Reserved	No
7EC8H	SSP	Yes
7EC7H - 7EA0H	Reserved	No
7E9CH	LDT Base (lower 32 bits)	No
7E98H	Reserved	No
7E94H	IDT Base (lower 32 bits)	No
7E90H	Reserved	No
7E8CH	GDT Base (lower 32 bits)	No
7E8BH - 7E48H	Reserved	No
7E40H	CR4 (64 bits)	No
7E3FH - 7DF0H	Reserved	No
7DE8H	IO_RIP	Yes
7DE7H - 7DDCH	Reserved	No
7DD8H	IDT Base (Upper 32 bits)	No
7DD4H	LDT Base (Upper 32 bits)	No
7DD0H	GDT Base (Upper 32 bits)	No
7DCFH - 7C00H	Reserved	No

**NOTE:**

1. The two most significant bytes are reserved.

### 30.4.2 SMRAM Caching

An IA-32 processor does not automatically write back and invalidate its caches before entering SMM or before exiting SMM. Because of this behavior, care must be taken in the placement of the SMRAM in system memory and in the caching of the SMRAM to prevent cache incoherence when switching back and forth between SMM and protected mode operation. Any of the following three methods of locating the SMRAM in system memory will guarantee cache coherency.

- Place the SMRAM in a dedicated section of system memory that the operating system and applications are prevented from accessing. Here, the SMRAM can be designated as cacheable (WB, WT, or WC) for optimum processor performance, without risking cache incoherence when entering or exiting SMM.
- Place the SMRAM in a section of memory that overlaps an area used by the operating system (such as the video memory), but designate the SMRAM as uncacheable (UC). This method prevents cache access when in SMM to maintain cache coherency, but the use of uncacheable memory reduces the performance of SMM code.
- Place the SMRAM in a section of system memory that overlaps an area used by the operating system and/or application code, but explicitly flush (write back and invalidate) the caches upon entering and exiting SMM mode. This method maintains cache coherency, but incurs the overhead of two complete cache flushes.

For Pentium 4, Intel Xeon, and P6 family processors, a combination of the first two methods of locating the SMRAM is recommended. Here the SMRAM is split between an overlapping and a dedicated region of memory. Upon entering SMM, the SMRAM space that is accessed overlaps video memory (typically located in low memory). This SMRAM section is designated as UC memory. The initial SMM code then jumps to a second SMRAM section that is

located in a dedicated region of system memory (typically in high memory). This SMRAM section can be cached for optimum processor performance.

For systems that explicitly flush the caches upon entering SMM (the third method described above), the cache flush can be accomplished by asserting the FLUSH# pin at the same time as the request to enter SMM (generally initiated by asserting the SMI# pin). The priorities of the FLUSH# and SMI# pins are such that the FLUSH# is serviced first. To guarantee this behavior, the processor requires that the following constraints on the interaction of FLUSH# and SMI# be met. In a system where the FLUSH# and SMI# pins are synchronous and the set up and hold times are met, then the FLUSH# and SMI# pins may be asserted in the same clock. In asynchronous systems, the FLUSH# pin must be asserted at least one clock before the SMI# pin to guarantee that the FLUSH# pin is serviced first.

Upon leaving SMM (for systems that explicitly flush the caches), the WBINVD instruction should be executed prior to leaving SMM to flush the caches.

## NOTES

In systems based on the Pentium processor that use the FLUSH# pin to write back and invalidate cache contents before entering SMM, the processor will prefetch at least one cache line in between when the Flush Acknowledge cycle is run and the subsequent recognition of SMI# and the assertion of SMIACK#.

It is the obligation of the system to ensure that these lines are not cached by returning KEN# inactive to the Pentium processor.

### 30.4.2.1 System Management Range Registers (SMRR)

SMI handler code and data stored by SMM code resides in SMRAM. The SMRR interface is an enhancement in Intel 64 architecture to limit cacheable reference of addresses in SMRAM to code running in SMM. The SMRR interface can be configured only by code running in SMM. Details of SMRR is described in Section 11.11.2.4.

## 30.5 SMI HANDLER EXECUTION ENVIRONMENT

Section 30.5.1 describes the initial execution environment for an SMI handler. An SMI handler may re-configure its execution environment to other supported operating modes. Section 30.5.2 discusses modifications an SMI handler can make to its execution environment. Section 30.5.3 discusses Control-flow Enforcement Technology (CET) interactions in the environment.

### 30.5.1 Initial SMM Execution Environment

After saving the current context of the processor, the processor initializes its core registers to the values shown in Table 30-4. Upon entering SMM, the PE and PG flags in control register CR0 are cleared, which places the processor in an environment similar to real-address mode. The differences between the SMM execution environment and the real-address mode execution environment are as follows:

- The addressable address space ranges from 0 to FFFFFFFFH (4 GBytes).
- The normal 64-KByte segment limit for real-address mode is increased to 4 GBytes.
- The default operand and address sizes are set to 16 bits, which restricts the addressable SMRAM address space to the 1-MByte real-address mode limit for native real-address-mode code. However, operand-size and address-size override prefixes can be used to access the address space beyond the 1-MByte.

**Table 30-4. Processor Register Initialization in SMM**

Register	Contents
General-purpose registers	Undefined
EFLAGS	00000002H
EIP	00008000H
CS selector	SMM Base shifted right 4 bits (default 3000H)



**Table 30-4. Processor Register Initialization in SMM**

CS base	SMM Base (default 30000H)
DS, ES, FS, GS, SS Selectors	0000H
DS, ES, FS, GS, SS Bases	000000000H
DS, ES, FS, GS, SS Limits	0FFFFFFFFH
CR0	PE, EM, TS, and PG flags set to 0; others unmodified
CR4	Cleared to zero
DR6	Undefined
DR7	00000400H

- Near jumps and calls can be made to anywhere in the 4-GByte address space if a 32-bit operand-size override prefix is used. Due to the real-address-mode style of base-address formation, a far call or jump cannot transfer control to a segment with a base address of more than 20 bits (1 MByte). However, since the segment limit in SMM is 4 GBytes, offsets into a segment that go beyond the 1-MByte limit are allowed when using 32-bit operand-size override prefixes. Any program control transfer that does not have a 32-bit operand-size override prefix truncates the EIP value to the 16 low-order bits.
- Data and the stack can be located anywhere in the 4-GByte address space, but can be accessed only with a 32-bit address-size override if they are located above 1 MByte. As with the code segment, the base address for a data or stack segment cannot be more than 20 bits.

The value in segment register CS is automatically set to the default of 30000H for the SMBASE shifted 4 bits to the right; that is, 3000H. The EIP register is set to 8000H. When the EIP value is added to shifted CS value (the SMBASE), the resulting linear address points to the first instruction of the SMI handler.

The other segment registers (DS, SS, ES, FS, and GS) are cleared to 0 and their segment limits are set to 4 GBytes. In this state, the SMRAM address space may be treated as a single flat 4-GByte linear address space. If a segment register is loaded with a 16-bit value, that value is then shifted left by 4 bits and loaded into the segment base (hidden part of the segment register). The limits and attributes are not modified.

Maskable hardware interrupts, exceptions, NMI interrupts, SMI interrupts, A20M interrupts, single-step traps, breakpoint traps, and INIT operations are inhibited when the processor enters SMM. Maskable hardware interrupts, exceptions, single-step traps, and breakpoint traps can be enabled in SMM if the SMM execution environment provides and initializes an interrupt table and the necessary interrupt and exception handlers (see Section 30.6).

### 30.5.2 SMI Handler Operating Mode Switching

Within SMM, an SMI handler may change the processor's operating mode (e.g., to enable PAE paging, enter 64-bit mode, etc.) after it has made proper preparation and initialization to do so. For example, if switching to 32-bit protected mode, the SMI handler should follow the guidelines provided in Chapter 9, "Processor Management and Initialization". If the SMI handler does wish to change operating mode, it is responsible for executing the appropriate mode-transition code after each SMI.

It is recommended that the SMI handler make use of all means available to protect the integrity of its critical code and data. In particular, it should use the system-management range register (SMRR) interface if it is available (see Section 11.11.2.4). The SMRR interface can protect only the first 4 GBytes of the physical address space. The SMI handler should take that fact into account if it uses operating modes that allow access to physical addresses beyond that 4-GByte limit (e.g. PAE paging or 64-bit mode).

Execution of the RSM instruction restores the pre-SMI processor state from the SMRAM state-state map (see Section 30.4.1) into which it was stored when the processor entered SMM. (The SMBASE field in the SMRAM state-save map does not determine the state following RSM but rather the initial environment following the next entry to SMM.) Any required change to operating mode is performed by the RSM instruction; there is no need for the SMI handler to change modes explicitly prior to executing RSM.

### 30.5.3 Control-flow Enforcement Technology Interactions

On processors that support CET shadow stacks, when the processor enters SMM, the processor saves the SSP register to the SMRAM state save area (see Table 30-3) and clears CR4.CET to 0. Thus, the initial execution environment of the SMI handler has CET disabled and all of the CET state of the interrupted program is still in the machine. An SMM that uses CET is required to save the interrupted program's CET state and restore the CET state prior to exiting SMM.

## 30.6 EXCEPTIONS AND INTERRUPTS WITHIN SMM

When the processor enters SMM, all hardware interrupts are disabled in the following manner:

- The IF flag in the EFLAGS register is cleared, which inhibits maskable hardware interrupts from being generated.
- The TF flag in the EFLAGS register is cleared, which disables single-step traps.
- Debug register DR7 is cleared, which disables breakpoint traps. (This action prevents a debugger from accidentally breaking into an SMI handler if a debug breakpoint is set in normal address space that overlays code or data in SMRAM.)
- NMI, SMI, and A20M interrupts are blocked by internal SMM logic. (See Section 30.8 for more information about how NMIs are handled in SMM.)

Software-invoked interrupts and exceptions can still occur, and maskable hardware interrupts can be enabled by setting the IF flag. Intel recommends that SMM code be written in so that it does not invoke software interrupts (with the INT *n*, INTO, INT1, INT3, or BOUND instructions) or generate exceptions.

If the SMI handler requires interrupt and exception handling, an SMM interrupt table and the necessary exception and interrupt handlers must be created and initialized from within SMM. Until the interrupt table is correctly initialized (using the LIDT instruction), exceptions and software interrupts will result in unpredictable processor behavior.

The following restrictions apply when designing SMM interrupt and exception-handling facilities:

- The interrupt table should be located at linear address 0 and must contain real-address mode style interrupt vectors (4 bytes containing CS and IP).
- Due to the real-address mode style of base address formation, an interrupt or exception cannot transfer control to a segment with a base address of more than 20 bits.
- An interrupt or exception cannot transfer control to a segment offset of more than 16 bits (64 KBytes).
- When an exception or interrupt occurs, only the 16 least-significant bits of the return address (EIP) are pushed onto the stack. If the offset of the interrupted procedure is greater than 64 KBytes, it is not possible for the interrupt/exception handler to return control to that procedure. (One solution to this problem is for a handler to adjust the return address on the stack.)
- The SMBASE relocation feature affects the way the processor will return from an interrupt or exception generated while the SMI handler is executing. For example, if the SMBASE is relocated to above 1 MByte, but the exception handlers are below 1 MByte, a normal return to the SMI handler is not possible. One solution is to provide the exception handler with a mechanism for calculating a return address above 1 MByte from the 16-bit return address on the stack, then use a 32-bit far call to return to the interrupted procedure.
- If an SMI handler needs access to the debug trap facilities, it must ensure that an SMM accessible debug handler is available and save the current contents of debug registers DR0 through DR3 (for later restoration). Debug registers DR0 through DR3 and DR7 must then be initialized with the appropriate values.
- If an SMI handler needs access to the single-step mechanism, it must ensure that an SMM accessible single-step handler is available, and then set the TF flag in the EFLAGS register.
- If the SMI design requires the processor to respond to maskable hardware interrupts or software-generated interrupts while in SMM, it must ensure that SMM accessible interrupt handlers are available and then set the IF flag in the EFLAGS register (using the STI instruction). Software interrupts are not blocked upon entry to SMM, so they do not need to be enabled.

## 30.7 MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS

When coding for a multiprocessor system or a system with Intel HT Technology, it was not always possible for an SMI handler to distinguish between a synchronous SMI (triggered during an I/O instruction) and an asynchronous SMI. To facilitate the discrimination of these two events, incremental state information has been added to the SMM state save map.

Processors that have an SMM revision ID of 30004H or higher have the incremental state information described below.

### 30.7.1 I/O State Implementation

Within the extended SMM state save map, a bit (IO\_SMI) is provided that is set only when an SMI is either taken immediately after a *successful* I/O instruction or is taken after a *successful* iteration of a REP I/O instruction (the *successful* notion pertains to the processor point of view; not necessarily to the corresponding platform function). When set, the IO\_SMI bit provides a strong indication that the corresponding SMI was synchronous. In this case, the SMM State Save Map also supplies the port address of the I/O operation. The IO\_SMI bit and the I/O Port Address may be used in conjunction with the information logged by the platform to confirm that the SMI was indeed synchronous.

The IO\_SMI bit by itself is a strong indication, not a guarantee, that the SMI is synchronous. This is because an asynchronous SMI might coincidentally be taken after an I/O instruction. In such a case, the IO\_SMI bit would still be set in the SMM state save map.

Information characterizing the I/O instruction is saved in two locations in the SMM State Save Map (Table 30-5). The IO\_SMI bit also serves as a valid bit for the rest of the I/O information fields. The contents of these I/O information fields are not defined when the IO\_SMI bit is not set.

**Table 30-5. I/O Instruction Information in the SMM State Save Map**

State (SMM Rev. ID: 30004H or higher)	Format								
	31	16	15	8	7	4	3	1	0
I/O State Field SMRAM offset 7FA4		I/O Port	Reserved		I/O Type		I/O Length		IO_SMI
	31								0
I/O Memory Address Field SMRAM offset 7FA0	I/O Memory Address								

When IO\_SMI is set, the other fields may be interpreted as follows:

- I/O length:
  - 001 – Byte
  - 010 – Word
  - 100 – Dword
- I/O instruction type (Table 30-6)

**Table 30-6. I/O Instruction Type Encodings**

Instruction	Encoding
IN Immediate	1001
IN DX	0001
OUT Immediate	1000

Table 30-6. I/O Instruction Type Encodings (Contd.)

Instruction	Encoding
OUT DX	0000
INS	0011
OUTS	0010
REP INS	0111
REP OUTS	0110

## 30.8 NMI HANDLING WHILE IN SMM

NMI interrupts are blocked upon entry to the SMI handler. If an NMI request occurs during the SMI handler, it is latched and serviced after the processor exits SMM. Only one NMI request will be latched during the SMI handler. If an NMI request is pending when the processor executes the RSM instruction, the NMI is serviced before the next instruction of the interrupted code sequence. This assumes that NMIs were not blocked before the SMI occurred. If NMIs were blocked before the SMI occurred, they are blocked after execution of RSM.

Although NMI requests are blocked when the processor enters SMM, they may be enabled through software by executing an IRET instruction. If the SMI handler requires the use of NMI interrupts, it should invoke a dummy interrupt service routine for the purpose of executing an IRET instruction. Once an IRET instruction is executed, NMI interrupt requests are serviced in the same "real mode" manner in which they are handled outside of SMM.

Also, for the Pentium processor, exceptions that invoke a trap or fault handler will enable NMI interrupts from inside of SMM. This behavior is implementation specific for the Pentium processor and is not part of the IA-32 architecture.

## 30.9 SMM REVISION IDENTIFIER

The SMM revision identifier field is used to indicate the version of SMM and the SMM extensions that are supported by the processor (see Figure 30-2). The SMM revision identifier is written during SMM entry and can be examined in SMRAM space at offset 7EFCH. The lower word of the SMM revision identifier refers to the version of the base SMM architecture.

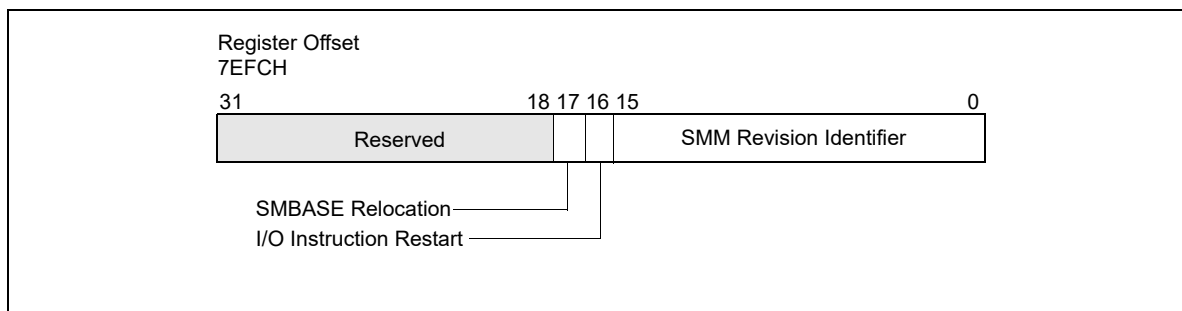


Figure 30-2. SMM Revision Identifier

The upper word of the SMM revision identifier refers to the extensions available. If the I/O instruction restart flag (bit 16) is set, the processor supports the I/O instruction restart (see Section 30.12); if the SMBASE relocation flag (bit 17) is set, SMRAM base address relocation is supported (see Section 30.11).

## 30.10 AUTO HALT RESTART

If the processor is in a HALT state (due to the prior execution of a HLT instruction) when it receives an SMI, the processor records the fact in the auto HALT restart flag in the saved processor state (see Figure 30-3). (This flag is located at offset 7F02H and bit 0 in the state save area of the SMRAM.)

If the processor sets the auto HALT restart flag upon entering SMM (indicating that the SMI occurred when the processor was in the HALT state), the SMI handler has two options:

- It can leave the auto HALT restart flag set, which instructs the RSM instruction to return program control to the HLT instruction. This option in effect causes the processor to re-enter the HALT state after handling the SMI. (This is the default operation.)
- It can clear the auto HALT restart flag, which instructs the RSM instruction to return program control to the instruction following the HLT instruction.

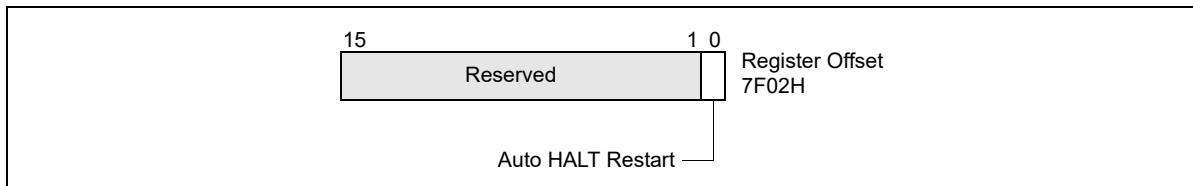


Figure 30-3. Auto HALT Restart Field

These options are summarized in Table 30-7. If the processor was not in a HALT state when the SMI was received (the auto HALT restart flag is cleared), setting the flag to 1 will cause unpredictable behavior when the RSM instruction is executed.

Table 30-7. Auto HALT Restart Flag Values

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
0	0	Returns to next instruction in interrupted program or task.
0	1	Unpredictable.
1	0	Returns to next instruction after HLT instruction.
1	1	Returns to HALT state.

If the HLT instruction is restarted, the processor will generate a memory access to fetch the HLT instruction (if it is not in the internal cache), and execute a HLT bus transaction. This behavior results in multiple HLT bus transactions for the same HLT instruction.

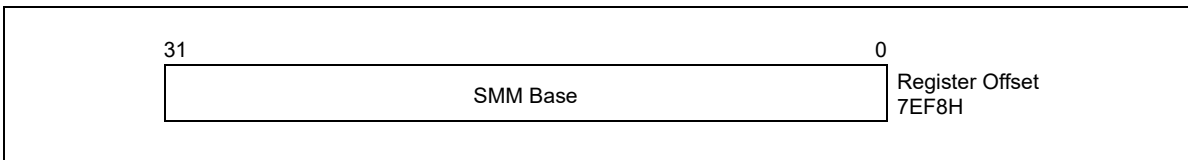
### 30.10.1 Executing the HLT Instruction in SMM

The HLT instruction should not be executed during SMM, unless interrupts have been enabled by setting the IF flag in the EFLAGS register. If the processor is halted in SMM, the only event that can remove the processor from this state is a maskable hardware interrupt or a hardware reset.

## 30.11 SMBASE RELOCATION

The default base address for the SMRAM is 30000H. This value is contained in an internal processor register called the SMBASE register. The operating system or executive can relocate the SMRAM by setting the SMBASE field in the saved state map (at offset 7EF8H) to a new value (see Figure 30-4). The RSM instruction reloads the internal SMBASE register with the value in the SMBASE field each time it exits SMM. All subsequent SMI requests will use the new SMBASE value to find the starting address for the SMI handler (at SMBASE + 8000H) and the SMRAM state

save area (from SMBASE + FE00H to SMBASE + FFFFH). (The processor resets the value in its internal SMBASE register to 30000H on a RESET, but does not change it on an INIT.)



**Figure 30-4. SMBASE Relocation Field**

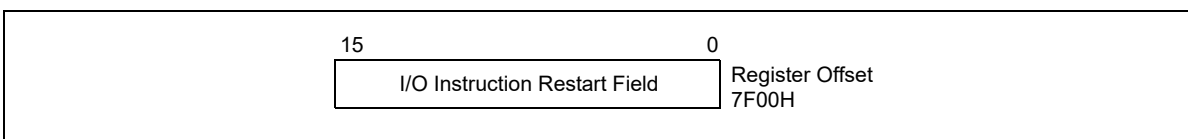
In multiple-processor systems, initialization software must adjust the SMBASE value for each processor so that the SMRAM state save areas for each processor do not overlap. (For Pentium and Intel486 processors, the SMBASE values must be aligned on a 32-KByte boundary or the processor will enter shutdown state during the execution of a RSM instruction.)

If the SMBASE relocation flag in the SMM revision identifier field is set, it indicates the ability to relocate the SMBASE (see Section 30.9).

## 30.12 I/O INSTRUCTION RESTART

If the I/O instruction restart flag in the SMM revision identifier field is set (see Section 30.9), the I/O instruction restart mechanism is present on the processor. This mechanism allows an interrupted I/O instruction to be re-executed upon returning from SMM mode. For example, if an I/O instruction is used to access a powered-down I/O device, a chip set supporting this device can intercept the access and respond by asserting SMI#. This action invokes the SMI handler to power-up the device. Upon returning from the SMI handler, the I/O instruction restart mechanism can be used to re-execute the I/O instruction that caused the SMI.

The I/O instruction restart field (at offset 7F00H in the SMM state-save area, see Figure 30-5) controls I/O instruction restart. When an RSM instruction is executed, if this field contains the value FFH, then the EIP register is modified to point to the I/O instruction that received the SMI request. The processor will then automatically re-execute the I/O instruction that the SMI trapped. (The processor saves the necessary machine state to ensure that re-execution of the instruction is handled coherently.)



**Figure 30-5. I/O Instruction Restart Field**

If the I/O instruction restart field contains the value 00H when the RSM instruction is executed, then the processor begins program execution with the instruction following the I/O instruction. (When a repeat prefix is being used, the next instruction may be the next I/O instruction in the repeat loop.) Not re-executing the interrupted I/O instruction is the default behavior; the processor automatically initializes the I/O instruction restart field to 00H upon entering SMM. Table 30-8 summarizes the states of the I/O instruction restart field.

**Table 30-8. I/O Instruction Restart Field Values**

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
00H	00H	Does not re-execute trapped I/O instruction.
00H	FFH	Re-executes trapped I/O instruction.

The I/O instruction restart mechanism does not indicate the cause of the SMI. It is the responsibility of the SMI handler to examine the state of the processor to determine the cause of the SMI and to determine if an I/O instruction was interrupted and should be restarted upon exiting SMM. If an SMI interrupt is signaled on a non-I/O instruction boundary, setting the I/O instruction restart field to FFH prior to executing the RSM instruction will likely result in a program error.

### 30.12.1 Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used

If an SMI interrupt is signaled while the processor is servicing an SMI interrupt that occurred on an I/O instruction boundary, the processor will service the new SMI request before restarting the originally interrupted I/O instruction. If the I/O instruction restart field is set to FFH prior to returning from the second SMI handler, the EIP will point to an address different from the originally interrupted I/O instruction, which will likely lead to a program error. To avoid this situation, the SMI handler must be able to recognize the occurrence of back-to-back SMI interrupts when I/O instruction restart is being used and ensure that the handler sets the I/O instruction restart field to 00H prior to returning from the second invocation of the SMI handler.

## 30.13 SMM MULTIPLE-PROCESSOR CONSIDERATIONS

The following should be noted when designing multiple-processor systems:

- Any processor in a multiprocessor system can respond to an SMM.
- Each processor needs its own SMRAM space. This space can be in system memory or in a separate RAM.
- The SMRAMs for different processors can be overlapped in the same memory space. The only stipulation is that each processor needs its own state save area and its own dynamic data storage area. (Also, for the Pentium and Intel486 processors, the SMBASE address must be located on a 32-KByte boundary.) Code and static data can be shared among processors. Overlapping SMRAM spaces can be done more efficiently with the P6 family processors because they do not require that the SMBASE address be on a 32-KByte boundary.
- The SMI handler will need to initialize the SMBASE for each processor.
- Processors can respond to local SMIs through their SMI# pins or to SMIs received through the APIC interface. The APIC interface can distribute SMIs to different processors.
- Two or more processors can be executing in SMM at the same time.
- When operating Pentium processors in dual processing (DP) mode, the SMIACK# pin is driven only by the MRM processor and should be sampled with ADS#. For additional details, see Chapter 14 of the *Pentium Processor Family User's Manual, Volume 1*.

SMM is not re-entrant, because the SMRAM State Save Map is fixed relative to the SMBASE. If there is a need to support two or more processors in SMM mode at the same time then each processor should have dedicated SMRAM spaces. This can be done by using the SMBASE Relocation feature (see Section 30.11).

## 30.14 DEFAULT TREATMENT OF SMIS AND SMM WITH VMX OPERATION AND SMX OPERATION

Under the default treatment, the interactions of SMIs and SMM with VMX operation are few. This section details those interactions. It also explains how this treatment affects SMX operation.



### 30.14.1 Default Treatment of SMI Delivery

Ordinary SMI delivery saves processor state into SMRAM and then loads state based on architectural definitions. Under the default treatment, processors that support VMX operation perform SMI delivery as follows:

```

enter SMM;
save the following internal to the processor:
    CR4.VMXE
    an indication of whether the logical processor was in VMX operation (root or non-root)
IF the logical processor is in VMX operation
    THEN
        save current VMCS pointer internal to the processor;
        leave VMX operation;
        save VMX-critical state defined below;
FI;
IF the logical processor supports SMX operation
    THEN
        save internal to the logical processor an indication of whether the Intel® TXT private space is locked;
        IF the TXT private space is unlocked
            THEN lock the TXT private space;
        FI;
FI;
CR4.VMXE := 0;
perform ordinary SMI delivery:
    save processor state in SMRAM;
    set processor state to standard SMM values;1
    invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H
    are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 27.3);

```

The pseudocode above makes reference to the saving of **VMX-critical state**. This state consists of the following: (1) SS.DPL (the current privilege level); (2) RFLAGS.VM<sup>2</sup>; (3) the state of blocking by STI and by MOV SS (see Table 23-3 in Section 23.4.2); (4) the state of virtual-NMI blocking (only if the processor is in VMX non-root operation and the “virtual NMIs” VM-execution control is 1); and (5) an indication of whether an MTF VM exit is pending (see Section 24.5.2). These data may be saved internal to the processor or in the VMCS region of the current VMCS. Processors that do not support SMI recognition while there is blocking by STI or by MOV SS need not save the state of such blocking.

If the logical processor supports the 1-setting of the “enable EPT” VM-execution control and the logical processor was in VMX non-root operation at the time of an SMI, it saves the value of that control into bit 0 of the 32-bit field at offset SMBASE + 8000H + 7EE0H (SMBASE + FEE0H; see Table 30-3).<sup>3</sup> If the logical processor was not in VMX non-root operation at the time of the SMI, it saves 0 into that bit. If the logical processor saves 1 into that bit (it was in VMX non-root operation and the “enable EPT” VM-execution control was 1), it saves the value of the EPT pointer (EPTP) into the 64-bit field at offset SMBASE + 8000H + 7ED8H (SMBASE + FED8H).

Because SMI delivery causes a logical processor to leave VMX operation, all the controls associated with VMX non-root operation are disabled in SMM and thus cannot cause VM exits while the logical processor in SMM.

- 
1. This causes the logical processor to block INIT signals, NMIs, and SMIs.
  2. Section 30.14 and Section 30.15 use the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of these registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to the lower 32 bits of the register.
  3. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, SMI functions as the “enable EPT” VM-execution control were 0. See Section 23.6.2.



### 30.14.2 Default Treatment of RSM

Ordinary execution of RSM restores processor state from SMRAM. Under the default treatment, processors that support VMX operation perform RSM as follows:

```

IF VMXE = 1 in CR4 image in SMRAM
  THEN fail and enter shutdown state;
  ELSE
    restore state normally from SMRAM;
    invalidate linear mappings and combined mappings associated with all VPIDs and all PCIDs; combined mappings are invalidated
    for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 27.3);
    IF the logical processor supports SMX operation and the Intel® TXT private space was unlocked at the time of the last SMI (as
    saved)
      THEN unlock the TXT private space;
    FI;
    CR4.VMXE := value stored internally;
    IF internal storage indicates that the logical processor
    had been in VMX operation (root or non-root)
      THEN
        enter VMX operation (root or non-root);
        restore VMX-critical state as defined in Section 30.14.1;
        set to their fixed values any bits in CR0 and CR4 whose values must be fixed in VMX operation (see Section 22.8);1
        IF RFLAGS.VM = 0 AND (in VMX root operation OR the “unrestricted guest” VM-execution control is 0)2
          THEN
            CS.RPL := SS.DPL;
            SS.RPL := SS.DPL;
          FI;
        restore current VMCS pointer;
      FI;
    leave SMM;
    IF logical processor will be in VMX operation or in SMX operation after RSM
      THEN block A20M and leave A20M mode;
    FI;
  FI;

```

RSM unblocks SMIs. It restores the state of blocking by NMI (see Table 23-3 in Section 23.4.2) as follows:

- If the RSM is not to VMX non-root operation or if the “virtual NMIs” VM-execution control will be 0, the state of NMI blocking is restored normally.
- If the RSM is to VMX non-root operation and the “virtual NMIs” VM-execution control will be 1, NMIs are not blocked after RSM. The state of virtual-NMI blocking is restored as part of VMX-critical state.

INIT signals are blocked after RSM if and only if the logical processor will be in VMX root operation.

If RSM returns a logical processor to VMX non-root operation, it re-establishes the controls associated with the current VMCS. If the “interrupt-window exiting” VM-execution control is 1, a VM exit occurs immediately after RSM if the enabling conditions apply. The same is true for the “NMI-window exiting” VM-execution control. Such VM exits occur with their normal priority. See Section 24.2.

If an MTF VM exit was pending at the time of the previous SMI, an MTF VM exit is pending on the instruction boundary following execution of RSM. The following items detail the treatment of MTF VM exits that may be pending following RSM:

1. If the RSM is to VMX non-root operation and both the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls will be 1, CR0.PE and CR0.PG retain the values that were loaded from SMRAM regardless of what is reported in the capability MSR IA32\_VMX\_CRO\_FIXED0.
2. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 23.6.2.

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these MTF VM exits. These MTF VM exits take priority over debug-trap exceptions and lower priority events.
- These MTF VM exits wake the logical processor if RSM caused the logical processor to enter the HLT state (see Section 30.10). They do not occur if the logical processor just entered the shutdown state.

### 30.14.3 Protection of CR4.VMXE in SMM

Under the default treatment, CR4.VMXE is treated as a reserved bit while a logical processor is in SMM. Any attempt by software running in SMM to set this bit causes a general-protection exception. In addition, software cannot use VMX instructions or enter VMX operation while in SMM.

### 30.14.4 VMXOFF and SMI Unblocking

The VMXOFF instruction can be executed only with the default treatment (see Section 30.15.1) and only outside SMM. If SMIs are blocked when VMXOFF is executed, VMXOFF unblocks them unless IA32\_SMM\_MONITOR\_CTL[bit 2] is 1 (see Section 30.15.5 for details regarding this MSR).<sup>1</sup> Section 30.15.7 identifies a case in which SMIs may be blocked when VMXOFF is executed.

Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6) to determine whether this is allowed.

## 30.15 DUAL-MONITOR TREATMENT OF SMIs AND SMM

Dual-monitor treatment is activated through the cooperation of the **executive monitor** (the VMM that operates outside of SMM to provide basic virtualization) and the **SMM-transfer monitor (STM)**; the VMM that operates inside SMM—while in VMX operation—to support system-management functions). Control is transferred to the STM through VM exits; VM entries are used to return from SMM.

The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether it is supported.

### 30.15.1 Dual-Monitor Treatment Overview

The dual-monitor treatment uses an executive monitor and an SMM-transfer monitor (STM). Transitions from the executive monitor or its guests to the STM are called **SMM VM exits** and are discussed in Section 30.15.2. SMM VM exits are caused by SMIs as well as executions of VMCALL in VMX root operation. The latter allow the executive monitor to call the STM for service.

The STM runs in VMX root operation and uses VMX instructions to establish a VMCS and perform VM entries to its own guests. This is done all inside SMM (see Section 30.15.3). The STM returns from SMM, not by using the RSM instruction, but by using a VM entry that returns from SMM. Such VM entries are described in Section 30.15.4.

Initially, there is no STM and the default treatment (Section 30.14) is used. The dual-monitor treatment is not used until it is enabled and activated. The steps to do this are described in Section 30.15.5 and Section 30.15.6.

It is not possible to leave VMX operation under the dual-monitor treatment; VMXOFF will fail if executed. The dual-monitor treatment must be deactivated first. The STM deactivates dual-monitor treatment using a VM entry that returns from SMM with the “deactivate dual-monitor treatment” VM-entry control set to 1 (see Section 30.15.7).

The executive monitor configures any VMCS that it uses for VM exits to the executive monitor. SMM VM exits, which transfer control to the STM, use a different VMCS. Under the dual-monitor treatment, each logical processor uses a separate VMCS called the **SMM-transfer VMCS**. When the dual-monitor treatment is active, the logical processor maintains another VMCS pointer called the **SMM-transfer VMCS pointer**. The SMM-transfer VMCS pointer is established when the dual-monitor treatment is activated.

1. Setting IA32\_SMM\_MONITOR\_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register’s valid bit (bit 0).

## 30.15.2 SMM VM Exits

An SMM VM exit is a VM exit that begins outside SMM and that ends in SMM.

Unlike other VM exits, SMM VM exits can begin in VMX root operation. SMM VM exits result from the arrival of an SMI outside SMM or from execution of VMCALL in VMX root operation outside SMM. Execution of VMCALL in VMX root operation causes an SMM VM exit only if the valid bit is set in the IA32\_SMM\_MONITOR\_CTL MSR (see Section 30.15.5).

Execution of VMCALL in VMX root operation causes an SMM VM exit even under the default treatment. This SMM VM exit activates the dual-monitor treatment (see Section 30.15.6).

Differences between SMM VM exits and other VM exits are detailed in Sections 30.15.2.1 through 30.15.2.5. Differences between SMM VM exits that activate the dual-monitor treatment and other SMM VM exits are described in Section 30.15.6.

### 30.15.2.1 Architectural State Before a VM Exit

System-management interrupts (SMIs) that cause SMM VM exits always do so directly. They do not save state to SMRAM as they do under the default treatment.

### 30.15.2.2 Updating the Current-VMCS and Executive-VMCS Pointers

SMM VM exits begin by performing the following steps:

1. The executive-VMCS pointer field in the SMM-transfer VMCS is loaded as follows:
  - If the SMM VM exit commenced in VMX non-root operation, it receives the current-VMCS pointer.
  - If the SMM VM exit commenced in VMX root operation, it receives the VMXON pointer.
2. The current-VMCS pointer is loaded with the value of the SMM-transfer VMCS pointer.

The last step ensures that the current VMCS is the SMM-transfer VMCS. VM-exit information is recorded in that VMCS, and VM-entry control fields in that VMCS are updated. State is saved into the guest-state area of that VMCS. The VM-exit controls and host-state area of that VMCS determine how the VM exit operates.

### 30.15.2.3 Recording VM-Exit Information

SMM VM exits differ from other VM exit with regard to the way they record VM-exit information. The differences follow.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. The field is loaded with the reason for the SMM VM exit: I/O SMI (an SMI arrived immediately after retirement of an I/O instruction), other SMI, or VMCALL. See Appendix C, “VMX Basic Exit Reasons”.
  - SMM VM exits are the only VM exits that may occur in VMX root operation. Because the SMM-transfer monitor may need to know whether it was invoked from VMX root or VMX non-root operation, this information is stored in bit 29 of the exit-reason field (see Table 23-16 in Section 23.9.1). The bit is set by SMM VM exits from VMX root operation.
  - If the SMM VM exit occurred in VMX non-root operation and an MTF VM exit was pending, bit 28 of the exit-reason field is set; otherwise, it is cleared.
  - Bits 27:16 and bits 31:30 are cleared.
- **Exit qualification.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, the exit qualification contains information about the I/O instruction that retired immediately before the SMI. It has the format given in Table 30-9.
- **Guest linear address.** This field is used for VM exits due to SMIs that arrive immediately after the retirement of an INS or OUTS instruction for which the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) is usable. The field receives the value of the linear address generated by ES:(E)DI (for INS) or segment:(E)SI (for OUTS; the default segment is DS but can be overridden by a segment override

**Table 30-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction**

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte  Other values not used.
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)
15:7	Reserved (cleared to 0)
31:16	Port number (as specified in the I/O instruction)
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

prefix) at the time the instruction started. If the relevant segment is not usable, the value is undefined. On processors that support Intel 64 architecture, bits 63:32 are clear if the logical processor was not in 64-bit mode before the VM exit.

- **I/O RCX, I/O RSI, I/O RDI, and I/O RIP.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, these fields receive the values that were in RCX, RSI, RDI, and RIP, respectively, before the I/O instruction executed. Thus, the value saved for I/O RIP addresses the I/O instruction.

### 30.15.2.4 Saving Guest State

SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area.

The value of the VMX-preemption timer is saved into the corresponding field in the guest-state area if the “save VMX-preemption timer value” VM-exit control is 1. That field becomes undefined if, in addition, either the SMM VM exit is from VMX root operation or the SMM VM exit is from VMX non-root operation and the “activate VMX-preemption timer” VM-execution control is 0.

### 30.15.2.5 Updating State

If an SMM VM exit is from VMX non-root operation and the “Intel PT uses guest physical addresses” VM-execution control is 1, the IA32\_RTIT\_CTL MSR is cleared to 00000000\_00000000H.<sup>1</sup> This is done even if the “clear IA32\_RTIT\_CTL” VM-exit control is 0.

SMM VM exits affect the non-register state of a logical processor as follows:

- SMM VM exits cause non-maskable interrupts (NMIs) to be blocked; they may be unblocked through execution of IRET or through a VM entry (depending on the value loaded for the interruptibility state and the setting of the “virtual NMIs” VM-execution control).
- SMM VM exits cause SMIs to be blocked; they may be unblocked by a VM entry that returns from SMM (see Section 30.15.4).

1. In this situation, the value of this MSR was saved earlier into the guest-state area. All VM exits save this MSR if the 1-setting of the “load IA32\_RTIT\_CTL” VM-entry control is supported (see Section 26.3.1), which must be the case if the “Intel PT uses guest physical addresses” VM-execution control is 1 (see Section 25.2.1.1).

SMM VM exits invalidate linear mappings and combined mappings associated with VPID 0000H for all PCIDs. Combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 27.3). (Ordinary VM exits are not required to perform such invalidation if the “enable VPID” VM-execution control is 1; see Section 26.5.5.)

### 30.15.3 Operation of the SMM-Transfer Monitor

Once invoked, the SMM-transfer monitor (STM) is in VMX root operation and can use VMX instructions to configure VMCSs and to cause VM entries to virtual machines supported by those structures. As noted in Section 30.15.1, the VMXOFF instruction cannot be used under the dual-monitor treatment and thus cannot be used by the STM.

The RSM instruction also cannot be used under the dual-monitor treatment. As noted in Section 24.1.3, it causes a VM exit if executed in SMM in VMX non-root operation. If executed in VMX root operation, it causes an invalid-opcode exception. The STM uses VM entries to return from SMM (see Section 30.15.4).

### 30.15.4 VM Entries that Return from SMM

The SMM-transfer monitor (STM) returns from SMM using a VM entry with the “entry to SMM” VM-entry control clear. VM entries that return from SMM reverse the effects of an SMM VM exit (see Section 30.15.2).

VM entries that return from SMM may differ from other VM entries in that they do not necessarily enter VMX non-root operation. If the executive-VMCS pointer field in the current VMCS contains the VMXON pointer, the logical processor remains in VMX root operation after VM entry.

For differences between VM entries that return from SMM and other VM entries see Sections 30.15.4.1 through 30.15.4.10.

#### 30.15.4.1 Checks on the Executive-VMCS Pointer Field

VM entries that return from SMM perform the following checks on the executive-VMCS pointer field in the current VMCS:

- Bits 11:0 must be 0.
- The pointer must not set any bits beyond the processor’s physical-address width.<sup>1,2</sup>
- The 32 bits located in memory referenced by the physical address in the pointer must contain the processor’s VMCS revision identifier (see Section 23.2).

The checks above are performed before the checks described in Section 30.15.4.2 and before any of the following checks:

- If the “deactivate dual-monitor treatment” VM-entry control is 0 and the executive-VMCS pointer field does not contain the VMXON pointer, the launch state of the executive VMCS (the VMCS referenced by the executive-VMCS pointer field) must be launched (see Section 23.11.3).
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the executive-VMCS pointer field must contain the VMXON pointer (see Section 30.15.7).<sup>3</sup>

#### 30.15.4.2 Checks on VM-Execution Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-execution control fields specified in Section 25.2.1.1. They do not apply the checks to the current VMCS. Instead, VM-entry behavior depends on whether the executive-VMCS pointer field contains the VMXON pointer:

- 
1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
  2. If IA32\_VMX\_BASIC[48] is read as 1, this pointer must not set any bits in the range 63:32; see Appendix A.1.
  3. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are not performed at all.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the checks are performed on the VM-execution control fields in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS). These checks are performed after checking the executive-VMCS pointer field itself (for proper alignment).

Other VM entries ensure that, if “activate VMX-preemption timer” VM-execution control is 0, the “save VMX-preemption timer value” VM-execution control is also 0. This check is not performed by VM entries that return from SMM.

### 30.15.4.3 Checks on VM-Entry Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-entry control fields specified in Section 25.2.1.3.

Specifically, if the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the VM-entry interruption-information field must not indicate injection of a pending MTF VM exit (see Section 25.6.2). Specifically, the following cannot all be true for that field:

- the valid bit (bit 31) is 1
- the interruption type (bits 10:8) is 7 (other event); and
- the vector (bits 7:0) is 0 (pending MTF VM exit).

### 30.15.4.4 Checks on the Guest State Area

Section 25.3.1 specifies checks performed on fields in the guest-state area of the VMCS. Some of these checks are conditioned on the settings of certain VM-execution controls (e.g., “virtual NMIs” or “unrestricted guest”).

VM entries that return from SMM modify these checks based on whether the executive-VMCS pointer field contains the VMXON pointer:<sup>1</sup>

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are performed as all relevant VM-execution controls were 0. (As a result, some checks may not be performed at all.)
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), this check is performed based on the settings of the VM-execution controls in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS).

For VM entries that return from SMM, the activity-state field must not indicate the wait-for-SIPI state if the executive-VMCS pointer field contains the VMXON pointer (the VM entry is to VMX root operation).

### 30.15.4.5 Loading Guest State

VM entries that return from SMM load the SMBASE register from the SMBASE field.

VM entries that return from SMM invalidate linear mappings and combined mappings associated with all VPIDs. Combined mappings are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 27.3). (Ordinary VM entries are required to perform such invalidation only for VPID 0000H and are not required to do even that if the “enable VPID” VM-execution control is 1; see Section 25.3.2.5.)

### 30.15.4.6 VMX-Preemption Timer

A VM entry that returns from SMM activates the VMX-preemption timer only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation) and the “activate VMX-preemption timer” VM-execution control is 1 in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field). In this case, VM entry starts the VMX-preemption timer with the value in the VMX-preemption timer-value field in the current VMCS.

---

1. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

### 30.15.4.7 Updating the Current-VMCS and SMM-Transfer VMCS Pointers

Successful VM entries (returning from SMM) load the SMM-transfer VMCS pointer with the current-VMCS pointer. Following this, they load the current-VMCS pointer from a field in the current VMCS:

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the current-VMCS pointer is loaded from the VMCS-link pointer field.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the current-VMCS pointer is loaded with the value of the executive-VMCS pointer field.

If the VM entry successfully enters VMX non-root operation, the VM-execution controls in effect after the VM entry are those from the new current VMCS. This includes any structures external to the VMCS referenced by VM-execution control fields.

The updating of these VMCS pointers occurs before event injection. Event injection is determined, however, by the VM-entry control fields in the VMCS that was current when the VM entry commenced.

### 30.15.4.8 VM Exits Induced by VM Entry

Section 25.6.1.2 describes how the event-delivery process invoked by event injection may lead to a VM exit. Section 25.7.3 to Section 25.7.7 describe other situations that may cause a VM exit to occur immediately after a VM entry.

Whether these VM exits occur is determined by the VM-execution control fields in the current VMCS. For VM entries that return from SMM, they can occur only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation).

In this case, determination is based on the VM-execution control fields in the VMCS that is current after the VM entry. This is the VMCS referenced by the value of the executive-VMCS pointer field at the time of the VM entry (see Section 30.15.4.7). This VMCS also controls the delivery of such VM exits. Thus, VM exits induced by a VM entry returning from SMM are to the executive monitor and not to the STM.

### 30.15.4.9 SMI Blocking

VM entries that return from SMM determine the blocking of system-management interrupts (SMIs) as follows:

- If the “deactivate dual-monitor treatment” VM-entry control is 0, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the blocking of SMIs depends on whether the logical processor is in SMX operation:<sup>1</sup>
  - If the logical processor is in SMX operation, SMIs are blocked after VM entry.
  - If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

VM entries that return from SMM and that do not deactivate the dual-monitor treatment may leave SMIs blocked. This feature exists to allow the STM to invoke functionality outside of SMM without unblocking SMIs.

### 30.15.4.10 Failures of VM Entries That Return from SMM

Section 25.8 describes the treatment of VM entries that fail during or after loading guest state. Such failures record information in the VM-exit information fields and load processor state as would be done on a VM exit. The VMCS used is the one that was current before the VM entry commenced. Control is thus transferred to the STM and the logical processor remains in SMM.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*.



### 30.15.5 Enabling the Dual-Monitor Treatment

Code and data for the SMM-transfer monitor (STM) reside in a region of SMRAM called the **monitor segment** (MSEG). Code running in SMM determines the location of MSEG and establishes its content. This code is also responsible for enabling the dual-monitor treatment.

SMM code enables the dual-monitor treatment and specifies the location of MSEG by writing to the IA32\_SMM\_MONITOR\_CTL MSR (index 9BH). The MSR has the following format:

- Bit 0 is the register's valid bit. The STM may be invoked using VMCALL only if this bit is 1. Because VMCALL is used to activate the dual-monitor treatment (see Section 30.15.6), the dual-monitor treatment cannot be activated if the bit is 0. This bit is cleared when the logical processor is reset.
- Bit 1 is reserved.
- Bit 2 determines whether executions of VMXOFF unblock SMIs under the default treatment of SMIs and SMM. Executions of VMXOFF unblock SMIs unless bit 2 is 1 (the value of bit 0 is irrelevant). See Section 30.14.4. Certain leaf functions of the GETSEC instruction clear this bit (see Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*).
- Bits 11:3 are reserved.
- Bits 31:12 contain a value that, when shifted left 12 bits, is the physical address of MSEG (the MSEG base address).
- Bits 63:32 are reserved.

The following items detail use of this MSR:

- The IA32\_SMM\_MONITOR\_CTL MSR is supported only on processors that support the dual-monitor treatment.<sup>1</sup> On other processors, accesses to the MSR using RDMSR or WRMSR generate a general-protection fault (#GP(0)).
- A write to the IA32\_SMM\_MONITOR\_CTL MSR using WRMSR generates a general-protection fault (#GP(0)) if executed outside of SMM or if an attempt is made to set any reserved bit. An attempt to write to the IA32\_SMM\_MONITOR\_CTL MSR fails if made as part of a VM exit that does not end in SMM or part of a VM entry that does not begin in SMM.
- Reads from the IA32\_SMM\_MONITOR\_CTL MSR using RDMSR are allowed any time RDMSR is allowed. The MSR may be read as part of any VM exit.
- The dual-monitor treatment can be activated only if the valid bit in the MSR is set to 1.

The 32 bytes located at the MSEG base address are called the **MSEG header**. The format of the MSEG header is given in Table 30-10 (each field is 32 bits).

**Table 30-10. Format of MSEG Header**

Byte Offset	Field
0	MSEG-header revision identifier
4	SMM-transfer monitor features
8	GDTR limit
12	GDTR base offset
16	CS selector
20	EIP offset
24	ESP offset
28	CR3 offset

1. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.



To ensure proper behavior in VMX operation, software should maintain the MSEG header in writeback cacheable memory. Future implementations may allow or require a different memory type.<sup>1</sup> Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1).

SMM code should enable the dual-monitor treatment (by setting the valid bit in IA32\_SMM\_MONITOR\_CTL MSR) only after establishing the content of the MSEG header as follows:

- Bytes 3:0 contain the **MSEG revision identifier**. Different processors may use different MSEG revision identifiers. These identifiers enable software to avoid using an MSEG header formatted for one processor on a processor that uses a different format. Software can discover the MSEG revision identifier that a processor uses by reading the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6).
- Bytes 7:4 contain the **SMM-transfer monitor features** field. Bits 31:1 of this field are reserved and must be zero. Bit 0 of the field is the **IA-32e mode SMM feature bit**. It indicates whether the logical processor will be in IA-32e mode after the STM is activated (see Section 30.15.6).
- Bytes 31:8 contain fields that determine how processor state is loaded when the STM is activated (see Section 30.15.6.5). SMM code should establish these fields so that activating of the STM invokes the STM's initialization code.

### 30.15.6 Activating the Dual-Monitor Treatment

The dual-monitor treatment may be enabled by SMM code as described in Section 30.15.5. The dual-monitor treatment is activated only if it is enabled and only by the executive monitor. The executive monitor activates the dual-monitor treatment by executing VMCALL in VMX root operation.

When VMCALL activates the dual-monitor treatment, it causes an SMM VM exit. Differences between this SMM VM exit and other SMM VM exits are discussed in Sections 30.15.6.1 through 30.15.6.6. See also "VMCALL—Call to VM Monitor" in Chapter 29.

#### 30.15.6.1 Initial Checks

An execution of VMCALL attempts to activate the dual-monitor treatment if (1) the processor supports the dual-monitor treatment;<sup>2</sup> (2) the logical processor is in VMX root operation; (3) the logical processor is outside SMM and the valid bit is set in the IA32\_SMM\_MONITOR\_CTL MSR; (4) the logical processor is not in virtual-8086 mode and not in compatibility mode; (5) CPL = 0; and (6) the dual-monitor treatment is not active.

Such an execution of VMCALL begins with some initial checks. These checks are performed before updating the current-VMCS pointer and the executive-VMCS pointer field (see Section 30.15.2.2).

The VMCS that manages SMM VM exit caused by this VMCALL is the current VMCS established by the executive monitor. The VMCALL performs the following checks on the current VMCS in the order indicated:

1. There must be a current VMCS pointer.
2. The launch state of the current VMCS must be clear.
3. Reserved bits in the VM-exit controls in the current VMCS must be set properly. Software may consult the VMX capability MSR IA32\_VMX\_EXIT\_CTLS to determine the proper settings (see Appendix A.4).

If any of these checks fail, subsequent checks are skipped and VMCALL fails. If all these checks succeed, the logical processor uses the IA32\_SMM\_MONITOR\_CTL MSR to determine the base address of MSEG. The following checks are performed in the order indicated:

1. The logical processor reads the 32 bits at the base of MSEG and compares them to the processor's MSEG revision identifier.

---

1. Alternatively, software may map the MSEG header with the UC memory type; this may be necessary, depending on how memory is organized. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly. In addition, the processor will continue to use the memory type reported in the VMX capability MSR IA32\_VMX\_BASIC with exceptions noted in Appendix A.1.

2. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.

2. The logical processor reads the SMM-transfer monitor features field:
  - Bit 0 of the field is the IA-32e mode SMM feature bit, and it indicates whether the logical processor will be in IA-32e mode after the SMM-transfer monitor (STM) is activated.
    - If the VMCALL is executed on a processor that does not support Intel 64 architecture, the IA-32e mode SMM feature bit must be 0.
    - If the VMCALL is executed in 64-bit mode, the IA-32e mode SMM feature bit must be 1.
  - Bits 31:1 of this field are currently reserved and must be zero.

If any of these checks fail, subsequent checks are skipped and the VMCALL fails.

### 30.15.6.2 Updating the Current-VMCS and Executive-VMCS Pointers

Before performing the steps in Section 30.15.2.2, SMM VM exits that activate the dual-monitor treatment begin by loading the SMM-transfer VMCS pointer with the value of the current-VMCS pointer.

### 30.15.6.3 Saving Guest State

As noted in Section 30.15.2.4, SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area. While this is true also for SMM VM exits that activate the dual-monitor treatment, the VMCS used for those VM exits exists outside SMRAM.

The SMM-transfer monitor (STM) can also discover the current value of the SMBASE register by using the RDMSR instruction to read the IA32\_SMBASE MSR (MSR address 9EH). The following items detail use of this MSR:

- The MSR is supported only if IA32\_VMX\_MISC[15] = 1 (see Appendix A.6).
- A write to the IA32\_SMBASE MSR using WRMSR generates a general-protection fault (#GP(0)). An attempt to write to the IA32\_SMBASE MSR fails if made as part of a VM exit or part of a VM entry.
- A read from the IA32\_SMBASE MSR using RDMSR generates a general-protection fault (#GP(0)) if executed outside of SMM. An attempt to read from the IA32\_SMBASE MSR fails if made as part of a VM exit that does not end in SMM.

### 30.15.6.4 Saving MSRs

The VM-exit MSR-store area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are saved into that area.

### 30.15.6.5 Loading Host State

The VMCS that is current during an SMM VM exit that activates the dual-monitor treatment was established by the executive monitor. It does not contain the VM-exit controls and host state required to initialize the STM. For this reason, such SMM VM exits do not load processor state as described in Section 26.5. Instead, state is set to fixed values or loaded based on the content of the MSEG header (see Table 30-10):

- CR0 is set to as follows:
  - PG, NE, ET, MP, and PE are all set to 1.
  - CD and NW are left unchanged.
  - All other bits are cleared to 0.
- CR3 is set as follows:
  - Bits 63:32 are cleared on processors that support IA-32e mode.
  - Bits 31:12 are set to bits 31:12 of the sum of the MSEG base address and the CR3-offset field in the MSEG header.
  - Bits 11:5 and bits 2:0 are cleared (the corresponding bits in the CR3-offset field in the MSEG header are ignored).

- Bits 4:3 are set to bits 4:3 of the CR3-offset field in the MSEG header.
- CR4 is set as follows:
  - MCE, PGE, CET, and PCIDE are cleared.
  - PAE is set to the value of the IA-32e mode SMM feature bit.
  - If the IA-32e mode SMM feature bit is clear, PSE is set to 1 if supported by the processor; if the bit is set, PSE is cleared.
  - All other bits are unchanged.
- DR7 is set to 400H.
- The IA32\_DEBUGCTL MSR is cleared to 00000000\_00000000H.
- The registers CS, SS, DS, ES, FS, and GS are loaded as follows:
  - All registers are usable.
  - CS.selector is loaded from the corresponding field in the MSEG header (the high 16 bits are ignored), with bits 2:0 cleared to 0. If the result is 0000H, CS.selector is set to 0008H.
  - The selectors for SS, DS, ES, FS, and GS are set to CS.selector+0008H. If the result is 0000H (if the CS selector was FFF8H), these selectors are instead set to 0008H.
  - The base addresses of all registers are cleared to zero.
  - The segment limits for all registers are set to FFFFFFFFH.
  - The AR bytes for the registers are set as follows:
    - CS.Type is set to 11 (execute/read, accessed, non-conforming code segment).
    - For SS, DS, ES, FS, and GS, the Type is set to 3 (read/write, accessed, expand-up data segment).
    - The S bits for all registers are set to 1.
    - The DPL for each register is set to 0.
    - The P bits for all registers are set to 1.
    - On processors that support Intel 64 architecture, CS.L is loaded with the value of the IA-32e mode SMM feature bit.
    - CS.D is loaded with the inverse of the value of the IA-32e mode SMM feature bit.
    - For each of SS, DS, ES, FS, and GS, the D/B bit is set to 1.
    - The G bits for all registers are set to 1.
- LDTR is unusable. The LDTR selector is cleared to 0000H, and the register is otherwise undefined (although the base address is always canonical)
- GDTR.base is set to the sum of the MSEG base address and the GDTR base-offset field in the MSEG header (bits 63:32 are always cleared on processors that support IA-32e mode). GDTR.limit is set to the corresponding field in the MSEG header (the high 16 bits are ignored).
- IDTR.base is unchanged. IDTR.limit is cleared to 0000H.
- RIP is set to the sum of the MSEG base address and the value of the RIP-offset field in the MSEG header (bits 63:32 are always cleared on logical processors that support IA-32e mode).
- RSP is set to the sum of the MSEG base address and the value of the RSP-offset field in the MSEG header (bits 63:32 are always cleared on logical processor that supports IA-32e mode).
- RFLAGS is cleared, except bit 1, which is always set.
- The logical processor is left in the active state.
- Event blocking after the SMM VM exit is as follows:
  - There is no blocking by STI or by MOV SS.
  - There is blocking by non-maskable interrupts (NMIs) and by SMIs.
- There are no pending debug exceptions after the SMM VM exit.

- For processors that support IA-32e mode, the IA32\_EFER MSR is modified so that LME and LMA both contain the value of the IA-32e mode SMM feature bit.

If any of CR3[63:5], CR4.PAE, CR4.PSE, or IA32\_EFER.LMA is changing, the TLBs are updated so that, after VM exit, the logical processor does not use translations that were cached before the transition. This is not necessary for changes that would not affect paging due to the settings of other bits (for example, changes to CR4.PSE if IA32\_EFER.LMA was 1 before and after the transition).

### 30.15.6.6 Loading MSRs

The VM-exit MSR-load area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are loaded from that area.

### 30.15.7 Deactivating the Dual-Monitor Treatment

The SMM-transfer monitor may deactivate the dual-monitor treatment and return the processor to default treatment of SMIs and SMM (see Section 30.14). It does this by executing a VM entry with the “deactivate dual-monitor treatment” VM-entry control set to 1.

As noted in Section 25.2.1.3 and Section 30.15.4.1, an attempt to deactivate the dual-monitor treatment fails in the following situations: (1) the processor is not in SMM; (2) the “entry to SMM” VM-entry control is 1; or (3) the executive-VMCS pointer does not contain the VMXON pointer (the VM entry is to VMX non-root operation).

As noted in Section 30.15.4.9, VM entries that deactivate the dual-monitor treatment ignore the SMI bit in the interruptibility-state field of the guest-state area. Instead, the blocking of SMIs following such a VM entry depends on whether the logical processor is in SMX operation:<sup>1</sup>

- If the logical processor is in SMX operation, SMIs are blocked after VM entry. SMIs may later be unblocked by the VMXOFF instruction (see Section 30.14.4) or by certain leaf functions of the GETSEC instruction (see Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*).
- If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

## 30.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT

On processors that support processor extended states using XSAVE/XRSTOR (see Chapter 13, “Managing State Using the XSAVE Feature Set” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*), the processor does not save any XSAVE/XRSTOR related state on an SMI. It is the responsibility of the SMI handler code to properly preserve the state information (including CR4.OSXSAVE, XCR0, and possibly processor extended states using XSAVE/XRSTOR). Therefore, the SMI handler must follow the rules described in Chapter 13, “Managing State Using the XSAVE Feature Set” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## 30.17 MODEL-SPECIFIC SYSTEM MANAGEMENT ENHANCEMENT

This section describes enhancement of system management features that apply only to the 4th generation Intel Core processors. These features are model-specific. BIOS and SMM handler must use CPUID to enumerate DisplayFamily\_DisplayModel signature when programming with these interfaces.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

### 30.17.1 SMM Handler Code Access Control

The BIOS may choose to restrict the address ranges of code that SMM handler executes. When SMM handler code execution check is enabled, an attempt by the SMM handler to execute outside the ranges specified by SMRR (see Section 30.4.2.1) will cause the assertion of an unrecoverable machine check exception (MCE).

The interface to enable SMM handler code access check resides in a per-package scope model-specific register MSR\_SMM\_FEATURE\_CONTROL at address 4E0H. An attempt to access MSR\_SMM\_FEATURE\_CONTROL outside of SMM will cause a #GP. Writes to MSR\_SMM\_FEATURE\_CONTROL is further protected by configuration interface of MSR\_SMM\_MCA\_CAP at address 17DH.

Details of the interface of MSR\_SMM\_FEATURE\_CONTROL and MSR\_SMM\_MCA\_CAP are described in Table 2-29 in Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

### 30.17.2 SMI Delivery Delay Reporting

Entry into the system management mode occurs at instruction boundary. In situations where a logical processor is executing an instruction involving a long flow of internal operations, servicing an SMI by that logical processor will be delayed. Delayed servicing of SMI of each logical processor due to executing long flows of internal operation in a physical processor can be queried via a package-scope register MSR\_SMM\_DELAYED at address 4E2H.

The interface to enable reporting of SMI delivery delay due to long internal flows resides in a per-package scope model-specific register MSR\_SMM\_DELAYED. An attempt to access MSR\_SMM\_DELAYED outside of SMM will cause a #GP. Availability to MSR\_SMM\_DELAYED is protected by configuration interface of MSR\_SMM\_MCA\_CAP at address 17DH.

Details of the interface of MSR\_SMM\_DELAYED and MSR\_SMM\_MCA\_CAP are described in Table 2-29 in Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

### 30.17.3 Blocked SMI Reporting

A logical processor may have entered into a state and blocked from servicing other interrupts (including SMI). Logical processors in a physical processor that are blocked in serving SMI can be queried in a package-scope register MSR\_SMM\_BLOCKED at address 4E3H. An attempt to access MSR\_SMM\_BLOCKED outside of SMM will cause a #GP.

Details of the interface of MSR\_SMM\_BLOCKED is described in Table 2-29 in Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

## 25. Updates to Chapter 32, Volume 3D

Change bars and green text show changes to Chapter 32 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: Update to Table 32-7, "CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources".

## CHAPTER 32 INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

### 32.1 OVERVIEW

Intel® Software Guard Extensions (Intel® SGX) is a set of instructions and mechanisms for memory accesses added to Intel® Architecture processors. Intel SGX can encompass two collections of instruction extensions, referred to as SGX1 and SGX2, see Table 32-1 and Table 32-2. The SGX1 extensions allow an application to instantiate a protected container, referred to as an enclave. The enclave is a trusted area of memory, where critical aspects of the application functionality have hardware-enhanced confidentiality and integrity protections. New access controls to restrict access to software not resident in the enclave are also introduced. The SGX2 extensions allow additional flexibility in runtime management of enclave resources and thread execution within an enclave.

Chapter 33 covers main concepts, objects and data structure formats that interact within the Intel SGX architecture. Chapter 34 covers operational aspects ranging from preparing an enclave, transferring control to enclave code, and programming considerations for the enclave code and system software providing support for enclave execution. Chapter 35 describes the behavior of Asynchronous Enclave Exit (AEX) caused by events while executing enclave code. Chapter 36 covers the syntax and operational details of the instruction and associated leaf functions available in Intel SGX. Chapter 37 describes interaction of various aspects of IA32 and Intel® 64 architectures with Intel SGX. Chapter 38 covers Intel SGX support for application debug, profiling and performance monitoring.

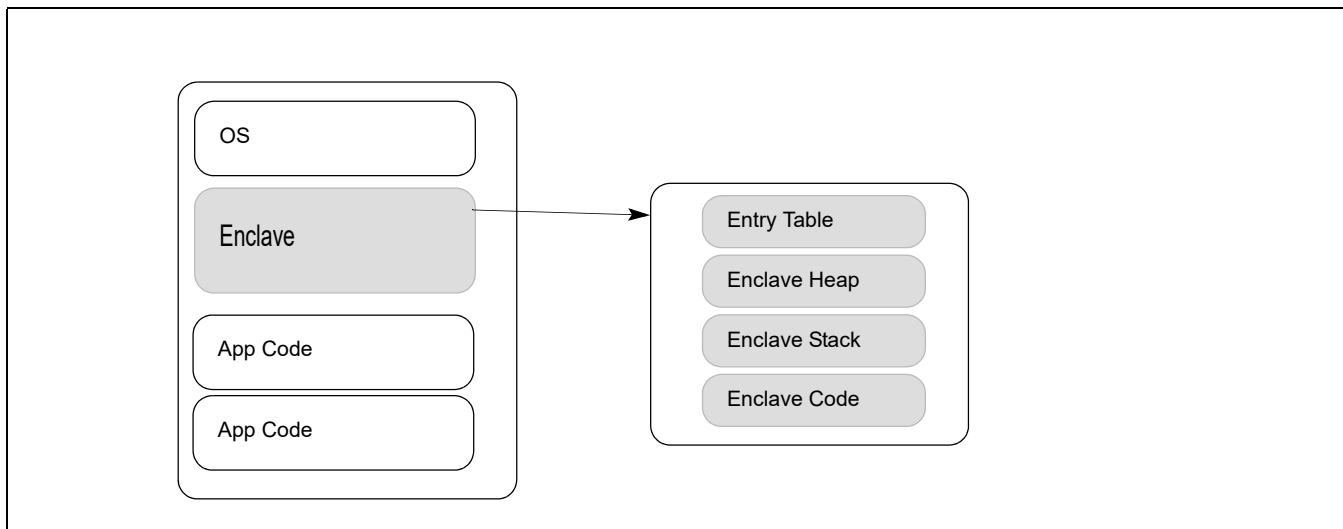


Figure 32-1. An Enclave Within the Application's Virtual Address Space

### 32.2 ENCLAVE INTERACTION AND PROTECTION

Intel SGX allows the protected portion of an application to be distributed in the clear. Before the enclave is built, the enclave code and data are free for inspection and analysis. The protected portion is loaded into an enclave where its code and data is measured. Once the application's protected portion of the code and data are loaded into an enclave, memory access controls are in place to restrict access by external software. An enclave can prove its identity to a remote party and provide the necessary building-blocks for secure provisioning of keys and credentials. The application can also request an enclave-specific and platform-specific key that it can use to protect keys and data that it wishes to store outside the enclave.<sup>1</sup>

1. For additional information, see white papers on Intel SGX at <http://software.intel.com/en-us/intel-isa-extensions>.

Intel SGX introduces two significant capabilities to the Intel Architecture. First is the change in enclave memory access semantics. The second is protection of the address mappings of the application.

## 32.3 ENCLAVE LIFE CYCLE

Enclave memory management is divided into two parts: address space allocation and memory commitment. Address space allocation is the specification of the range of linear addresses that the enclave may use. This range is called the ELRANGE. No actual resources are committed to this region. Memory commitment is the assignment of actual memory resources (as pages) within the allocated address space. This two-phase technique allows flexibility for enclaves to control their memory usage and to adjust dynamically without overusing memory resources when enclave needs are low. Commitment adds physical pages to the enclave. An operating system may support separate allocate and commit operations.

During enclave creation, code and data for an enclave are loaded from a clear-text source, i.e. from non-enclave memory.

Untrusted application code starts using an initialized enclave typically by using the EENTER leaf function provided by Intel SGX to transfer control to the enclave code residing in the protected Enclave Page Cache (EPC). The enclave code returns to the caller via the EEXIT leaf function. Upon enclave entry, control is transferred by hardware to software inside the enclave. The software inside the enclave switches the stack pointer to one inside the enclave. When returning back from the enclave, the software swaps back the stack pointer then executes the EEXIT leaf function.

On processors that support the SGX2 extensions, an enclave writer may add memory to an enclave using the SGX2 instruction set, after the enclave is built and running. These instructions allow adding additional memory resources to the enclave for use in such areas as the heap. In addition, SGX2 instructions allow the enclave to add new threads to the enclave. The SGX2 features provide additional capabilities to the software model without changing the security properties of the Intel SGX architecture.

Calling an external procedure from an enclave could be done using the EEXIT leaf function. Software would use EEXIT and a software convention between the trusted section and the untrusted section.

An active enclave consumes resources from the Enclave Page Cache (EPC, see Section 32.5). Intel SGX provides the EREMOVE instruction that an EPC manager can use to reclaim EPC pages committed to an enclave. The EPC manager uses EREMOVE on every enclave page when the enclave is torn down. After successful execution of EREMOVE the EPC page is available for allocation to another enclave.

## 32.4 DATA STRUCTURES AND ENCLAVE OPERATION

There are 2 main data structures associated with operating an enclave, the SGX Enclave Control Structure (SECS, see Section 33.7) and the Thread Control Structure (TCS, see Section 33.8).

There is one SECS for each enclave. The SECS contains meta-data about the enclave which is used by the hardware and cannot be directly accessed by software. Included in the SECS is a field that stores the enclave build measurement value. This field, MRENCLAVE, is initialized by the ECREATE instruction and updated by every EADD and EEXTEND. It is locked by EINIT.

Every enclave contains one or more TCS structures. The TCS contains meta-data used by the hardware to save and restore thread specific information when entering/exiting the enclave. There is one field, FLAGS, that may be accessed by software. This field can only be accessed by debug enclaves. The flag bit, DBGOPTIN, allows to single step into the thread associated with the TCS. (see Section 33.8.1)

The SECS is created when ECREATE (see Table 32-1) is executed. The TCS can be created using the EADD instruction or the SGX2 instructions (see Table 32-2).

## 32.5 ENCLAVE PAGE CACHE

The Enclave Page Cache (EPC) is the secure storage used to store enclave pages when they are a part of an executing enclave. For an EPC page, hardware performs additional access control checks to restrict access to the page. After the current page access checks and translations are performed, the hardware checks that the EPC page



is accessible to the program currently executing. Generally an EPC page is only accessed by the owner of the executing enclave or an instruction which is setting up an EPC page

The EPC is divided into EPC pages. An EPC page is 4KB in size and always aligned on a 4KB boundary.

Pages in the EPC can either be valid or invalid. Every valid page in the EPC belongs to one enclave instance. Each enclave instance has an EPC page that holds its SECS. The security metadata for each EPC page is held in an internal micro-architectural structure called Enclave Page Cache Map (EPCM, see Section 32.5.1).

The EPC is managed by privileged software. Intel SGX provides a set of instructions for adding and removing content to and from the EPC. The EPC may be configured by BIOS at boot time. On implementations in which EPC memory is part of system DRAM, the contents of the EPC are protected by an encryption engine.

### 32.5.1 Enclave Page Cache Map (EPCM)

The EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds one entry for each page in the EPC. The format of the EPCM is micro-architectural, and consequently is implementation dependent. However, the EPCM contains the following architectural information:

- The status of EPC page with respect to validity and accessibility.
- An SECS identifier (see Section 33.20) of the enclave to which the page belongs.
- The type of page: regular, SECS, TCS or VA.
- The linear address through which the enclave is allowed to access the page.
- The specified read/write/execute permissions on that page.

The EPCM structure is used by the CPU in the address-translation flow to enforce access-control on the EPC pages. The EPCM structure is described in Table 33-29, and the conceptual access-control flow is described in Section 33.5.

The EPCM entries are managed by the processor as part of various instruction flows.

## 32.6 ENCLAVE INSTRUCTIONS AND INTEL® SGX

The enclave instructions available with Intel SGX are organized as leaf functions under three instruction mnemonics: ENCLS (ring 0), ENCLU (ring 3), and ENCLV (VT root mode). Each leaf function uses EAX to specify the leaf function index, and may require additional implicit input registers as parameters. The use of EAX is implied implicitly by the ENCLS, ENCLU, and ENCLV instructions; ModR/M byte encoding is not used with ENCLS, ENCLU, and ENCLV. The use of additional registers does not use ModR/M encoding and is implied implicitly by the respective leaf function index.

Each leaf function index is also associated with a unique, leaf-specific mnemonic. A long-form expression of Intel SGX instruction takes the form of ENCLx[LEAF\_MNEMONIC], where 'x' is either 'S', 'U', or 'V'. The long-form expression provides clear association of the privilege-level requirement of a given "leaf mnemonic". For simplicity, the unique "Leaf\_Mnemonic" name is used (omitting the ENCLx for convenience) throughout in this document.

Details of individual SGX leaf functions are described in Chapter 36. Table 32-1 provides a summary of the instruction leaves that are available in the initial implementation of Intel SGX, which is introduced in the 6th generation Intel Core processors. Table 32-2 summarizes enhancement of Intel SGX for future Intel processors.

**Table 32-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1**

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add an EPC page to an enclave.	ENCLU[EENTER]	Enter an enclave.
ENCLS[EBLOCK]	Block an EPC page.	ENCLU[EEXIT]	Exit an enclave.
ENCLS[ECREATE]	Create an enclave.	ENCLU[EGETKEY]	Create a cryptographic key.
ENCLS[EDBGDRD]	Read data from a debug enclave by debugger.	ENCLU[EREPORT]	Create a cryptographic report.

**Table 32-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1**

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EDBGWR]	Write data into a debug enclave by debugger.	ENCLU[ERESUME]	Re-enter an enclave.
ENCLS[EEXTEND]	Extend EPC page measurement.		
ENCLS[EINIT]	Initialize an enclave.		
ENCLS[ELDB]	Load an EPC page in blocked state.		
ENCLS[ELDU]	Load an EPC page in unblocked state.		
ENCLS[EPA]	Add an EPC page to create a version array.		
ENCLS[EREMOVE]	Remove an EPC page from an enclave.		
ENCLS[ETRACK]	Activate EBLOCK checks.		
ENCLS[EWB]	Write back/invalidate an EPC page.		

**Table 32-2. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX2**

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EAUG]	Allocate EPC page to an existing enclave.	ENCLU[EACCEPT]	Accept EPC page into the enclave.
ENCLS[EMODPR]	Restrict page permissions.	ENCLU[EMODPE]	Enhance page permissions.
ENCLS[EMODT]	Modify EPC page type.	ENCLU[EACCEPTCOPY]	Copy contents to an augmented EPC page and accept the EPC page into the enclave.

**Table 32-3. VMX Operation and Supervisor Mode Enclave Instruction Leaf Functions in Long-Form of OVERSUB**

VMX Operation	Description	Supervisor Instruction	Description
ENCLV[EDECVRTCHILD]	Decrement the virtual child page count.	ENCLS[ERDINFO]	Read information about EPC page.
ENCLV[EINCVIRTCHILD]	Increment the virtual child page count.	ENCLS[ETRACKC]	Activate EBLOCK checks with conflict reporting.
ENCLV[ESETCONTEXT]	Set virtualization context.	ENCLS[ELDBC/UC]	Load an EPC page with conflict reporting.

## 32.7 DISCOVERING SUPPORT FOR INTEL® SGX AND ENABLING ENCLAVE INSTRUCTIONS

Detection of support of Intel SGX and enumeration of available and enabled Intel SGX resources are queried using the CPUID instruction. The enumeration interface comprises the following:

- Processor support of Intel SGX is enumerated by a feature flag in CPUID leaf 07H: CPUID.(EAX=07H, ECX=0H):EBX.SGX[bit 2]. If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor has support for Intel SGX, and requires opt-in enabling by BIOS via IA32\_FEATURE\_CONTROL MSR.  
If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, CPUID will report via the available sub-leaves of CPUID.(EAX=12H) on available and/or configured Intel SGX resources.
- The available and configured Intel SGX resources enumerated by the sub-leaves of CPUID.(EAX=12H) depend on the state of BIOS configuration.

### 32.7.1 Intel® SGX Opt-In Configuration

On processors that support Intel SGX, IA32\_FEATURE\_CONTROL provides the SGX\_ENABLE field (bit 18). Before system software can configure and enable Intel SGX resources, BIOS is required to set IA32\_FEATURE\_CONTROL.SGX\_ENABLE = 1 to opt-in the use of Intel SGX by system software.

The semantics of setting SGX\_ENABLE follows the rules of IA32\_FEATURE\_CONTROL.LOCK (bit 0). Software is considered to have opted into Intel SGX if and only if IA32\_FEATURE\_CONTROL.SGX\_ENABLE and IA32\_FEATURE\_CONTROL.LOCK are set to 1. The setting of IA32\_FEATURE\_CONTROL.SGX\_ENABLE (bit 18) is not reflected by CPUID.

**Table 32-4. Intel® SGX Opt-in and Enabling Behavior**

CPUID.(07H,0H):EBX.SGX	CPUID.(12H)	FEATURE_CONTROL.LOCK	FEATURE_CONTROL.SGX_ENABLE	Enclave Instruction
0	Invalid	X	X	#UD
1	Valid*	X	X	#UD**
1	Valid*	0	X	#GP
1	Valid*	1	0	#GP
1	Valid*	1	1	Available (see Table 32-5 for details of SGX1 and SGX2).

\* Leaf 12H enumeration results are dependent on enablement.  
 \*\* See list of conditions in the #UD section of the reference pages of ENCLS and ENCLU

### 32.7.2 Intel® SGX Resource Enumeration Leaves

If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor also supports querying CPUID with EAX=12H on Intel SGX resource capability and configuration. The number of available sub-leaves in leaf 12H depends on the Opt-in and system software configuration. Information returned by CPUID.12H is thread specific; software should not assume that if Intel SGX instructions are supported on one hardware thread, they are also supported elsewhere.

A properly configured processor exposes Intel SGX functionality with CPUID.EAX=12H reporting valid information (non-zero content) in three or more sub-leaves, see Table 32-5.

- CPUID.(EAX=12H, ECX=0H) enumerates Intel SGX capability, including enclave instruction opcode support.
- CPUID.(EAX=12H, ECX=1H) enumerates Intel SGX capability of processor state configuration and enclave configuration in the SECS structure (see Table 33-3).
- CPUID.(EAX=12H, ECX > 1) enumerates available EPC resources.

**Table 32-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities**

CPUID.(EAX=12H,ECX=0)		Description Behavior
Register	Bits	
EAX	0	SGX1: If 1, indicates leaf functions of SGX1 instruction listed in Table 32-1 are supported.
	1	SGX2: If 1, indicates leaf functions of SGX2 instruction listed in Table 32-2 are supported.
	4:2	Reserved (0)
	5	OVERSUB: If 1, indicates Intel SGX supports instructions: EINCVRTCHILD, EDECVRTCHILD, and ESETCONTEXT.
	6	OVERSUB: If 1, indicates Intel SGX supports instructions: ETRACKC, ERDINFO, ELDBC, and ELDUC.
	31:7	Reserved (0)
EBX	31:0	MISCSELECT: Reports the bit vector of supported extended features that can be written to the MISC region of the SSA.
ECX	31:0	Reserved (0).

**Table 32-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities**

CPUID.(EAX=12H,ECX=0)		Description Behavior
Register	Bits	
EDX	7:0	MaxEnclaveSize_Not64: the maximum supported enclave size is 2^(EDX[7:0]) bytes when not in 64-bit mode.
	15:8	MaxEnclaveSize_64: the maximum supported enclave size is 2^(EDX[15:8]) bytes when operating in 64-bit mode.
	31:16	Reserved (0).

**Table 32-6. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities**

CPUID.(EAX=12H,ECX=1)		Description Behavior
Register	Bits	
EAX	31:0	Report the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE. SECS.ATTRIBUTES[n] can be set to 1 using ECREATE only if EAX[n] is 1, where n < 32.
EBX	31:0	Report the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE. SECS.ATTRIBUTES[n+32] can be set to 1 using ECREATE only if EBX[n] is 1, where n < 32.
ECX	31:0	Report the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE. SECS.ATTRIBUTES[n+64] can be set to 1 using ECREATE only if ECX[n] is 1, where n < 32.
EDX	31:0	Report the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE. SECS.ATTRIBUTES[n+96] can be set to 1 using ECREATE only if EDX[n] is 1, where n < 32.

On processors that support Intel SGX1 and SGX2, CPUID leaf 12H sub-leaf 2 report physical memory resources available for use with Intel SGX. These physical memory sections are typically allocated by BIOS as **Processor Reserved Memory**, and available to the OS to manage as EPC.

To enumerate how many EPC sections are available to the EPC manager, software can enumerate CPUID leaf 12H with sub-leaf index starting from 2, and decode the sub-leaf-type encoding (returned in EAX[3:0]) until the sub-leaf type is invalid. All invalid sub-leaves of CPUID leaf 12H return EAX/EBX/ECX/EDX with 0.

**Table 32-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources**

CPUID.(EAX=12H,ECX > 1)		Description Behavior
Register	Bits	
EAX	3:0	0000b: This sub-leaf is invalid; EDX:ECX:EBX:EAX return 0. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other encodings are reserved.
	11:4	Reserved (enumerate 0).
	31:12	If EAX[3:0] = 0001b, these are bits 31:12 of the physical address of the base of the EPC section.
EBX	19:0	If EAX[3:0] = 0001b, these are bits 51:32 of the physical address of the base of the EPC section.
	31:20	Reserved.
ECX	3:0	If ECX[3:0] = 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If ECX[3:0] = 0001b, then this section has confidentiality and integrity protection. If ECX[3:0] = 0010b, then this section has confidentiality protection only. All other encodings are reserved.
	11:4	Reserved (enumerate 0).
	31:12	If EAX[3:0] = 0001b, these are bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.

**Table 32-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources**

CPUID.(EAX=12H,ECX > 1)		Description Behavior
Register	Bits	
EDX	19:0	If EAX[3:0] = 0001b, these are bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory.
	31:20	Reserved.

## 32.8 INTEL® SGX INTERACTIONS WITH CONTROL-FLOW ENFORCEMENT TECHNOLOGY

This section discusses extensions to the Intel SGX architecture to support CET.

### 32.8.1 CET in Enclaves Model

Each enclave has its private configuration for CET that is not shared with the CET configurations of the enclosing application. On entry into the enclave, the CET state of the enclosing application is saved into scratchpad registers inside the processor and the CET state of the enclave is established. On an asynchronous exit, the enclave CET state is saved into the enclave state save area frame. On exit from the enclave, the CET state of the enclosing application is re-established from the scratchpad registers.

A new page type, `PT_SS_FIRST`, is used to denote pages in an enclave that can be used as a first page of a shadow stack.

A new page type, `PT_SS_REST`, is used to denote pages in an enclave that can be used as a non-first page of a shadow stack.

A page denoted as `PT_SS_FIRST` and `PT_SS_REST` will be a legal target for `shadow_stack_load`, `shadow_stack_store`, and regular load operations. Regular stores will be disallowed to such pages. A `PT_SS_FIRST/PT_SS_REST` page must be writeable in the IA page tables and in EPT.

When in enclave mode, `shadow_stack_load` and `shadow_stack_store` operations must be to addresses in the enclave `ELRANGE`.

The `EAUG` instruction is extended to allocate pages of type `PT_SS_FIRST/PT_SS_REST`; this page type requires specifying a `SECINFO` structure with page parameters. Shadow page permission must be R/W. Regular R/W pages may continue to be allocated by providing a `SECINFO` pointer value of 0. Regular R/W pages may also be allocated by providing a `SECINFO` structure that specifies the page parameters. The `EAUG` instruction creates a shadow-stack-restore token at offset `0xFF8` on a `PT_SS_FIRST` page. This allows a dynamically created shadow stack to be restored using the `RSTORSSP` instruction. The `EADD` and `EAUG` instructions disallow creation of a `PT_SS_FIRST` or `PT_SS_REST` page as the first or last page in `ELRANGE`.

The `EADD` instruction requires that the `PT_SS_REST` page be all zeroes. The `EADD` instruction requires that a `PT_SS_FIRST` page be all zeroes except the 8 bytes at offset `0xFF8` on that page that must have a shadow-stack-restore token. This shadow-stack-restore token must have a linear address which is the linear address of the `PT_SS_FIRST` page + 4096. As an enclave could be loaded at varying linear addresses, the enclave builder should not extend the measurement of the `PT_SS_FIRST` pages into the measurement registers. On first entry on to the enclave using a TCS, the enclave software can use the `RSTORSSP` instruction to restore its SSP. Subsequent to performing a `RSTORSSP`, the enclave software can use the `INCSSP` instruction to pop the previous-ssp token that is created by the `RSTORSSP` instruction at the top of the restored shadow stack.

On an enclave entry, the SSP will be initialized to the value in a new TCS field called `PREVSSP`. The `PREVSSP` field is written with the value of SSP on enclave exit and is loaded into SSP at enclave entry. When a TCS page is added using `EADD` or accepted using `EACCEPT`, the processor requires the `PREVSSP` field to be initialized to 0.

### 32.8.2 Operations Not Supported on Shadow Stack Pages

The following operations are not allowed on pages of type PT\_SS\_FIRST and PT\_SS\_REST:

- EACCEPTCOPY
- EMODPR
- EMODPE

### 32.8.3 Indirect Branch Tracking - Legacy Compatibility Treatment

The legacy code page bitmap is tested using the page offset within the ELRANGE instead of the absolute linear address of the address where ENDBRANCH was missed; see the detailed algorithm in Section 18.3.6, “Legacy Compatibility Treatment” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for additional details.

## 26. Updates to Chapter 35, Volume 3D

Change bars and green text show changes to Chapter 35 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: Updated AEX flow with CET and Intel SGX interaction clarifications.

## CHAPTER 35

# ENCLAVE EXITING EVENTS

Certain events, such as exceptions and interrupts, incident to (but asynchronous with) enclave execution may cause control to transition outside of enclave mode. (Most of these also cause a change of privilege level.) To protect the integrity and security of the enclave, the processor will exit the enclave (and enclave mode) before invoking the handler for such an event. For that reason, such events are called **enclave-exiting events** (EEE); EEEs include external interrupts, non-maskable interrupts, system-management interrupts, exceptions, and VM exits.

The process of leaving an enclave in response to an EEE is called an **asynchronous enclave exit** (AEX). To protect the secrecy of the enclave, an AEX saves the state of certain registers within enclave memory and then loads those registers with fixed values called **synthetic state**.

### 35.1 COMPATIBLE SWITCH TO THE EXITING STACK OF AEX

AEXs load registers with a pre-determined synthetic state. These registers may be later pushed onto the appropriate stack in a form as defined by the enclave-exiting event. To allow enclave execution to resume after the invoking handler has processed the enclave exiting event, the asynchronous enclave exit loads the address of trampoline code outside of the enclave into RIP. This trampoline code eventually returns to the enclave by means of an ENCLU(ERESUME) leaf function. Prior to exiting the enclave the RSP and RBP registers are restored to their values prior to enclave entry.

The stack to be used is chosen using the same rules as for non-SGX mode:

- If there is a privilege level change, the stack will be the one associated with the new ring.
- If there is no privilege level change, the current application stack is used.
- If the IA-32e IST mechanism is used, the exit stack is chosen using that method.

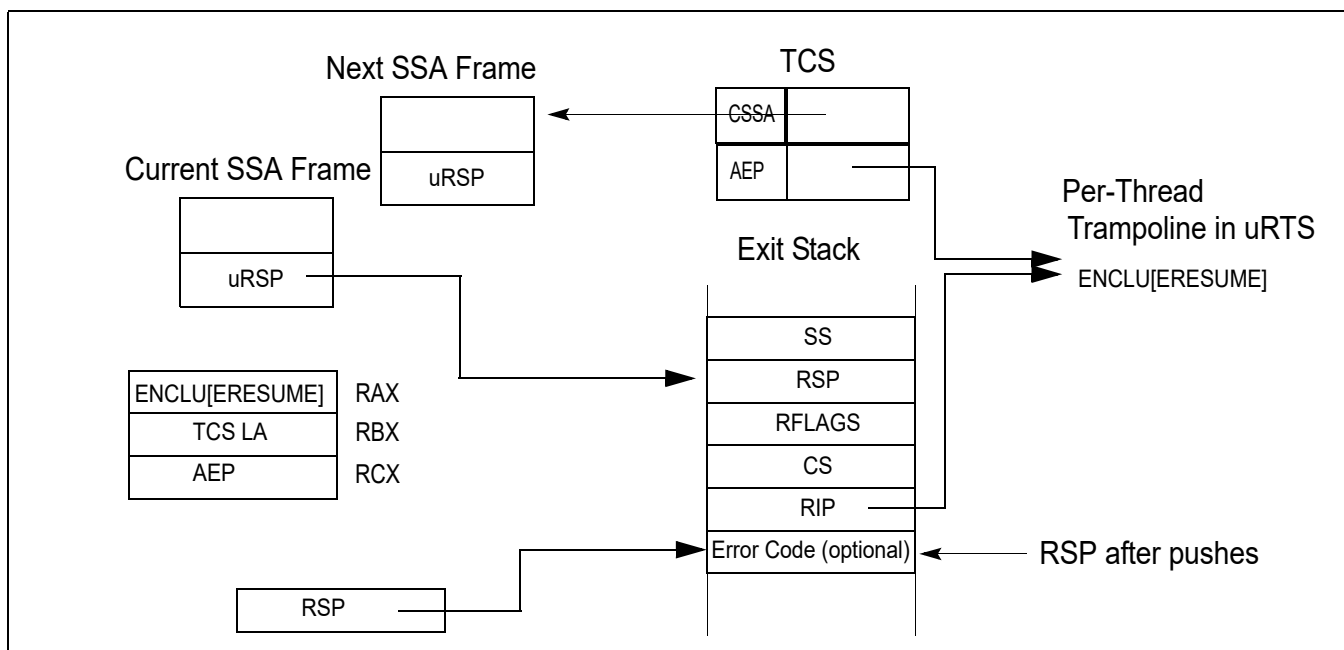


Figure 35-1. Exit Stack Just After Interrupt with Stack Switch



In all cases, the choice of exit stack and the information pushed onto it is consistent with non-SGX operation. Figure 35-1 shows the Application and Exiting Stacks after an exit with a stack switch. An exit without a stack switch uses the Application Stack. The ERESUME leaf index value is placed into RAX, the TCS pointer is placed in RBX and the AEP (see below) is placed into RCX to facilitate resuming the enclave after the exit.

Upon an AEX, the AEP (Asynchronous Exit Pointer) is loaded into the RIP. The AEP points to a trampoline code sequence which includes the ERESUME instruction that is later used to reenter the enclave.

The following bits of RFLAGS are cleared before RFLAGS is pushed onto the exit stack: CF, PF, AF, ZF, SF, OF, RF. The remaining bits are left unchanged.

## 35.2 STATE SAVING BY AEX

The State Save Area holds the processor state at the time of an AEX. To allow handling events within the enclave and re-entering it after an AEX, the SSA can be a stack of multiple SSA frames as illustrated in Figure 35-2.

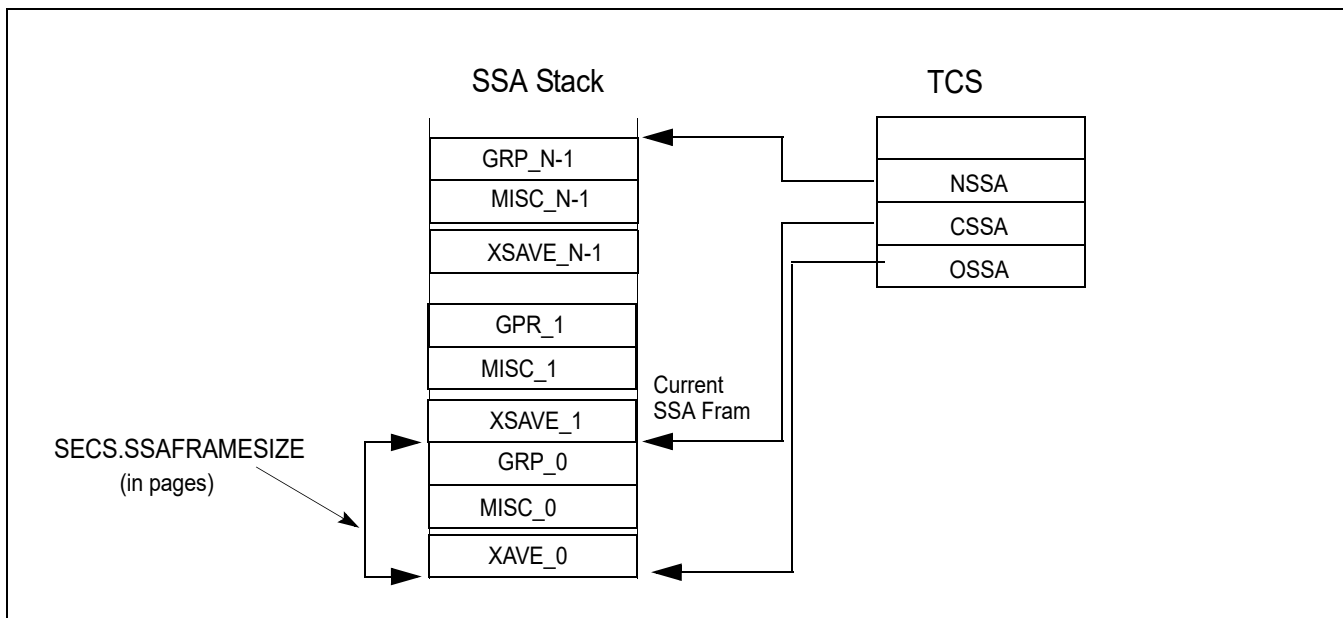


Figure 35-2. The SSA Stack

The location of the SSA frames to be used is controlled by the following variables in the TCS and the SECS:

- Size of a frame in the State Save Area (SECS.SSAFRAMESIZE): This defines the number of 4-KByte pages in a single frame in the State Save Area. The SSA frame size must be large enough to hold the GPR state, the XSAVE state, and the MISC state.
- Base address of the enclave (SECS.BASEADDR): This defines the enclave's base linear address from which the offset to the base of the SSA stack is calculated.
- Number of State Save Area Slots (TCS.NSSA): This defines the total number of slots (frames) in the State Save Area stack.
- Current State Save Area Slot (TCS.CSSA): This defines the slot to use on the next exit.
- State Save Area Offset (TCS.OSSA): This defines the offset of the base address of a set of State Save Area slots from the enclave's base address.

When an AEX occurs, hardware selects the SSA frame to use by examining TCS.CSSA. Processor state is saved into the SSA frame (see Section 35.4) and loaded with a synthetic state (as described in Section 35.3.1) to avoid leaking secrets, RSP and RBP are restored to their values prior to enclave entry, and TCS.CSSA is incremented. As will be described later, if an exception takes the last slot, it will not be possible to reenter the enclave to handle the

exception from within the enclave. A subsequent ERESUME restores the processor state from the current SSA frame and frees the SSA frame.

The format of the XSAVE section of SSA is identical to the format used by the XSAVE/XRSTOR instructions. On EENTER, CSSA must be less than NSSA, ensuring that there is at least one State Save Area slot available for exits. If there is no free SSA frame when executing EENTER, the entry will fail.

## 35.3 SYNTHETIC STATE ON ASYNCHRONOUS ENCLAVE EXIT

### 35.3.1 Processor Synthetic State on Asynchronous Enclave Exit

Table 35-1 shows the synthetic state loaded on AEX. The values shown are the lower 32 bits when the processor is in 32 bit mode and 64 bits when the processor is in 64 bit mode.

**Table 35-1. GPR, x87, SSE Synthetic States on Asynchronous Enclave Exit**

Register	Value
RAX	3 (ENCLU[3] is ERESUME).
RBX	Pointer to TCS of interrupted enclave thread.
RCX	AEP of interrupted enclave thread.
RDX, RSI, RDI	0.
RSP	Restored from SSA.uRSP.
RBP	Restored from SSA.uRBP.
R8-R15	0 in 64-bit mode; unchanged in 32-bit mode.
RIP	AEP of interrupted enclave thread.
RFLAGS	CF, PF, AF, ZF, SF, OF, RF bits are cleared. All other bits are left unchanged.
x87/SSE State	Unless otherwise listed here, all x87 and SSE state are set to the INIT state. The INIT state is the state that would be loaded by the XRSTOR instruction with bits 1:0 both set in the requested feature bitmask (RFBM), and both clear in XSTATE_BV the XSAVE header.
FCW	On #MF exception: set to 037EH. On all other exits: set to 037FH.
FSW	On #MF exception: set to 8081H. On all other exits: set to 0H.
MXCSR	On #XM exception: set to 1F01H. On all other exits: set to 1FBOH.
CR2	If the event that caused the AEX is a #PF, and the #PF does not directly cause a VM exit, then the low 12 bits are cleared. If the #PF leads directly to a VM exit, CR2 is not updated (usual IA behavior). Note: The low 12 bits are not cleared if a #PF is encountered during the delivery of the EEE that caused the AEX. This is because the #PF was not the EEE.
FS, GS	Restored to values as of most recent EENTER/ERESUME.

### 35.3.2 Synthetic State for Extended Features

When CR4.OSXSAVE = 1, extended features (those controlled by XCR0[63:2]) are set to their respective INIT states when this corresponding bit of SECS.XFRM is set. The INIT state is the state that would be loaded by the XRSTOR instruction had the instruction mask and the XSTATE\_BV field of the XSAVE header each contained the value XFRM. (When the AEX occurs in 32-bit mode, those features that do not exist in 32-bit mode are unchanged.)

### 35.3.3 Synthetic State for MISC Features

State represented by SECS.MISCSELECT might also be overridden by synthetic state after it has been saved into the SSA. State represented by MISCSELECT[0] is not overridden but if the exiting event is a page fault then lower 12 bits of CR2 are cleared.

## 35.4 AEX FLOW

On Enclave Exiting Events (interrupts, exceptions, VM exits or SMIs), the processor state is securely saved inside the enclave, a synthetic state is loaded and the enclave is exited. The EEE then proceeds in the usual exit-defined fashion. The following sections describes the details of an AEX:

1. The exact processor state saved into the current SSA frame depends on whether the enclave is a 32-bit or a 64-bit enclave. In 32-bit mode (IA32\_EFER.LMA = 0 || CS.L = 0), the low 32 bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP and EFLAGS) are stored. The upper 32 bits of the legacy registers and the 64-bit registers (R8 ... R15) are not stored.

In 64-bit mode (IA32\_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 ... R15, RIP and RFLAGS) are stored.

The state of those extended features specified by SECS.ATTRIBUTES.XFRM are stored into the XSAVE area of the current SSA frame. The layout of the x87 and XMM portions (the 1st 512 bytes) depends on the current values of IA32\_EFER.LMA and CS.L:

If IA32\_EFER.LMA = 0 || CS.L = 0, the same format (32-bit) that XSAVE/FXSAVE uses with these values.

If IA32\_EFER.LMA = 1 && CS.L = 1, the same format (64-bit) that XSAVE/FXSAVE uses with these values when REX.W = 1.

The cause of the AEX is saved in the EXITINFO field. See Table 33-10 for details and values of the various fields.

The state of those miscellaneous features (see Section 33.7.2) specified by SECS.MISCSELECT are stored into the MISC area of the current SSA frame.

If CET was enabled in the enclave, then the CET state of the enclave is saved in the CET state save area. If shadow stacks were enabled in the enclave, then the SSP is also saved into the TCS.PREVSSP field.

2. Synthetic state is created for a number of processor registers to present an opaque view of the enclave state. Table 35-1 shows the values for GPRs, x87, SSE, FS, GS, Debug and performance monitoring on AEX. The synthetic state for other extended features (those controlled by XCR0[62:2]) is set to their respective INIT states when their corresponding bit of SECS.ATTRIBUTES.XFRM is set. The INIT state is that state as defined by the behavior of the XRSTOR instruction when HEADER.XSTATE\_BV[n] is 0. Synthetic state of those miscellaneous features specified by SECS.MISCSELECT depends on the miscellaneous feature. There is no synthetic state required for the miscellaneous state controlled by SECS.MISCSELECT[0].
3. Any code and data breakpoints that were suppressed at the time of enclave entry are unsuppressed when exiting the enclave.
4. RFLAGS.TF is set to the value that it had at the time of the most recent enclave entry (except for the situation that the entry was opt-in for debug; see Section 38.2). In the SSA, RFLAGS.TF is set to 0.
5. RFLAGS.RF is set to 0 in the synthetic state. In the SSA, the value saved is the same as what would have been saved on stack in the non-SGX case (architectural value of RF). Thus, AEXs due to interrupts, traps, and code breakpoints save RF unmodified into SSA, while AEXs due to other faults save RF as 1 in the SSA.  
If the event causing AEX happened on intermediate iteration of a REP-prefixed instruction, then RF=1 is saved on SSA, irrespective of its priority.
6. Any performance monitoring activity (including PEBS) or profiling activity (LBR, Tracing using Intel PT) on the exiting thread that was suppressed due to the enclave entry on that thread is unsuppressed. Any counting that had been demoted from AnyThread counting to MyThread counting (on one logical processor) is promoted back to AnyThread counting.
7. The CET state of the enclosing application is restored to the state at the time of the most recent enclave entry, and if CET indirect branch tracking was enabled then the indirect branch tracker is unsuppressed and moved to the WAIT\_FOR\_ENDBRANCH state.

## 35.4.1 AEX Operational Detail

Temp Variables in AEX Operational Flow

Name	Type	Size (bits)	Description
TMP_RIP	Effective Address	32/64	Address of instruction at which to resume execution on ERESUME.
TMP_MODE64	binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_BRANCH_RECORD	LBR Record	2x64	From/To address to be pushed onto LBR stack.

The pseudo code in this section describes the internal operations that are executed when an AEX occurs in enclave mode. These operations occur just before the normal interrupt or exception processing occurs.

(\* Save RIP for later use \*)

TMP\_RIP = Linear Address of Resume RIP

(\* Is the processor in 64-bit mode? \*)

TMP\_MODE64 := ((IA32\_EFER.LMA = 1) && (CS.L = 1));

(\* Save all registers, When saving EFLAGS, the TF bit is set to 0 and the RF bit is set to what would have been saved on stack in the non-SGX case \*)

IF (TMP\_MODE64 = 0)

THEN

Save EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EFLAGS, EIP into the current SSA frame using CR\_GPR\_PA; (\* see Table 36-5 for list of CREGs used to describe internal operation within Intel SGX \*)

SSA.RFLAGS.TF := 0;

ELSE (\* TMP\_MODE64 = 1 \*)

Save RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8-R15, RFLAGS, RIP into the current SSA frame using CR\_GPR\_PA;

SSA.RFLAGS.TF := 0;

FI;

Save FS and GS BASE into SSA using CR\_GPR\_PA;

(\* store XSAVE state into the current SSA frame's XSAVE area using the physical addresses that were determined and cached at enclave entry time with CR\_XSAVE\_PAGE\_i. \*)

For each XSAVE state i defined by (SECS.ATTRIBUTES.XFRM[i] = 1, destination address cached in CR\_XSAVE\_PAGE\_i)

SSA.XSAVE.i := XSAVE\_STATE\_i;

(\* Clear bytes 8 to 23 of XSAVE\_HEADER, i.e. the next 16 bytes after XHEADER\_BV \*)

CR\_XSAVE\_PAGE\_0.XHEADER\_BV[191:64] := 0;

(\* Clear bits in XHEADER\_BV[63:0] that are not enabled in ATTRIBUTES.XFRM \*)

CR\_XSAVE\_PAGE\_0.XHEADER\_BV[63:0] :=

CR\_XSAVE\_PAGE\_0.XHEADER\_BV[63:0] & SECS(CR\_ACTIVE\_SECS).ATTRIBUTES.XFRM;

Apply synthetic state to GPRs, RFLAGS, extended features, etc.

(\* Restore the RSP and RBP from the current SSA frame's GPR area using the physical address that was determined and cached at enclave entry time with CR\_GPR\_PA. \*)

RSP := CR\_GPR\_PA.URSP;

RBP := CR\_GPR\_PA.URBP;

## ENCLAVE EXITING EVENTS

```
(* Restore the FS and GS *)
FS.selector := CR_SAVE_FS.selector;
FS.base := CR_SAVE_FS.base;
FS.limit := CR_SAVE_FS.limit;
FS.access_rights := CR_SAVE_FS.access_rights;
GS.selector := CR_SAVE_GS.selector;
GS.base := CR_SAVE_GS.base;
GS.limit := CR_SAVE_GS.limit;
GS.access_rights := CR_SAVE_GS.access_rights;

(* Examine exception code and update enclave internal states*)
exception_code := Exception or interrupt vector;

(* Indicate the exit reason in SSA *)
IF (exception_code = (#DE OR #DB OR #BP OR #BR OR #UD OR #MF OR #AC OR #XM ))
THEN
    CR_GPR_PA.EXITINFO.VECTOR := exception_code;
    IF (exception_code = #BP)
        THEN CR_GPR_PA.EXITINFO.EXIT_TYPE := 6;
        ELSE CR_GPR_PA.EXITINFO.EXIT_TYPE := 3;
    FI;
    CR_GPR_PA.EXITINFO.VALID := 1;
ELSE IF (exception_code is #PF or #GP )
THEN
    (* Check SECS.MISCSELECT using CR_ACTIVE_SECS *)
    IF (SECS.MISCSELECT[0] is set)
        THEN
            CR_GPR_PA.EXITINFO.VECTOR := exception_code;
            CR_GPR_PA.EXITINFO.EXIT_TYPE := 3;
            IF (exception_code is #PF)
                THEN
                    SSA.MISC.EXINFO.MADDR := CR2;
                    SSA.MISC.EXINFO.ERRCD := PFEC;
                    SSA.MISC.EXINFO.RESERVED := 0;
                ELSE
                    SSA.MISC.EXINFO.MADDR := 0;
                    SSA.MISC.EXINFO.ERRCD := GPEC;
                    SSA.MISC.EXINFO.RESERVED := 0;
                FI;
            CR_GPR_PA.EXITINFO.VALID := 1;
        ELSE IF (exception_code is #CP)
            THEN
                IF (SECS.MISCSELECT[1] is set)
                    THEN
                        CR_GPR_PA.EXITINFO.VECTOR := exception_code;
                        CR_GPR_PA.EXITINFO.EXIT_TYPE := 3;
                        CR_GPR_PA.EXITINFO.VALID := 1;
                        SSA.MISC.EXINFO.MADDR := 0;
                        SSA.MISC.EXINFO.ERRCD := CPEC;
                        SSA.MISC.EXINFO.RESERVED := 0;
                    FI;
                FI;
            ELSE
```

```

    CR_GPR_PA.EXITINFO.VECTOR := 0;
    CR_GPR_PA.EXITINFO.EXIT_TYPE := 0
    CR_GPR_PA.REASON.VALID := 0;
FI;

(* Execution will resume at the AEP *)
RIP := CR_TCS_PA.AEP;

(* Set EAX to the ERESUME leaf index *)
EAX := 3;

(* Put the TCS LA into RBX for later use by ERESUME *)
RBX := CR_TCS_LA;

(* Put the AEP into RCX for later use by ERESUME *)
RCX := CR_TCS_PA.AEP;

(* Increment the SSA frame # *)
CR_TCS_PA.CSSA := CR_TCS_PA.CSSA + 1;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
    THEN XCR0 := CR_SAVE_XCR0; FI;

Un-suppress all code breakpoints that are outside ELRANGE

IF (CPUID.(EAX=12H, ECX=1):EAX[6]= 1)
    THEN
        IF (CR4.CET == 1 AND IA32_U_CET.SH_STK_EN == 1)
            THEN
                CR_CET_SAVE_AREA_PA.SSP := SSP;
                CR_TCS_PA.PREVSSP := SSP;
            FI;
        IF (CR4.CET == 1 AND IA32_U_CET.ENDBR_EN == 1)
            THEN
                CR_CET_SAVE_AREA_PA.TRACKER := IA32_U_CET.TRACKER;
                CR_CET_SAVE_AREA_PA.SUPPRESS := IA32_U_CET.SUPPRESS;
            FI;
        FI;
    FI;
IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* restore enclosing applications CET state *)
        IA32_U_CET := CR_SAVE_IA32_U_CET;

        IF (CPUID.(EAX=7, ECX=0):ECX[CET_SS])
            SSP := CR_SAVE_SSP; FI;

        (* If indirect branch tracking enabled for enclosing application *)
        (* then move the tracker to wait_for_endbranch *)
        IF (CR4.CET == 1 AND IA32_U_CET.ENDBR_EN == 1)
            THEN
                IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS := 0;
            FI;
    FI;

```

## ENCLAVE EXITING EVENTS

FI;

(\* Update the thread context to show not in enclave mode \*)  
CR\_ENCLAVE\_MODE := 0;

(\* Assure consistent translations. \*)  
Flush linear context including TLBs and paging-structure caches

IF (CR\_DBGOPTIN = 0)

THEN

Un-suppress all breakpoints that overlap ELRANGE

(\* Clear suppressed breakpoint matches \*)

Restore suppressed breakpoint matches

(\* Restore TF \*)

RFLAGS.TF := CR\_SAVE\_TF;

Un-suppress monitor trap flag;

Un-suppress branch recording facilities;

Un-suppress all suppressed performance monitoring activity;

Promote any sibling-thread counters that were demoted from AnyThread to MyThread during enclave

entry back to AnyThread;

FI;

IF the "monitor trap flag" VM-execution control is 1

THEN Pend MTF VM Exit at the end of exit; FI;

(\* Clear low 12 bits of CR2 on #PF \*)

IF (Exception code is #PF)

THEN CR2 := CR2 & ~0xFFF; FI;

(\* end\_of\_flow \*)

(\* Execution continues with normal event processing. \*)

## 27. Updates to Chapter 36, Volume 3D

Change bars and green text show changes to Chapter 36 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: Updated EAUG, EENTER, EEXIT, and ERESUME with CET and Intel SGX interaction clarifications.



## CHAPTER 36

# INTEL® SGX INSTRUCTION REFERENCES

This chapter describes the supervisor and user level instructions provided by Intel® Software Guard Extensions (Intel® SGX). In general, various functionality is encoded as leaf functions within the ENCLS (supervisor), ENCLU (user), and the ENCLV (virtualization operation) instruction mnemonics. Different leaf functions are encoded by specifying an input value in the EAX register of the respective instruction mnemonic.

## 36.1 INTEL® SGX INSTRUCTION SYNTAX AND OPERATION

ENCLS, ENCLU and ENCLV instruction mnemonics for all leaf functions are covered in this section.

For all instructions, the value of CS.D is ignored; addresses and operands are 64 bits in 64-bit mode and are otherwise 32 bits. Aside from EAX specifying the leaf number as input, each instruction leaf may require all or some subset of the RBX/RCX/RDX as input parameters. Some leaf functions may return data or status information in one or more of the general purpose registers.

### 36.1.1 ENCLS Register Usage Summary

Table 36-1 summarizes the implicit register usage of supervisor mode enclave instructions.

**Table 36-1. Register Usage of Privileged Enclave Instruction Leaf Functions**

Instr. Leaf	EAX	RBX	RCX	RDX
ECREATE	00H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EADD	01H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EINIT	02H (In)	SIGSTRUCT (In, EA)	SECS (In, EA)	EINITTOKEN (In, EA)
EREMOVE	03H (In)		EPCPAGE (In, EA)	
EDBGGRD	04H (In)	Result Data (Out)	EPCPAGE (In, EA)	
EDBGWR	05H (In)	Source Data (In)	EPCPAGE (In, EA)	
EEXTEND	06H (In)	SECS (In, EA)	EPCPAGE (In, EA)	
ELDB	07H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDU	08H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
EBLOCK	09H (In)		EPCPAGE (In, EA)	
EPA	0AH (In)	PT_VA (In)	EPCPAGE (In, EA)	
EWB	0BH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ETRACK	0CH (In)		EPCPAGE (In, EA)	
EAUG	0DH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EMODPR	0EH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODT	0FH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
ERDINFO	010H (In)	RDINFO (In, EA*)	EPCPAGE (In, EA)	
ETRACKC	011H (In)		EPCPAGE (In, EA)	
ELDBC	012H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDUC	013H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)

EA: Effective Address

### 36.1.2 ENCLU Register Usage Summary

Table 36-2 summarizes the implicit register usage of user mode enclave instructions.

**Table 36-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions**

Instr. Leaf	EAX	RBX	RCX	RDY
EReport	00H (In)	TARGETINFO (In, EA)	REPORTDATA (In, EA)	OUTPUTDATA (In, EA)
EGetKey	01H (In)	KEYREQUEST (In, EA)	KEY (In, EA)	
EEnter	02H (In)	TCS (In, EA)	AEP (In, EA)	
	RBX.CSSA (Out)		Return (Out, EA)	
EResume	03H (In)	TCS (In, EA)	AEP (In, EA)	
EExit	04H (In)	Target (In, EA)	Current AEP (Out)	
EAccept	05H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EModPr	06H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EAcceptCopy	07H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	EPCPAGE (In, EA)
EA: Effective Address				

### 36.1.3 ENCLV Register Usage Summary

Table 36-3 summarizes the implicit register usage of virtualization operation enclave instructions.

**Table 36-3. Register Usage of Virtualization Operation Enclave Instruction Leaf Functions**

Instr. Leaf	EAX	RBX	RCX	RDY
EDECvirtChild	00H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
EINCvirtChild	01H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
ESetContext	02H (In)		EPCPAGE (In, EA)	Context Value (In, EA)
EA: Effective Address				

### 36.1.4 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 36-4 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

**Table 36-4. Error or Information Codes for Intel® SGX Instructions**

Name	Value	Returned By
No Error	0	
SGX_INVALID_SIG_STRUCT	1	EINIT
SGX_INVALID_ATTRIBUTE	2	EINIT, EGETKEY
SGX_BLKSTATE	3	EBLOCK
SGX_INVALID_MEASUREMENT	4	EINIT
SGX_NOTBLOCKABLE	5	EBLOCK
SGX_PG_INVLD	6	EBLOCK, ERDINFO, ETRACKC
SGX_EPC_PAGE_CONFLICT	7	EBLOCK, EModPr, EModT, ERDINFO, EDECvirtChild, EINCvirtChild, ELDBC, ELDUC, ESetContext, ETRACKC

**Table 36-4. Error or Information Codes for Intel® SGX Instructions**

Name	Value	Returned By
SGX_INVALID_SIGNATURE	8	EINIT
SGX_MAC_COMPARE_FAIL	9	ELDB, ELDU, ELDBC, ELDUC
SGX_PAGE_NOT_BLOCKED	10	EWB
SGX_NOT_TRACKED	11	EWB, EACCEPT
SGX_VA_SLOT_OCCUPIED	12	EWB
SGX_CHILD_PRESENT	13	EWB, EREMOVE
SGX_ENCLAVE_ACT	14	EREMOVE
SGX_ENTRYEPOCH_LOCKED	15	EBLOCK
SGX_INVALID_EINITTOKEN	16	EINIT
SGX_PREV_TRK_INCMPL	17	ETRACK, ETRACKC
SGX_PG_IS_SECS	18	EBLOCK
SGX_PAGE_ATTRIBUTES_MISMATCH	19	EACCEPT, EACCEPTCOPY
SGX_PAGE_NOT_MODIFIABLE	20	EMODPR, EMODT
SGX_PAGE_NOT_DEBUGGABLE	21	EDBGRD, EDBGWR
SGX_INVALID_COUNTER	25	EDECVRTCHILD
SGX_PG_NONEPC	26	ERDINFO
SGX_TRACK_NOT_REQUIRED	27	ETRACKC
SGX_INVALID_CPUSVN	32	EINIT, EGETKEY
SGX_INVALID_ISVSVN	64	EGETKEY
SGX_UNMASKED_EVENT	128	EINIT
SGX_INVALID_KEYNAME	256	EGETKEY

### 36.1.5 Internal CREGs

The CREGs as shown in Table 5-4 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 36-5 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

**Table 36-5. List of Internal CREG**

Name	Size (Bits)	Scope
CR_ENCLAVE_MODE	1	LP
CR_DBGOPTIN	1	LP
CR_TCS_LA	64	LP
CR_TCS_PA	64	LP
CR_ACTIVE_SECS	64	LP
CR_EL RANGE	128	LP
CR_SAVE_TF	1	LP
CR_SAVE_FS	64	LP
CR_GPR_PA	64	LP
CR_XSAVE_PAGE_n	64	LP
CR_SAVE_DR7	64	LP
CR_SAVE_PERF_GLOBAL_CTRL	64	LP

Table 36-5. List of Internal CREG

Name	Size (Bits)	Scope
CR_SAVE_DEBUGCTL	64	LP
CR_SAVE_PEBS_ENABLE	64	LP
CR_CPUSVN	128	PACKAGE
CR_SGXOWNERPOCH	128	PACKAGE
CR_SAVE_XCRO	64	LP
CR_SGX_ATTRIBUTES_MASK	128	LP
CR_PAGING_VERSION	64	PACKAGE
CR_VERSION_THRESHOLD	64	PACKAGE
CR_NEXT_EID	64	PACKAGE
CR_BASE_PK	128	PACKAGE
CR_SEAL_FUSES	128	PACKAGE
CR_CET_SAVE_AREA_PA	64	LP
CR_ENCLAVE_SS_TOKEN_PA	64	LP
CR_SAVE_IA32_U_CET	64	LP
CR_SAVE_SSP	64	LP

### 36.1.6 Concurrent Operation Restrictions

Under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leaves to concurrently operate on the same EPC page.
  - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
  - EADD, EEXTEND, and EINIT leaves are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function may do one of the following:

- Return an SGX\_EPC\_PAGE\_CONFLICT error code in RAX.
- Cause a #GP(0) exception.

To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same EPC page.

#### 36.1.6.1 Concurrency Tables of Intel® SGX Instructions

The tables below detail the concurrent operation restrictions of all SGX leaf functions. For each leaf function, the table has a separate line for each of the EPC pages the leaf function accesses.

For each such EPC page, the base concurrency requirements are detailed as follows:

- **Exclusive Access** means that no other leaf function that requires either shared or exclusive access to the same EPC page may be executed concurrently. For example, EADD requires an exclusive access to the target page it accesses.
- **Shared Access** means that no other leaf function that requires an exclusive access to the same EPC page may be executed concurrently. Other leaf functions that require shared access may run concurrently. For example, EADD requires a shared access to the SECS page it accesses.

- **Concurrent Access** means that any other leaf function that requires any access to the same EPC page may be executed concurrently. For example, EGETKEY has no concurrency requirements for the KEYREQUEST page.

In addition to the base concurrency requirements, additional concurrency requirements are listed, which apply only to specific sets of leaf functions. For example, there are additional requirements that apply for EADD, EXTEND and EINIT. EADD and EEXTEND can't execute concurrently on the same SECS page.

The tables also detail the leaf function's behavior when a conflict happens, i.e., a concurrency requirement is not met. In this case, the leaf function may return an SGX\_EPC\_PAGE\_CONFLICT error code in RAX, or it may cause an exception. In addition, the tables detail those conflicts where a VM Exit may be triggered, and list the Exit Qualification code that is provided in such cases.

**Table 36-6. Base Concurrency Restrictions**

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target	[DS:RCX]	Shared	#GP	
	SECINFO	[DS:RBX]	Concurrent		
EACCEPTCOPY	Target	[DS:RCX]	Concurrent		
	Source	[DS:RDX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EADD	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EAUG	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EBLOCK	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
ECREATE	SECS	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EDBGGRD	Target	[DS:RCX]	Shared	#GP	
EDBGWR	Target	[DS:RCX]	Shared	#GP	
EDECVRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EENTERTCS	SECS	[DS:RBX]	Shared	#GP	
EEXIT			Concurrent		
EEXTEND	Target	[DS:RCX]	Shared	#GP	
	SECS	[DS:RBX]	Concurrent		
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		
	OUTPUTDATA	[DS:RCX]	Concurrent		
EINCVIRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EINIT	SECS	[DS:RCX]	Shared	#GP	
ELDB/ELDU	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	

**Table 36-6. Base Concurrency Restrictions**

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDLBC/ELDUC	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA	[DS:RDX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	SGX_EPC_PAGE_CONFLICT	
EMODPE	Target	[DS:RCX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EMODPR	Target	[DS:RCX]	Shared	#GP	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR
EPA	VA	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
ERDINFO	Target	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
EREMOVE	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		
	REPORTDATA	[DS:RCX]	Concurrent		
	OUTPUTDATA	[DS:RDX]	Concurrent		
ERESUME	TCS	[DS:RBX]	Shared	#GP	
ESETCONTEXT	SECS	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
ETRACK	SECS	[DS:RCX]	Shared	#GP	
ETRACKC	Target	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS	Implicit	Concurrent		
EWB	Source	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	

**Table 36-7. Additional Concurrency Restrictions**

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EACCEPTCOPY	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	

Table 36-7. Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EADD	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Exclusive	#GP	Concurrent	
EAUG	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EBLOCK	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ECREATE	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGDR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGWR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDECVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EENTERTCS	SECS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EEXIT			Concurrent		Concurrent		Concurrent	
EEXTEND	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINCVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINIT	SECS	[DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	
ELDB/ELDU	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EDLBC/ELDUC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EMODPE	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EMODPR	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CON FLICT	Concurrent		Concurrent	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CON FLICT	Concurrent		Concurrent	
EPA	VA	[DS:RCX]	Concurrent		Concurrent		Concurrent	

**Table 36-7. Additional Concurrency Restrictions**

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREMOVE	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
ERESUME	TCS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
ESETCONTEXT	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ETRACK	SECS	[DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT <sup>1</sup>
ETRACKC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	Implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT <sup>1</sup>
EWB	Source	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	

**NOTES:**

1. SGX\_CONFLICT VM Exit Qualification =TRACKING\_RESOURCE\_CONFLICT.

## 36.2 INTEL® SGX INSTRUCTION REFERENCE



## ENCLS—Execute an Enclave System Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CF ENCLS	Z0	V/V	NA	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
Z0	NA	NA	NA	See Section 36.3

### Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the “enable ENCLS exiting” VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the “enable ENCLS exiting” VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32\_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32\_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

### Operation

IF TSX\_ACTIVE

THEN GOTO TSX\_ABORT\_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX\_LEAF.0:EAX.SE1 = 0

THEN #UD; FI;

IF (CPL > 0)

THEN #UD; FI;

IF in VMX non-root operation and the “enable ENCLS exiting” VM-execution control is 1

THEN

IF EAX < 63 and ENCLS\_exiting\_bitmap[EAX] = 1 or EAX > 62 and ENCLS\_exiting\_bitmap[63] = 1

THEN VM exit;

FI;

FI;

IF IA32\_FEATURE\_CONTROL.LOCK = 0 or IA32\_FEATURE\_CONTROL.SGX\_ENABLE = 0

THEN #GP(0); FI;

IF (EAX is an invalid leaf number)

THEN #GP(0); FI;

IF CR0.PG = 0  
 THEN #GP(0); FI;

(\* DS must not be an expanded down segment \*)  
 IF not in 64-bit mode and DS.Type is expand-down data  
 THEN #GP(0); FI;

Jump to leaf specific flow

### Flags Affected

See individual leaf functions

### Protected Mode Exceptions

#UD                    If any of the LOCK/66H/REP/VEX prefixes are used.  
                           If current privilege level is not 0.  
                           If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.  
                           If logical processor is in SMM.

#GP(0)                If IA32\_FEATURE\_CONTROL.LOCK = 0.  
                           If IA32\_FEATURE\_CONTROL.SGX\_ENABLE = 0.  
                           If input value in EAX encodes an unsupported leaf.  
                           If data segment expand down.  
                           If CR0.PG=0.

### Real-Address Mode Exceptions

#UD                    ENCLS is not recognized in real mode.

### Virtual-8086 Mode Exceptions

#UD                    ENCLS is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD                    If any of the LOCK/66H/REP/VEX prefixes are used.  
                           If current privilege level is not 0.  
                           If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.  
                           If logical processor is in SMM.

#GP(0)                If IA32\_FEATURE\_CONTROL.LOCK = 0.  
                           If IA32\_FEATURE\_CONTROL.SGX\_ENABLE = 0.  
                           If input value in EAX encodes an unsupported leaf.

## ENCLU—Execute an Enclave User Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D7 ENCLU	Z0	V/V	NA	This instruction is used to execute non-privileged Intel SGX leaf functions.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
Z0	NA	NA	NA	See Section 36.4

### Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32\_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32\_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

### Operation

```
IN_64BIT_MODE := 0;
```

```
IF TSX_ACTIVE
```

```
    THEN GOTO TSX_ABORT_PROCESSING; FI;
```

(\* If enclosing app has CET indirect branch tracking enabled then if it is not ERESUME leaf cause a #CP fault \*)

(\* If the ERESUME is not successful it will leave tracker in WAIT\_FOR\_ENDBRANCH \*)

```
TRACKER = (CPL == 3) ? IA32_U_CET.TRACKER : IA32_S_CET.TRACKER
```

```
IF EndbranchEnabledAndNotSuppressed(CPL) and TRACKER = WAIT_FOR_ENDBRANCH and  
(EAX != ERESUME or CR0.TS or (in SMM) or (CPUID.SGX_LEAF.0:EAX.SE1 = 0) or (CPL < 3))
```

```
    THEN
```

```
        Handle CET State machine violation          (* see Section 18.3.6, "Legacy Compatibility Treatment" in the  
                                                    Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. *)
```

```
    FI;
```

```
IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
```

```
    THEN #UD; FI;
```

```
IF CR0.TS = 1
```

```
    THEN #NM; FI;
```

```
IF CPL < 3
```

```
    THEN #UD; FI;
```

```
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
```

```
    THEN #GP(0); FI;
```

IF EAX is invalid leaf number  
THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0  
THEN #GP(0); FI;

IN\_64BIT\_MODE := IA32\_EFER.LMA AND CS.L ? 1 : 0;  
(\* Check not in 16-bit mode and DS is not a 16-bit segment \*)  
IF not in 64-bit mode and CS.D = 0  
THEN #GP(0); FI;

IF CR\_ENCLAVE\_MODE = 1 and (EAX = 2 or EAX = 3) (\* EENTER or ERESUME \*)  
THEN #GP(0); FI;

IF CR\_ENCLAVE\_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or EAX = 6 or EAX = 7)  
(\* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, or EACCEPTCOPY \*)  
THEN #GP(0); FI;

Jump to leaf specific flow

### Flags Affected

See individual leaf functions

### Protected Mode Exceptions

#UD	<p>If any of the LOCK/66H/REP/VEX prefixes are used. If current privilege level is not 3. If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0. If logical processor is in SMM.</p>
#GP(0)	<p>If IA32_FEATURE_CONTROL.LOCK = 0. If IA32_FEATURE_CONTROL.SGX_ENABLE = 0. If input value in EAX encodes an unsupported leaf. If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1. If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0. If operating in 16-bit mode. If data segment is in 16-bit mode. If CR0.PG = 0 or CR0.NE = 0.</p>
#NM	<p>If CR0.TS = 1.</p>

### Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.
-----	---------------------------------------

### Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.
-----	---

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD	<p>If any of the LOCK/66H/REP/VEX prefixes are used.</p> <p>If current privilege level is not 3.</p> <p>If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.</p> <p>If logical processor is in SMM.</p>
#GP(0)	<p>If IA32_FEATURE_CONTROL.LOCK = 0.</p> <p>If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.</p> <p>If input value in EAX encodes an unsupported leaf.</p> <p>If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.</p> <p>If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.</p> <p>If CR0.NE = 0.</p>
#NM	<p>If CR0.TS = 1.</p>

## ENCLV—Execute an Enclave VMM Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 01 C0 ENCLV	Z0	V/V	NA	This instruction is used to execute privileged SGX leaf functions that are reserved for VMM use. They are used for managing the enclaves.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
Z0	NA	NA	NA	See Section 36.3

### Description

The ENCLV instruction invokes the virtualization SGX leaf functions for managing enclaves in a virtualized environment. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In non 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLV instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, if it is executed in system-management mode (SMM), or not in VMX operation. Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

Software in VMX root mode of operation can enable execution of the ENCLV instruction in VMX non-root mode by setting enable ENCLV execution control in the VMCS. If enable ENCLV execution control in the VMCS is clear, execution of the ENCLV instruction in VMX non-root mode results in #UD.

When execution of ENCLV instruction in VMX non-root mode is enabled, software in VMX root operation can intercept the invocation of various ENCLV leaf functions in VMX non-root operation by setting the corresponding bits in the ENCLV-exiting bitmap.

Addresses and operands are 32 bits in 32-bit mode (IA32\_EFER.LMA == 0 || CS.L == 0) and are 64 bits in 64-bit mode (IA32\_EFER.LMA == 1 && CS.L == 1). CS.D value has no impact on address calculation.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

### Operation

IF TSX\_ACTIVE

THEN GOTO TSX\_ABORT\_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX\_LEAF.0:EAX.OSS = 0

THEN #UD; FI;

IF not in VMX Operation or (IA32\_EFER.LMA = 1 and CS.L = 0)

THEN #UD; FI;

IF (CPL > 0)

THEN #UD; FI;

IF in VMX non-root operation

IF “enable ENCLV exiting” VM-execution control is 1

THEN

IF EAX < 63 and ENCLV\_exiting\_bitmap[EAX] = 1 or EAX > 62 and ENCLV\_exiting\_bitmap[63] = 1

THEN VM exit;

FI;

ELSE

#UD; FI;

FI;

IF IA32\_FEATURE\_CONTROL.LOCK = 0 or IA32\_FEATURE\_CONTROL.SGX\_ENABLE = 0  
THEN #GP(0); FI;

IF (EAX is an invalid leaf number)  
THEN #GP(0); FI;

IF CR0.PG = 0  
THEN #GP(0); FI;

(\* DS must not be an expanded down segment \*)  
IF not in 64-bit mode and DS.Type is expand-down data  
THEN #GP(0); FI;

Jump to leaf specific flow

### Flags Affected

See individual leaf functions.

### Protected Mode Exceptions

#UD	<p>If any of the LOCK/66H/REP/VEX prefixes are used. If current privilege level is not 0. If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0. If logical processor is in SMM.</p>
#GP(0)	<p>If IA32_FEATURE_CONTROL.LOCK = 0. If IA32_FEATURE_CONTROL.SGX_ENABLE = 0. If input value in EAX encodes an unsupported leaf. If data segment expand down. If CR0.PG=0.</p>

### Real-Address Mode Exceptions

#UD	ENCLV is not recognized in real mode.
-----	---------------------------------------

### Virtual-8086 Mode Exceptions

#UD	ENCLV is not recognized in virtual-8086 mode.
-----	---

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD	<p>If any of the LOCK/66H/REP/VEX prefixes are used. If current privilege level is not 0. If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0. If logical processor is in SMM.</p>
#GP(0)	<p>If IA32_FEATURE_CONTROL.LOCK = 0. If IA32_FEATURE_CONTROL.SGX_ENABLE = 0. If input value in EAX encodes an unsupported leaf.</p>

### 36.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLS instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.



## EADD—Add a Page to an Uninitialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLS[EADD]	IR	V/V	SGX1	This leaf function adds a page to an uninitialized enclave.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EADD (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function copies a source page from non-enclave memory into the EPC, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in EPCM. As part of the association, the enclave offset and the security attributes are measured and extended into the SECS.MRENCLAVE. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of EADD leaf function.

### EADD Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

### EADD Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	If security attributes specifies a TCS and the source page specifies unsupported TCS values or fields.
The SECS has been initialized.	The specified enclave offset is outside of the enclave address space.

### Concurrency Restrictions

**Table 36-8. Base Concurrency Restrictions of EADD**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EADD	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

**Table 36-9. Additional Concurrency Restrictions of EADD**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EADD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Exclusive	#GP	Concurrent	

**Operation**

**Temp Variables in EADD Operational Flow**

Name	Type	Size (bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

IF (DS:RBX is not 32Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

TMP\_SRCPGE := DS:RBX.SRCPGE;  
TMP\_SECS := DS:RBX.SECS;  
TMP\_SECINFO := DS:RBX.SECINFO;  
TMP\_LINADDR := DS:RBX.LINADDR;

IF (DS:TMP\_SRCPGE is not 4KByte aligned or DS:TMP\_SECS is not 4KByte aligned or DS:TMP\_SECINFO is not 64Byte aligned or TMP\_LINADDR is not 4KByte aligned)  
THEN #GP(0); FI;

IF (DS:TMP\_SECS does not resolve within an EPC)  
THEN #PF(DS:TMP\_SECS); FI;

SCRATCH\_SECINFO := DS:TMP\_SECINFO;

(\* Check for misconfigured SECINFO flags\*)  
IF (SCRATCH\_SECINFO reserved fields are not zero or

```

!(SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS or
(SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
(SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
THEN #GP(0); FI;

```

```

(* If PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST OR
SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST) AND CR4.CET == 0)
THEN #GP(0); FI;

```

```

(* Check the EPC page for concurrency *)
IF (EPC page is not available for EADD)
THEN
  IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
  THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
    VMCS.Guest-linear_address := DS:RCX;
    Deliver VMEXIT;
  ELSE
    #GP(0);
  FI;
FI;

```

```

IF (EPCM(DS:RCX).VALID ≠ 0)
THEN #PF(DS:RCX); FI;

```

```

(* Check the SECS for concurrency *)
IF (SECS is not available for EADD)
THEN #GP(0); FI;

```

```

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
THEN #PF(DS:TMP_SECS); FI;

```

```

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPGE[32767:0];

```

```

CASE (SCRATCH_SECINFO.FLAGS.PT)

```

```

PT_TCS:
  IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
  IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and
((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH)) ) #GP(0); FI;
  (* Ensure TCS.PREVSSP is zero *)
  IF (CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1) and (DS:RCX.PREVSSP != 0) #GP(0); FI;
  BREAK;

```

```

PT_REG:
  IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #GP(0); FI;
  BREAK;

```

```

PT_SS_FIRST:

```

```

PT_SS_REST:

```

```

(* SS pages cannot be created on first or last page of ELRANGE *)

```

```

IF ( TMP_LINADDR = DS:TMP_SECS.BASEADDR or TMP_LINADDR = (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
  THEN #GP(0); FI;
IF ( DS:RCX[4087:0] != 0 ) #GP(0); FI;
IF (SCRATCH_SECINFO.FLAGS.PT == PT_SS_FIRST)
  THEN
    (* Check that valid RSTORSSP token exists *)
    IF ( DS:RCX[4095:4088] != ((TMP_LINADDR + 0x1000) | DS:TMP_SECS.ATTRIBUTES.MODE64BIT) ) #GP(0); FI;
    (* Check the 8 bytes are zero *)
    IF ( DS:RCX[4095:4088] != 0 ) #GP(0); FI;
  FI;
IF (SCRATCH_SECINFO.FLAGS.W = 0 OR SCRATCH_SECINFO.FLAGS.R = 0 OR
  SCRATCH_SECINFO.FLAGS.X = 1) #GP(0); FI;
  BREAK;
ESAC;

```

```

(* Check the enclave offset is within the enclave linear address space *)
IF (TMP_LINADDR < DS:TMP_SECS.BASEADDR or TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE)
  THEN #GP(0); FI;

```

```

(* Check concurrency of measurement resource*)
IF (Measurement being updated)
  THEN #GP(0); FI;

```

```

(* Check if the enclave to which the page will be added is already in Initialized state *)
IF (DS:TMP_SECS already initialized)
  THEN #GP(0); FI;

```

```

(* For TCS pages, force EPCM.rwx bits to 0 and no debug access *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
  THEN
    SCRATCH_SECINFO.FLAGS.R := 0;
    SCRATCH_SECINFO.FLAGS.W := 0;
    SCRATCH_SECINFO.FLAGS.X := 0;
    (DS:RCX).FLAGS.DBGOPTIN := 0; // force TCS.FLAGS.DBGOPTIN off
    DS:RCX.CSSA := 0;
    DS:RCX.AEP := 0;
    DS:RCX.STATE := 0;
  FI;

```

```

(* Add enclave offset and security attributes to MRENCLAVE *)
TMP_ENCLAVEOFFSET := TMP_LINADDR - DS:TMP_SECS.BASEADDR;
TMPUPDATEFIELD[63:0] := 0000000044444145H; // "EADD"
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] := SCRATCH_SECINFO[375:0]; // 48 bytes
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

```

```

(* Add enclave offset and security attributes to MRENCLAVE *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;

```

(\* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP\_SECS \*)  
 Update EPCM(DS:RCX) SECS identifier to reference DS:TMP\_SECS identifier;

(\* Set EPCM entry fields \*)  
 EPCM(DS:RCX).BLOCKED := 0;  
 EPCM(DS:RCX).PENDING := 0;  
 EPCM(DS:RCX).MODIFIED := 0;  
 EPCM(DS:RCX).VALID := 1;

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)                If a memory operand effective address is outside the DS segment limit.  
                           If a memory operand is not properly aligned.  
                           If an enclave memory operand is outside of the EPC.  
                           If an enclave memory operand is the wrong type.  
                           If a memory operand is locked.  
                           If the enclave is initialized.  
                           If the enclave's MRENCLAVE is locked.  
                           If the TCS page reserved bits are set.  
                           If the TCS page PREVSSP field is not zero.  
                           If the PT\_SS\_REST or PT\_SS\_REST page is the first or last page in the enclave.  
                           If the PT\_SS\_FIRST or PT\_SS\_REST page is not initialized correctly.

#PF(error code)    If a page fault occurs in accessing memory operands.  
                           If the EPC page is valid.

### 64-Bit Mode Exceptions

#GP(0)                If a memory operand is non-canonical form.  
                           If a memory operand is not properly aligned.  
                           If an enclave memory operand is outside of the EPC.  
                           If an enclave memory operand is the wrong type.  
                           If a memory operand is locked.  
                           If the enclave is initialized.  
                           If the enclave's MRENCLAVE is locked.  
                           If the TCS page reserved bits are set.  
                           If the TCS page PREVSSP field is not zero.  
                           If the PT\_SS\_REST or PT\_SS\_REST page is the first or last page in the enclave.  
                           If the PT\_SS\_FIRST or PT\_SS\_REST page is not initialized correctly.

#PF(error code)    If a page fault occurs in accessing memory operands.  
                           If the EPC page is valid.

## EAUG—Add a Page to an Initialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0DH ENCLS[EAUG]	IR	V/V	SGX2	This leaf function adds a page to an initialized enclave.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EAUG (In)	Address of a SECFINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPT-COPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

### EAUG Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Must be zero	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

### EAUG Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	The specified enclave offset is outside of the enclave address space.
The SECS has been initialized.	

### Concurrency Restrictions

**Table 36-10. Base Concurrency Restrictions of EAUG**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EAUG	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

**Table 36-11. Additional Concurrency Restrictions of EAUG**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EAUG	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Concurrent		Concurrent	

**Operation****Temp Variables in EAUG Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.

IF (DS:RBX is not 32Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

TMP\_SECS := DS:RBX.SECS;  
TMP\_SECINFO := DS:RBX.SECINFO;  
IF (DS:RBX.SECINFO is not 0)  
THEN  
    IF (DS:TMP\_SECINFO is not 64B aligned)  
    THEN #GP(0); FI;

FI;

TMP\_LINADDR := DS:RBX.LINADDR;

IF ( DS:TMP\_SECS is not 4KByte aligned or TMP\_LINADDR is not 4KByte aligned )  
THEN #GP(0); FI;

IF DS:RBX.SRCPAGE is not 0  
THEN #GP(0); FI;

IF (DS:TMP\_SECS does not resolve within an EPC)  
THEN #PF(DS:TMP\_SECS); FI;

(\* Check the EPC page for concurrency \*)

```

IF (EPC page in use)
  THEN
    IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
      THEN
        VMCS.Exit_reason := SGX_CONFLICT;
        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
        VMCS.Exit_qualification.error := 0;
        VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
        VMCS.Guest-linear_address := DS:RCX;
        Deliver VMEXIT;
      ELSE
        #GP(0);
    FI;
  FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
  THEN #PF(DS:RCX); FI;

(* copy SECINFO contents into a scratch SECINFO *)
IF (DS:RBX.SECINFO is 0)
  THEN
    (* allocate and initialize a new scratch SECINFO structure *)
    SCRATCH_SECINFO.PT := PT_REG;
    SCRATCH_SECINFO.R := 1;
    SCRATCH_SECINFO.W := 1;
    SCRATCH_SECINFO.X := 0;
    << zero out remaining fields of SCRATCH_SECINFO >>
  ELSE
    (* copy SECINFO contents into scratch SECINFO *)
    SCRATCH_SECINFO := DS:TMP_SECINFO;
    (* check SECINFO flags for misconfiguration *)
    (* reserved flags must be zero *)
    (* SECINFO.FLAGS.PT must either be PT_SS_FIRST, or PT_SS_REST *)
    IF ( (SCRATCH_SECINFO reserved fields are not 0) OR
        CPUID.(EAX=12H, ECX=1):EAX[6] is 0) OR
        (SCRATCH_SECINFO.PT is not PT_SS_FIRST, or PT_SS_REST) OR
        ( (SCRATCH_SECINFO.FLAGS.R is 0) OR (SCRATCH_SECINFO.FLAGS.W is 0) OR (SCRATCH_SECINFO.FLAGS.X is 1) ) )
      THEN #GP(0); FI;
  FI;

(* Check if PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) AND CR4.CET == 0 )
  THEN #GP(0); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EAUG)
  THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
  THEN #PF(DS:TMP_SECS); FI;

(* Check if the enclave to which the page will be added is in the Initialized state *)
IF (DS:TMP_SECS is not initialized)
  THEN #GP(0); FI;

```



```
(* Check the enclave offset is within the enclave linear address space *)
IF ( (TMP_LINADDR < DS:TMP_SECS.BASEADDR) or (TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE) )
  THEN #GP(0); FI;
```

```
IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) )
  THEN
    (* SS pages cannot be created on first or last page of ELRANGE *)
    IF ( TMP_LINADDR == DS:TMP_SECS.BASEADDR OR
        TMP_LINADDR == (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
      THEN
        #GP(0); FI;
  FI;
```

```
(* Clear the content of EPC page*)
DS:RCX[32767:0] := 0;
```

```
(* Set EPCM security attributes *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 1;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
```

```
(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;
```

```
(* Set EPCM valid fields *)
EPCM(DS:RCX).VALID := 1;
```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked. If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

### 64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked. If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

## EBLOCK—Mark a page in EPC as Blocked

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 09H ENCLS[EBLOCK]	IR	V/V	SGX1	This leaf function marks a page in the EPC as blocked.

### Instruction Operand Encoding

Op/En	EAX		RCX
IR	EBLOCK (In)	Return error code (Out)	Effective address of the EPC page (In)

### Description

This leaf function causes an EPC page to be marked as BLOCKED. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

An error code is returned in RAX.

The table below provides additional information on the memory parameter of EBLOCK leaf function.

### EBLOCK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

**Table 36-12. EBLOCK Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EBLOCK successful.
SGX_BLKSTATE	Page already blocked. This value is used to indicate to a VMM that the page was already in BLOCKED state as a result of EBLOCK and thus will need to be restored to this state when it is eventually reloaded (using ELDB).
SGX_ENTRYEPOCH_LOCKED	SECS locked for Entry Epoch update. This value indicates that an ETRACK is currently executing on the SECS. The EBLOCK should be reattempted.
SGX_NOTBLOCKABLE	Page type is not one which can be blocked.
SGX_PG_INVLD	Page is not valid and cannot be blocked.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

### Concurrency Restrictions

**Table 36-13. Base Concurrency Restrictions of EBLOCK**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EBLOCK	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

**Table 36-14. Additional Concurrency Restrictions of EBLOCK**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EBLOCK	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation****Temp Variables in EBLOCK Operational Flow**

Name	Type	Size (Bits)	Description
TMP_BLKSTATE	Integer	64	Page is already blocked.

```
IF (DS:RCX is not 4KByte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
  THEN #PF(DS:RCX); FI;
```

```
RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;
```

(\* Check the EPC page for concurrency\*)

```
IF (EPC page in use)
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_EPC_PAGE_CONFLICT;
    GOTO DONE;
```

```
FI;
```

```
IF (EPCM(DS:RCX).VALID = 0)
```

```
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_PG_INVLD;
    GOTO DONE;
```

```
FI;
```

```
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_TRIM)
and EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
```

```
  THEN
    RFLAGS.CF := 1;
    IF (EPCM(DS:RCX).PT = PT_SECS)
      THEN RAX := SGX_PG_IS_SECS;
    ELSE RAX := SGX_NOTBLOCKABLE;
```

```
  FI;
  GOTO DONE;
```

```
FI;
```

(\* Check if the page is already blocked and report blocked state \*)

```
TMP_BLKSTATE := EPCM(DS:RCX).BLOCKED;
```

```
(* at this point, the page must be valid and PT_TCS or PT_REG or PT_TRIM*)
IF (TMP_BLKSTATE = 1)
  THEN
    RFLAGS.CF := 1;
    RAX := SGX_BLKSTATE;
  ELSE
    EPCM(DS:RCX).BLOCKED := 1
FI;
DONE:
```

### Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Sets CF if page is BLOCKED or not blockable, otherwise cleared. Clears PF, AF, OF, SF.

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If the specified EPC resource is in use.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If the specified EPC resource is in use.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> </ul>

## ECREATE—Create an SECS page in the Enclave Page Cache

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLS[ECREATE]	IR	V/V	SGX1	This leaf function begins an enclave build by creating an SECS page in EPC.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ECREATE (In)	Address of a PAGEINFO (In)	Address of the destination SECS page (In)

### Description

ENCLS[ECREATE] is the first instruction executed in the enclave build process. ECREATE copies an SECS structure outside the EPC into an SECS page inside the EPC. The internal structure of SECS is not accessible to software.

ECREATE will set up fields in the protected SECS and mark the page as valid inside the EPC. ECREATE initializes or checks unused fields.

Software sets the following fields in the source structure: SECS:BASEADDR, SECS:SIZE in bytes, ATTRIBUTES, CONFIGID and CONFIGSVN. SECS:BASEADDR must be naturally aligned on an SECS.SIZE boundary. SECS.SIZE must be at least 2 pages (8192).

The source operand RBX contains an effective address of a PAGEINFO structure. PAGEINFO contains an effective address of a source SECS and an effective address of an SECINFO. The SECS field in PAGEINFO is not used.

The RCX register is the effective address of the destination SECS. It is an address of an empty slot in the EPC. The SECS structure must be page aligned. SECINFO flags must specify the page as an SECS page.

### ECREATE Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

ECREATE will fault if the SECS target page is in use; already valid; outside the EPC. It will also fault if addresses are not aligned; unused PAGEINFO fields are not zero.

If the amount of space needed to store the SSA frame is greater than the amount specified in SECS.SSAFRAME-SIZE, a #GP(0) results. The amount of space needed for an SSA frame is computed based on DS:TMP\_SECS.ATTRIBUTES.XFRM size. Details of computing the size can be found Section 37.7.

### Concurrency Restrictions

**Table 36-15. Base Concurrency Restrictions of ECREATE**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ECREATE	SECS [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

**Table 36-16. Additional Concurrency Restrictions of ECREATE**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ECREATE	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in ECREATE Operational Flow**

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the SECS source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the SECS page to be added.
TMP_XSIZE	SSA Size	64	The size calculation of SSA frame.
TMP_MISC_SIZE	MISC Field Size	64	Size of the selected MISC field components.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

IF (DS:RBX is not 32Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

TMP\_SRCPGE := DS:RBX.SRCPGE;  
 TMP\_SECINFO := DS:RBX.SECINFO;

IF (DS:TMP\_SRCPGE is not 4KByte aligned or DS:TMP\_SECINFO is not 64Byte aligned)  
 THEN #GP(0); FI;

IF (DS:RBX.LINADDR != 0 or DS:RBX.SECS != 0)  
 THEN #GP(0); FI;

(\* Check for misconfigured SECINFO flags\*)

IF (DS:TMP\_SECINFO reserved fields are not zero or DS:TMP\_SECINFO.FLAGS.PT != PT\_SECS)  
 THEN #GP(0); FI;

TMP\_SECS := RCX;

IF (EPC entry in use)  
 THEN  
     IF (<<VMX non-root operation>> AND <<ENABLE\_EPC\_VIRTUALIZATION\_EXTENSIONS>>)  
         THEN  
             VMCS.Exit\_reason := SGX\_CONFLICT;

```

        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
        VMCS.Exit_qualification.error := 0;
        VMCS.Guest-physical_address :=
            << translation of DS:TMP_SECS produced by paging >>;
        VMCS.Guest-linear_address := DS:TMP_SECS;
    Deliver VMEXIT;
    ELSE
        #GP(0);
FI;

IF (EPC entry in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPAGE[32767:0];

(* Check lower 2 bits of XFRM are set *)
IF ( ( DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
    THEN #GP(0); FI;

IF (XFRM is illegal)
    THEN #GP(0); FI;

(* Check legality of CET_ATTRIBUTES *)
IF ((DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_ATTRIBUTES ≠ 0) ||
    (DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[5:2] ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[1:0] ≠ 0) ||
    (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1 and
    (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) not canonical) ||
    (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0 and
    (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) & 0xFFFFFFFF00000000) ||
    (DS:TMP_SECS.CET_ATTRIBUTES.reserved fields not 0) or
    (DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) is not page aligned))
    THEN
        #GP(0);
FI;

(* Make sure that the SECS does not have any unsupported MISCSELECT options*)
IF ( !(CPUID.(EAX=12H, ECX=0):EBX[31:0] & DS:TMP_SECS.MISCSELECT[31:0]) )
    THEN
        EPCM(DS:TMP_SECS).EntryLock.Release();
        #GP(0);
FI;

(* Compute size of MISC area *)
TMP_MISC_SIZE := compute_misc_region_size();

(* Compute the size required to save state of the enclave on async exit, see Section 37.7.2.2*)

```

```
TMP_XSIZE := compute_xsave_size(DS:TMP_SECS.ATTRIBUTES.XFRM) + GPR_SIZE + TMP_MISC_SIZE;
```

```
(* Ensure that the declared area is large enough to hold XSAVE and GPR stat *)
```

```
IF ( DS:TMP_SECS.SSAFRAMESIZE*4096 < TMP_XSIZE)
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.BASEADDR is not canonical) )
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.BASEADDR and 0FFFFFFF00000000H) )
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) ) )
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) ) )
```

```
    THEN #GP(0); FI;
```

```
(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
```

```
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
```

```
    THEN #GP(0); FI;
```

```
(* Ensure base address of an enclave is aligned on size*)
```

```
IF ( ( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) ) )
```

```
    THEN #GP(0); FI;
```

```
(* Ensure the SECS does not have any unsupported attributes*)
```

```
IF ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
```

```
    THEN #GP(0); FI;
```

```
IF ( DS:TMP_SECS reserved fields are not zero)
```

```
    THEN #GP(0); FI;
```

```
(* Verify that CONFIGID/CONFIGSVN are not set with attribute *)
```

```
IF ( ((DS:TMP_SECS.CONFIGID ≠ 0) or (DS:TMP_SECS.CONFIGSVN ≠ 0)) AND (DS:TMP_SECS.ATTRIBUTES.KSS == 0) )
```

```
    THEN #GP(0); FI;
```

```
Clear DS:TMP_SECS to Uninitialized;
```

```
DS:TMP_SECS.MRENCLAVE := SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
```

```
DS:TMP_SECS.ISVSVN := 0;
```

```
DS:TMP_SECS.ISVPRODID := 0;
```

```
(* Initialize hash updates etc*)
```

```
Initialize enclave's MRENCLAVE update counter;
```

```
(* Add "ECREATE" string and SECS fields to MRENCLAVE *)
```

```
TMPUPDATEFIELD[63:0] := 0045544145524345H; // "ECREATE"
```

```
TMPUPDATEFIELD[95:64] := DS:TMP_SECS.SSAFRAMESIZE;
```

```
TMPUPDATEFIELD[159:96] := DS:TMP_SECS.SIZE;
```

```
IF (CPUID.(EAX=7, ECX=0):.EDX[CET_IBT] = 1)
```

```
    THEN
```

```
        TMPUPDATEFIELD[223:160] := DS:TMP_SECS.CET_LEG_BITMAP_OFFSET;
```

```
    ELSE
```

```
        TMPUPDATEFIELD[223:160] := 0;
```



```

FI;
TMPUPDATEFIELD[511:160] := 0;
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

```

```

(* Set EID *)
DS:TMP_SECS.EID := LockedXAdd(CR_NEXT_EID, 1);

```

```

(* Initialize the virtual child count to zero *)
DS:TMP_SECS.VIRTCHILDCNT := 0;

```

```

(* Load ENCLAVECONTEXT with Address out of paging of SECS *)
<< store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>

```

```

(* Set the EPCM entry, first create SECS identifier and store the identifier in EPCM *)
EPCM(DS:TMP_SECS).PT := PT_SECS;
EPCM(DS:TMP_SECS).ENCLAVEADDRESS := 0;
EPCM(DS:TMP_SECS).R := 0;
EPCM(DS:TMP_SECS).W := 0;
EPCM(DS:TMP_SECS).X := 0;

```

```

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).VALID := 1;

```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If the reserved fields are not zero.</li> <li>If PAGEINFO.SECS is not zero.</li> <li>If PAGEINFO.LINADDR is not zero.</li> <li>If the SECS destination is locked.</li> <li>If SECS.SSAFRAMESIZE is insufficient.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If the SECS destination is outside the EPC.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory address is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If the reserved fields are not zero.</li> <li>If PAGEINFO.SECS is not zero.</li> <li>If PAGEINFO.LINADDR is not zero.</li> <li>If the SECS destination is locked.</li> <li>If SECS.SSAFRAMESIZE is insufficient.</li> </ul>
--------	--

#PF(error code)    If a page fault occurs in accessing memory operands.  
                          If the SECS destination is outside the EPC.

## EDBGRD—Read From a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLS[EDBGRD]	IR	V/V	SGX1	This leaf function reads a dword/quadword from a debug enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDBGRD (In)	Return error code (Out)	Data read from a debug enclave (Out)	Address of source memory in the EPC (In)

### Description

This leaf function copies a quadword/doubleword from an EPC page belonging to a debug enclave into the RBX register. Eight bytes are read in 64-bit mode, four bytes are read in non-64-bit modes. The size of data read cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

### EDBGRD Memory Parameter Semantics

EPCQW
Read access permitted by Enclave

The error codes are:

**Table 36-17. EDBGRD Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EDBGRD successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

The instruction faults if any of the following:

### EDBGRD Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is beyond the architectural size of the TCS (SGX_TCS_LIMIT).
An operand causing any segment violation.	May page fault.
CPL > 0.	

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

**Table 36-18. Base Concurrency Restrictions of EDBGRD**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGRD	Target [DS:RCX]	Shared	#GP	

**Table 36-19. Additional Concurrency Restrictions of EDBGRD**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGRD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

**Temp Variables in EDBGRD Operational Flow**

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1))
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

TMP\_MODE64 := ((IA32\_EFER.LMA = 1) && (CS.L = 1));

IF ((TMP\_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )  
 THEN #GP(0); FI;

IF ((TMP\_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

(\* make sure no other Intel SGX instruction is accessing the same EPCM entry \*)

IF (Another instruction modifying the same EPCM entry is executing)  
 THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)  
 THEN #PF(DS:RCX); FI;

(\* make sure that DS:RCX (SOURCE) is pointing to a PT\_REG or PT\_TCS or PT\_VA or PT\_SS\_FIRST or PT\_SS\_REST \*)

IF ((EPCM(DS:RCX).PT ≠ PT\_REG) and (EPCM(DS:RCX).PT ≠ PT\_TCS) and (EPCM(DS:RCX).PT ≠ PT\_VA)  
 and (EPCM(DS:RCX).PT ≠ PT\_SS\_FIRST) and (EPCM(DS:RCX).PT ≠ PT\_SS\_REST))  
 THEN #PF(DS:RCX); FI;

(\* make sure that DS:RCX points to an accessible EPC page \*)

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )  
 THEN  
 RFLAGS.ZF := 1;

```

    RAX := SGX_PAGE_NOT_DEBUGGABLE;
    GOTO DONE;
FI;

(* If source is a TCS, then make sure that the offset into the page is not beyond the TCS size*)
IF ( ( EPCM(DS:RCX).PT = PT_TCS) and ((DS:RCX) & FFFH ≥ SGX_TCS_LIMIT) )
    THEN #GP(0); FI;

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF ( (EPCM(DS:RCX).PT = PT_REG) or (EPCM(DS:RCX).PT = PT_TCS) )
    THEN
        TMP_SECS := GET_SECS_ADDRESS;
        IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
            THEN #GP(0); FI;
        IF ( (TMP_MODE64 = 1) )
            THEN RBX[63:0] := (DS:RCX)[63:0];
            ELSE EBX[31:0] := (DS:RCX)[31:0];
        FI;
    ELSE
        TMP_64BIT_VAL[63:0] := (DS:RCX)[63:0] & (~07H); // Read contents from VA slot
        IF (TMP_MODE64 = 1)
            THEN
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN RBX[63:0] := 0FFFFFFFFFFFFFFFH;
                    ELSE RBX[63:0] := 0H;
                FI;
            ELSE
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN EBX[31:0] := 0FFFFFFFFFH;
                    ELSE EBX[31:0] := 0H;
                FI;
            FI;
    FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0;

```

### Flags Affected

ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

### Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights. If DS segment is unusable. If RCX points to a memory location not 4Byte-aligned. If the address in RCX points to a page belonging to a non-debug enclave. If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA. If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
--------	---

#PF(error code)    If a page fault occurs in accessing memory operands.  
                          If the address in RCX points to a non-EPC page.  
                          If the address in RCX points to an invalid EPC page.

**64-Bit Mode Exceptions**

#GP(0)                If RCX is non-canonical form.  
                          If RCX points to a memory location not 8Byte-aligned.  
                          If the address in RCX points to a page belonging to a non-debug enclave.  
                          If the address in RCX points to a page which is not PT\_TCS, PT\_REG or PT\_VA.  
                          If the address in RCX points to a location inside TCS that is beyond SGX\_TCS\_LIMIT.

#PF(error code)    If a page fault occurs in accessing memory operands.  
                          If the address in RCX points to a non-EPC page.  
                          If the address in RCX points to an invalid EPC page.

## EDBGWR—Write to a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLS[EDBGWR]	IR	V/V	SGX1	This leaf function writes a dword/quadword to a debug enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDBGWR (In)	Return error code (Out)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

### Description

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden. The effective address of the target location inside the EPC is provided in the register RCX.

### EDBGWR Memory Parameter Semantics

EPCQW
Write access permitted by Enclave

The instruction faults if any of the following:

### EDBGWR Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is not the FLAGS word.
An operand causing any segment violation.	May page fault.
CPL > 0.	

The error codes are:

**Table 36-20. EDBGWR Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EDBGWR successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGWR does not result in a #GP.

Concurrency Restrictions

**Table 36-21. Base Concurrency Restrictions of EDBGWR**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGWR	Target [DS:RCX]	Shared	#GP	

**Table 36-22. Additional Concurrency Restrictions of EDBGWR**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGWR	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

**Temp Variables in EDBGWR Operational Flow**

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

TMP\_MODE64 := ((IA32\_EFER.LMA = 1) && (CS.L = 1));

IF ((TMP\_MODE64 = 1) and (DS:RCX is not 8Byte Aligned))  
 THEN #GP(0); FI;

IF ((TMP\_MODE64 = 0) and (DS:RCX is not 4Byte Aligned))  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

(\* make sure no other Intel SGX instruction is accessing the same EPCM entry \*)

IF (Another instruction modifying the same EPCM entry is executing)  
 THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)  
 THEN #PF(DS:RCX); FI;

(\* make sure that DS:RCX (DST) is pointing to a PT\_REG or PT\_TCS or PT\_SS\_FIRST or PT\_SS\_REST \*)

IF ((EPCM(DS:RCX).PT ≠ PT\_REG) and (EPCM(DS:RCX).PT ≠ PT\_TCS)  
 and (EPCM(DS:RCX).PT ≠ PT\_SS\_FIRST) and (EPCM(DS:RCX).PT ≠ PT\_SS\_REST))  
 THEN #PF(DS:RCX); FI;

(\* make sure that DS:RCX points to an accessible EPC page \*)

IF ((EPCM(DS:RCX).PENDING is not 0) or (EPCM(DS:RCX).MODIFIED is not 0))  
 THEN  
 RFLAGS.ZF := 1;



```

    RAX := SGX_PAGE_NOT_DEBUGGABLE;
    GOTO DONE;
FI;

(* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field*)
IF ( ( EPCM(DS:RCX).PT = PT_TCS) and ((DS:RCX) & FF8H ≠ offset_of_FLAGS & OFF8H) )
    THEN #GP(0); FI;

(* Locate the SECS for the enclave to which the DS:RCX page belongs *)
TMP_SECS := GET_SECS_PHYS_ADDRESS(EPCM(DS:RCX).ENCLAVESECS);

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 1) )
    THEN (DS:RCX)[63:0] := RBX[63:0];
    ELSE (DS:RCX)[31:0] := EBX[31:0];
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0

```

### Flags Affected

ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If the address in RCS violates DS limit or access rights.</li> <li>If DS segment is unusable.</li> <li>If RCX points to a memory location not 4Byte-aligned.</li> <li>If the address in RCX points to a page belonging to a non-debug enclave.</li> <li>If the address in RCX points to a page which is not PT_TCS or PT_REG.</li> <li>If the address in RCX points to a location inside TCS that is not the FLAGS word.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If the address in RCX points to a non-EPC page.</li> <li>If the address in RCX points to an invalid EPC page.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If RCX is non-canonical form.</li> <li>If RCX points to a memory location not 8Byte-aligned.</li> <li>If the address in RCX points to a page belonging to a non-debug enclave.</li> <li>If the address in RCX points to a page which is not PT_TCS or PT_REG.</li> <li>If the address in RCX points to a location inside TCS that is not the FLAGS word.</li> </ul>
--------	--

#PF(error code)    If a page fault occurs in accessing memory operands.  
                          If the address in RCX points to a non-EPC page.  
                          If the address in RCX points to an invalid EPC page.

## EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLS[EEXTEND]	IR	V/V	SGX1	This leaf function measures 256 bytes of an uninitialized enclave page.

### Instruction Operand Encoding

Op/En	EAX	EBX	RCX
IR	EEXTEND (In)	Effective address of the SECS of the data chunk (In)	Effective address of a 256-byte chunk in the EPC (In)

### Description

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EXTEND string comprising of “EEXTEND” || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

### EEXTEND Memory Parameter Semantics

EPC[RCX]
Read access by Enclave

The instruction faults if any of the following:

### EEXTEND Faulting Conditions

RBX points to an address not 4KBytes aligned.	RBX does not resolve to an SECS.
RBX does not point to an SECS page.	RBX does not point to the SECS page of the data chunk.
RCX points to an address not 256B aligned.	RCX points to an unused page or a SECS.
RCX does not resolve in an EPC page.	If SECS is locked.
If the SECS is already initialized.	May page fault.
CPL > 0.	

### Concurrency Restrictions

**Table 36-23. Base Concurrency Restrictions of EEXTEND**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXTEND	Target [DS:RCX]	Shared	#GP	
	SECS [DS:RBX]	Concurrent		

**Table 36-24. Additional Concurrency Restrictions of EEXTEND**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXTEND	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	

**Operation**

**Temp Variables in EEXTEND Operational Flow**

Name	Type	Size (Bits)	Description
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.
TMP_ENCLAVEOFFS ET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

TMP\_MODE64 := ((IA32\_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RBX does resolve to an EPC page)  
 THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

(\* make sure no other Intel SGX instruction is accessing EPCM \*)  
 IF (Other instructions accessing EPCM)  
 THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)  
 THEN #PF(DS:RCX); FI;

(\* make sure that DS:RCX (DST) is pointing to a PT\_REG or PT\_TCS or PT\_SS\_FIRST or PT\_SS\_REST \*)  
 IF ( (EPCM(DS:RCX).PT ≠ PT\_REG) and (EPCM(DS:RCX).PT ≠ PT\_TCS)  
 and (EPCM(DS:RCX).PT ≠ PT\_SS\_FIRST) and (EPCM(DS:RCX).PT ≠ PT\_SS\_REST))  
 THEN #PF(DS:RCX); FI;

TMP\_SECS := Get\_SECS\_ADDRESS();

IF (DS:RBX does not resolve to TMP\_SECS)  
 THEN #GP(0); FI;

(\* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT \*)  
 IF ( (Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS))

```
THEN #GP(0); FI;
```

```
(* Calculate enclave offset *)
```

```
TMP_ENCLAVEOFFSET := EPCM(DS:RCX).ENCLAVEADDRESS - TMP_SECS.BASEADDR;
```

```
TMP_ENCLAVEOFFSET := TMP_ENCLAVEOFFSET + (DS:RCX & 0FFFH)
```

```
(* Add EEXTEND message and offset to MRENCLAVE *)
```

```
TMPUPDATEFIELD[63:0] := 00444E4554584545H; // "EEXTEND"
```

```
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
```

```
TMPUPDATEFIELD[511:128] := 0; // 48 bytes
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
```

```
INC enclave's MRENCLAVE update counter;
```

```
(*Add 256 bytes to MRENCLAVE, 64 byte at a time *)
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[511:0] );
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1023: 512] );
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1535: 1024] );
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[2047: 1536] );
```

```
INC enclave's MRENCLAVE update counter by 4;
```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If the address in RBX is outside the DS segment limit.</li> <li>If RBX points to an SECS page which is not the SECS of the data chunk.</li> <li>If the address in RCX is outside the DS segment limit.</li> <li>If RCX points to a memory location not 256Byte-aligned.</li> <li>If another instruction is accessing MRENCLAVE.</li> <li>If another instruction is checking or updating the SECS.</li> <li>If the enclave is already initialized.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If the address in RBX points to a non-EPC page.</li> <li>If the address in RCX points to a page which is not PT_TCS or PT_REG.</li> <li>If the address in RCX points to a non-EPC page.</li> <li>If the address in RCX points to an invalid EPC page.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If RBX is non-canonical form.</li> <li>If RBX points to an SECS page which is not the SECS of the data chunk.</li> <li>If RCX is non-canonical form.</li> <li>If RCX points to a memory location not 256 Byte-aligned.</li> <li>If another instruction is accessing MRENCLAVE.</li> <li>If another instruction is checking or updating the SECS.</li> <li>If the enclave is already initialized.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If the address in RBX points to a non-EPC page.</li> <li>If the address in RCX points to a page which is not PT_TCS or PT_REG.</li> <li>If the address in RCX points to a non-EPC page.</li> <li>If the address in RCX points to an invalid EPC page.</li> </ul>

## EINIT—Initialize an Enclave for Execution

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLS[EINIT]	IR	V/V	SGX1	This leaf function initializes the enclave and makes it ready to execute enclave code.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EINIT (In)	Error code (Out)	Address of SIGSTRUCT (In)	Address of SECS (In)	Address of EINITTOKEN (In)

### Description

This leaf function is the final instruction executed in the enclave build process. After EINIT, the MRENCLAVE measurement is complete, and the enclave is ready to start user code execution using the EENTER instruction.

EINIT takes the effective address of a SIGSTRUCT and EINITTOKEN. The SIGSTRUCT describes the enclave including MRENCLAVE, ATTRIBUTES, ISVSVN, a 3072 bit RSA key, and a signature using the included key. SIGSTRUCT must be populated with two values, q1 and q2. These are calculated using the formulas shown below:

$$q1 = \text{floor}(\text{Signature}^2 / \text{Modulus});$$

$$q2 = \text{floor}((\text{Signature}^3 - q1 * \text{Signature} * \text{Modulus}) / \text{Modulus});$$

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

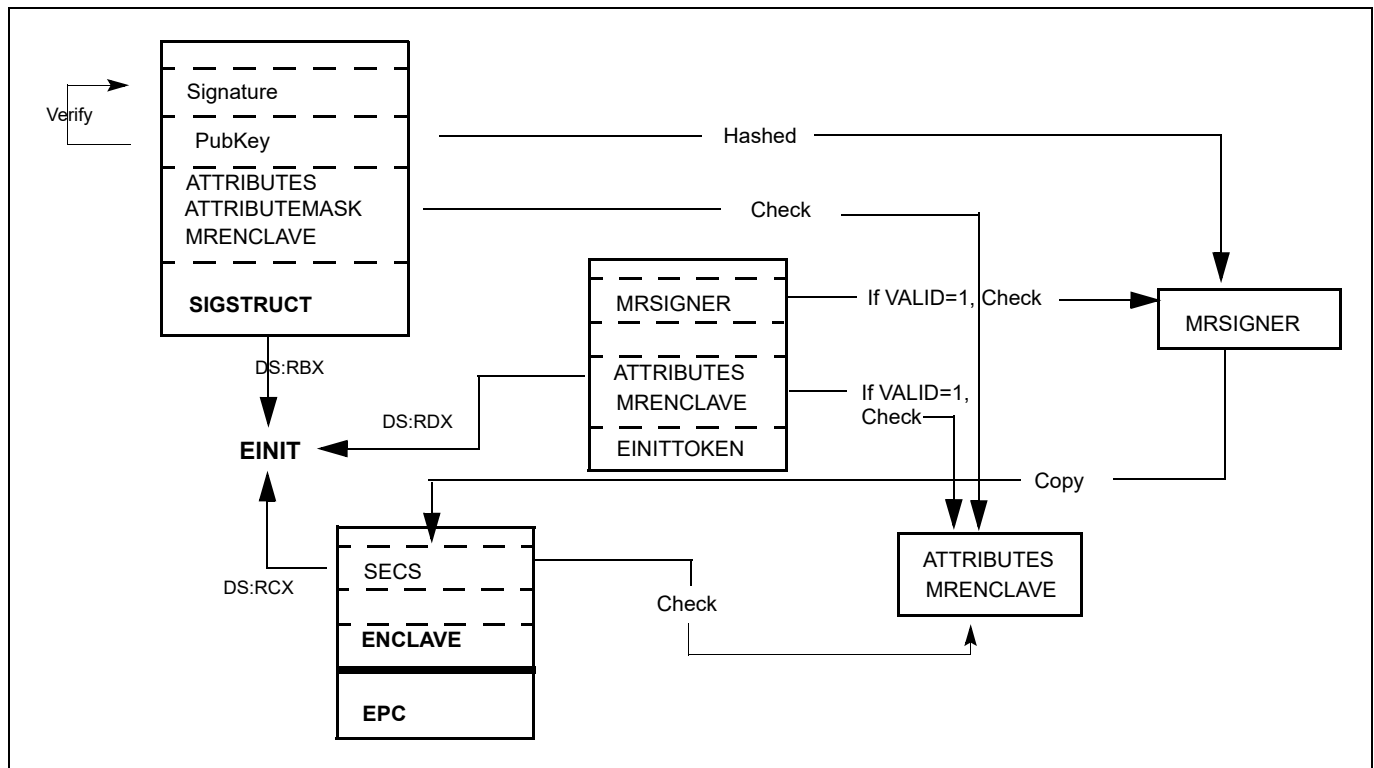


Figure 36-1. Relationships Between SECS, SIGSTRUCT and EINITTOKEN

**EINIT Memory Parameter Semantics**

SIGSTRUCT	SECS	EINITOKEN
Access by non-Enclave	Read/Write access by Enclave	Access by non-Enclave

EINIT performs the following steps, which can be seen in Figure 36-1:

Validates that SIGSTRUCT is signed using the enclosed public key.

Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.

Checks that no reserved bits are set to 1 in SIGSTRUCT.ATTRIBUTES and no reserved bits in SIGSTRUCT.ATTRIBUTESMASK are set to 0.

Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32\_SGX\_LEPUBKEYHASH.

Checks that SIGSTRUCT.ATTRIBUTES equals the result of logically and-ing SIGSTRUCT.ATTRIBUTESMASK with SECS.ATTRIBUTES.

If EINITOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32\_SGX\_LEPUBKEYHASH.

If EINITOKEN.VALID is 1, checks the validity of EINITOKEN.

If EINITOKEN.VALID is 1, checks that EINITOKEN.MRENCLAVE equals SECS.MRENCLAVE.

If EINITOKEN.VALID is 1 and EINITOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.

Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.

Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX\_UNMASKED\_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

**Table 36-25. EINIT Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EINIT successful.
SGX_INVALID_SIG_STRUCT	If SIGSTRUCT contained an invalid value.
SGX_INVALID_ATTRIBUTE	If SIGSTRUCT contains an unauthorized attributes mask.
SGX_INVALID_MEASUREMENT	If SIGSTRUCT contains an incorrect measurement. If EINITOKEN contains an incorrect measurement.
SGX_INVALID_SIGNATURE	If signature does not validate with enclosed public key.
SGX_INVALID_LICENSE	If license is invalid.
SGX_INVALID_CPUSVN	If license SVN is unsupported.
SGX_UNMASKED_EVENT	If an unmasked event is received before the instruction completes its operation.

Concurrency Restrictions

Table 36-26. Base Concurrency Restrictions of EINIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINIT	SECS [DS:RCX]	Shared	#GP	

Table 36-27. Additional Concurrency Restrictions of ENIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINIT	SECS [DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EINIT Operational Flow

Name	Type	Size	Description
TMP_SIG	SIGSTRUCT	1808Bytes	Temp space for SIGSTRUCT.
TMP_TOKEN	EINITTOKEN	304Bytes	Temp space for EINITTOKEN.
TMP_MRENCLAVE		32Bytes	Temp space for calculating MRENCLAVE.
TMP_MRSIGNER		32Bytes	Temp space for calculating MRSIGNER.
CONTROLLED_ATTRIBUTES	ATTRIBUTES	16Bytes	Constant mask of all ATTRIBUTE bits that can only be set for authorized enclaves.
TMP_KEYDEPENDENCIES	Buffer	224Bytes	Temp space for key derivation.
TMP_EINITTOKENKEY		16Bytes	Temp space for the derived EINITTOKEN Key.
TMP_SIG_PADDING	PKCS Padding Buffer	352Bytes	The value of the top 352 bytes from the computation of Signature <sup>3</sup> modulo MRSIGNER.

(\* make sure SIGSTRUCT and SECS are aligned \*)  
 IF ( ( DS:RBX is not 4KByte Aligned ) or ( DS:RCX is not 4KByte Aligned ) )  
 THEN #GP(0); FI;

(\* make sure the EINITTOKEN is aligned \*)  
 IF ( DS:RDX is not 512Byte Aligned )  
 THEN #GP(0); FI;

(\* make sure the SECS is inside the EPC \*)  
 IF ( DS:RCX does not resolve within an EPC )  
 THEN #PF(DS:RCX); FI;

TMP\_SIG[14463:0] := DS:RBX[14463:0]; // 1808 bytes  
 TMP\_TOKEN[2423:0] := DS:RDX[2423:0]; // 304 bytes



```

(* Verify SIGSTRUCT Header. *)
IF ( (TMP_SIG.HEADER ≠ 06000000E100000000001000000000h) or
    ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h) ) or
    (TMP_SIG.HEADER2 ≠ 0101000060000000600000001000000h) or
    (TMP_SIG.EXPONENT ≠ 00000003h) or (Reserved space is not 0's) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* Open "Event Window" Check for Interrupts. Verify signature using embedded public key, q1, and q2. Save upper 352 bytes of the
PKCS1.5 encoded message into the TMP_SIG_PADDING*)
IF (interrupt was pending) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_UNMASKED_EVENT;
    GOTO EXIT;
FI

IF (signature failed to verify) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_SIGNATURE;
    GOTO EXIT;
FI;

(*Close "Event Window" *)

(* make sure no other Intel SGX instruction is modifying SECS*)
IF (Other instructions modifying SECS)
    THEN #GP(0); FI;

IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT ≠ PT_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify ISVFAMILYID is not used on an enclave with KSS disabled *)
IF ((TMP_SIG.ISVFAMILYID != 0) AND (DS:RCX.ATTRIBUTES.KSS == 0))
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS's Initialized state))
    THEN #GP(0); FI;

(* Calculate finalized version of MRENCLAVE *)
(* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest *)
TMP_ENCLAVE := SHA256FINAL( (DS:RCX).MRENCLAVE, enclave's MRENCLAVE update count *512);

(* Verify MRENCLAVE from SIGSTRUCT *)
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

```

```
TMP_MRSIGNER := SHA256(TMP_SIG.MODULUS)
```

```
(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
```

```
CONTROLLED_ATTRIBUTES := 000000000000020H;
```

```
IF ( ( DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES ) ≠ 0 ) and ( TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH ) )
```

```
    RFLAGS.ZF := 1;
```

```
    RAX := SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
```

```
IF ( DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK ) ≠ ( TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK ) )
```

```
    RFLAGS.ZF := 1;
```

```
    RAX := SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(*Verify SIGSTRUCT.MISCSELECT requirements are met *)
```

```
IF ( DS:RCX.MISCSELECT & TMP_SIG.MISCMASK ) ≠ ( TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK ) )
```

```
    THEN
```

```
        RFLAGS.ZF := 1;
```

```
        RAX := SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT
```

```
FI;
```

```
IF ( CPUID.(EAX=12H, ECX=1):EAX[6] = 1 )
```

```
    IF ( DS:RCX.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIBUTES_MASK ≠ TMP_SIG.CET_ATTRIBUTES &
```

```
        TMP_SIG.CET_ATTRIBUTES_MASK )
```

```
        THEN
```

```
            RFLAGS.ZF := 1;
```

```
            RAX := SGX_INVALID_ATTRIBUTE;
```

```
            GOTO EXIT
```

```
    FI;
```

```
FI;
```

```
(* If EINITTOKEN.VALID[0] is 0, verify the enclave is signed by an authorized key *)
```

```
IF ( TMP_TOKEN.VALID[0] = 0 )
```

```
    IF ( TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH )
```

```
        RFLAGS.ZF := 1;
```

```
        RAX := SGX_INVALID_EINITTOKEN;
```

```
        GOTO EXIT;
```

```
    FI;
```

```
    GOTO COMMIT;
```

```
FI;
```

```
(* Debug Launch Enclave cannot launch Production Enclaves *)
```

```
IF ( DS:RDX.MASKEDATTRIBUTESLE.DEBUG = 1 ) and ( DS:RCX.ATTRIBUTES.DEBUG = 0 ) )
```

```
    RFLAGS.ZF := 1;
```

```
    RAX := SGX_INVALID_EINITTOKEN;
```

```
    GOTO EXIT;
```

```
FI;
```

(\* Check reserve space in EINIT token includes reserved regions and upper bits in valid field \*)

IF (TMP\_TOKEN.reserved\_space\_is\_not\_clear)

```
RFLAGS.ZF := 1;
RAX := SGX_INVALID_EINITTOKEN;
GOTO EXIT;
```

FI;

(\* EINIT token must not have been created by a configuration beyond the current CPU configuration \*)

IF (TMP\_TOKEN.CPUSVN must not be a configuration beyond CR\_CPUSVN)

```
RFLAGS.ZF := 1;
RAX := SGX_INVALID_CPUSVN;
GOTO EXIT;
```

FI;

(\* Derive Launch key used to calculate EINITTOKEN.MAC \*)

```
HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;
HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH
HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403650148866009060D30313000H;
```

```
TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_TOKEN.ISVPRODIDLE;
TMP_KEYDEPENDENCIES.ISVSVN := TMP_TOKEN.ISVSVNLE;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_TOKEN.MASKEDATTRIBUTESLE;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := IA32_SGXLEPUBKEYHASH;
TMP_KEYDEPENDENCIES.KEYID := TMP_TOKEN.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := TMP_TOKEN.CPUSVNLE;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_TOKEN.MASKEDMISCSELECTLE;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_TOKEN.CET_MASKED_ATTRIBUTES_LE;
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
```

FI;

(\* Calculate the derived key\*)

```
TMP_EINITTOKENKEY := derivekey(TMP_KEYDEPENDENCIES);
```

(\* Verify EINITTOKEN was generated using this CPU's Launch key and that it has not been modified since issuing by the Launch Enclave. Only 192 bytes of EINITTOKEN are CMACed \*)

IF (TMP\_TOKEN.MAC ≠ CMAC(TMP\_EINITTOKENKEY, TMP\_TOKEN[1535:0] ))

```
RFLAGS.ZF := 1;
RAX := SGX_INVALID_EINITTOKEN;
GOTO EXIT;
```

FI;

(\* Verify EINITOKEN (RDX) is for this enclave \*)

IF ( (TMP\_TOKEN.MRENCLAVE ≠ TMP\_MRENCLAVE) or (TMP\_TOKEN.MRSIGNER ≠ TMP\_MRSIGNER) )

RFLAGS.ZF := 1;

RAX := SGX\_INVALID\_MEASUREMENT;

GOTO EXIT;

FI;

(\* Verify ATTRIBUTES in EINITOKEN are the same as the enclave's \*)

IF (TMP\_TOKEN.ATTRIBUTES ≠ DS:RCX.ATTRIBUTES)

RFLAGS.ZF := 1;

RAX := SGX\_INVALID\_EINIT\_ATTRIBUTE;

GOTO EXIT;

FI;

COMMIT:

(\* Commit changes to the SECS; Set ISVPRODID, ISVSVN, MRSIGNER, INIT ATTRIBUTE fields in SECS (RCX) \*)

DS:RCX.MRENCLAVE := TMP\_MRENCLAVE;

(\* MRSIGNER stores a SHA256 in little endian implemented natively on x86 \*)

DS:RCX.MRSIGNER := TMP\_MRSIGNER;

DS:RCX.ISVEXTPRODID := TMP\_SIG.ISVEXTPRODID;

DS:RCX.ISVPRODID := TMP\_SIG.ISVPRODID;

DS:RCX.ISVSVN := TMP\_SIG.ISVSVN;

DS:RCX.ISVFAMILYID := TMP\_SIG.ISVFAMILYID;

DS:RCX.PADDING := TMP\_SIG.PADDING;

(\* Mark the SECS as initialized \*)

Update DS:RCX to initialized;

(\* Set RAX and ZF for success\*)

RFLAGS.ZF := 0;

RAX := 0;

EXIT:

RFLAGS.CF,PF,AF,OF,SF := 0;

### Flags Affected

ZF is cleared if successful, otherwise ZF is set and RAX contains the error code. CF, PF, AF, OF, SF are cleared.

### Protected Mode Exceptions

#GP(0)	If a memory operand is not properly aligned. If another instruction is modifying the SECS. If the enclave is already initialized. If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands. If RCX does not resolve in an EPC page. If the memory address is not a valid, uninitialized SECS.

### 64-Bit Mode Exceptions

#GP(0)	If a memory operand is not properly aligned. If another instruction is modifying the SECS. If the enclave is already initialized. If the SECS.MRENCLAVE is in use.
--------	---

#PF(error code) If a page fault occurs in accessing memory operands.  
If RCX does not resolve in an EPC page.  
If the memory address is not a valid, uninitialized SECS.

### ELDB/ELDU/ELDBC/ELDUC—Load an EPC Page and Mark its State

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLS[ELDB]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as blocked.
EAX = 08H ENCLS[ELDU]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as unblocked.
EAX = 12H ENCLS[ELDBC]	IR	V/V	EAX[6]	This leaf function behaves like ELDB but with improved conflict handling for oversubscription.
EAX = 13H ENCLS[ELDUC]	IR	V/V	EAX[6]	This leaf function behaves like ELDU but with improved conflict handling for oversubscription.

#### Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	ELDB/ELDU (In)	Return error code (Out)	Address of the PAGEINFO (In)	Address of the EPC page (In)	Address of the version- array slot (In)

#### Description

This leaf function copies a page from regular main memory to the EPC. As part of the copying process, the page is cryptographically authenticated and decrypted. This instruction can only be executed when current privilege level is 0.

The ELDB leaf function sets the BLOCK bit in the EPCM entry for the destination page in the EPC after copying. The ELDU leaf function clears the BLOCK bit in the EPCM entry for the destination page in the EPC after copying.

RBX contains the effective address of a PAGEINFO structure; RCX contains the effective address of the destination EPC page; RDX holds the effective address of the version array slot that holds the version of the page.

The ELDBC/ELDUC leafs are very similar to ELDB and ELDU. They provide an error code on the concurrency conflict for any of the pages which need to acquire a lock. These include the destination, SECS, and VA slot.

The table below provides additional information on the memory parameter of ELDB/ELDU leaf functions.

#### ELDB/ELDU/ELDBC/ELBUC Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	PAGEINFO.SECS	EPCPAGE	Version-Array Slot
Non-enclave read access	Non-enclave read access	Non-enclave read access	Enclave read/write access	Read/Write access permitted by Enclave	Read/Write access permitted by Enclave

The error codes are:

**Table 36-28. ELDB/ELDU/ELDBC/ELBUC Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	ELDB/ELDU successful.
SGX_MAC_COMPARE_FAIL	If the MAC check fails.

## Concurrency Restrictions

Table 36-29. Base Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ELDB/ELDU	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	
ELDBC/ELBUC	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA [DS:RDX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	SGX_EPC_PAGE_CONFLICT	

Table 36-30. Additional Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ELDB/ELDU	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	
ELDBC/ELBUC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	

## Operation

Temp Variables in ELDB/ELDU/ELDBC/ELBUC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Memory page	4KBytes	
TMP_SECS	Memory page	4KBytes	
TMP_PCMD	PCMD	128 Bytes	
TMP_HEADER	MACHEADER	128 Bytes	
TMP_VER	UINT64	64	
TMP_MAC	UINT128	128	
TMP_PK	UINT128	128	Page encryption/MAC key.
SCRATCH_PCMD	PCMD	128 Bytes	

(\* Check PAGEINFO and EPCPAGE alignment \*)

IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )  
THEN #GP(0); FI;

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;
```

```
(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
    THEN #GP(0); FI;
```

```
IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;
```

```
TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_PCMD := DS:RBX.PCMD;
```

```
(* Check alignment of PAGEINFO (RBX) linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
    THEN #GP(0); FI;
```

```
(* Check concurrency of EPC by other Intel SGX instructions *)
IF (other instructions accessing EPC)
    THEN
        IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
            THEN
                IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason := SGX_CONFLICT;
                        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                        VMCS.Exit_qualification.error := 0;
                        VMCS.Guest-physical_address :=
                            << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address := DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        #GP(0);
                FI;
            ELSE (* ELDBC/ELDUC *)
                IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason := SGX_CONFLICT;
                        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_ERROR;
                        VMCS.Exit_qualification.error := SGX_EPC_PAGE_CONFLICT;
                        VMCS.Guest-physical_address :=
                            << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address := DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        RFLAGS.ZF := 1;
                        RFLAGS.CF := 0;
                        RAX := SGX_EPC_PAGE_CONFLICT;
                        GOTO ERROR_EXIT;
                FI;
```



```

    FI;
FI;

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF (Other instructions modifying VA slot)
    THEN
        IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
            #GP(0);
            FI;
        ELSE (* ELDBC/ELDUC *)
            RFLAGS.ZF := 1;
            RFLAGS.CF := 0;
            RAX := SGX_EPC_PAGE_CONFLICT;
            GOTO ERROR_EXIT;
FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~OFFFH).VALID = 0) or (EPCM(DS:RDX & ~OFFFH).PT ≠ PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0] := DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0] := 0;

TMP_HEADER.SECINFO := SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD := SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR := DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) )
    THEN
        IF ( DS:TMP_SECS is not 4KByte aligned)
            THEN #GP(0) FI;
        IF (DS:TMP_SECS does not resolve within an EPC)
            THEN #PF(DS:TMP_SECS) FI;
        IF ( Other instructions modifying SECS)
            THEN
                IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
                    #GP(0);
                    FI;
                ELSE (* ELDBC/ELDUC *)
                    RFLAGS.ZF := 1;
                    RFLAGS.CF := 0;
                    RAX := SGX_EPC_PAGE_CONFLICT;
                    GOTO ERROR_EXIT;
FI;

```

```

FI;

IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
    THEN
        TMP_HEADER.EID := DS:TMP_SECS.EID;
    ELSE
        (* These pages do not have any parent, and hence no EID binding *)
        TMP_HEADER.EID := 0;
FI;

(* Copy 4KBytes SRCPGE to secure location *)
DS:RCX[32767: 0] := DS:TMP_SRCPGE[32767: 0];
TMP_VER := DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, {plain text, MAC} *)
(* Parameters for AES_GCM_DEC {Key, Counter, ..} *)
{DS:RCX, TMP_MAC} := AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_MAC_COMPARE_FAIL;
        GOTO ERROR_EXIT;
FI;

(* Check version before committing *)
IF (DS:RDX ≠ 0)
    THEN #GP(0);
    ELSE
        DS:RDX := TMP_VER;
FI;

(* Commit EPCM changes *)
EPCM(DS:RCX).PT := TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX := TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING := TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED := TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR := TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_HEADER.LINADDR;

IF ( ((EAX = 07H) or (EAX = 12H)) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA))
    THEN
        EPCM(DS:RCX).BLOCKED := 1;
    ELSE
        EPCM(DS:RCX).BLOCKED := 0;
FI;

IF (TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS)
    << store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
FI;

```

EPCM(DS:RCX). VALID := 1;

RAX := 0;  
RFLAGS.ZF := 0;

ERROR\_EXIT:  
RFLAGS.CF,PF,AF,OF,SF := 0;

### Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If the instruction's EPC resource is in use by others.</li> <li>If the instruction fails to verify MAC.</li> <li>If the version-array slot is in use.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand expected to be in EPC does not resolve to an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> <li>If the destination EPC page is already valid.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If the instruction's EPC resource is in use by others.</li> <li>If the instruction fails to verify MAC.</li> <li>If the version-array slot is in use.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand expected to be in EPC does not resolve to an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> <li>If the destination EPC page is already valid.</li> </ul>

## EMODPR—Restrict the Permissions of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0EH ENCLS[EMODPR]	IR	V/V	SGX2	This leaf function restricts the access rights associated with a EPC page in an initialized enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODPR (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

### EMODPR Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

### EMODPR Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

**Table 36-31. EMODPR Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EMODPR successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

### Concurrency Restrictions

**Table 36-32. Base Concurrency Restrictions of EMODPR**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPR	Target [DS:RCX]	Shared	#GP	

Table 36-33. Additional Concurrency Restrictions of EMODPR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPR	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

## Operation

### Temp Variables in EMODPR Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

SCRATCH\_SECINFO := DS:RBX;

(\* Check for misconfigured SECINFO flags\*)

IF ( (SCRATCH\_SECINFO reserved fields are not zero ) or  
(SCRATCH\_SECINFO.FLAGS.R is 0 and SCRATCH\_SECINFO.FLAGS.W is not 0) )  
THEN #GP(0); FI;

(\* Check concurrency with SGX1 or SGX2 instructions on the EPC page \*)  
IF (SGX1 or other SGX2 instructions accessing EPC page)  
THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID is 0 )  
THEN #PF(DS:RCX); FI;

(\* Check the EPC page for concurrency \*)

IF (EPC page in use by another SGX2 instruction)  
THEN  
RFLAGS.ZF := 1;  
RAX := SGX\_EPC\_PAGE\_CONFLICT;  
GOTO DONE;

FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )  
THEN  
RFLAGS.ZF := 1;  
RAX := SGX\_PAGE\_NOT\_MODIFIABLE;

```

    GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR := 1;

(* Update EPCM permissions *)
EPCM(DS:RCX).R := EPCM(DS:RCX).R & SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := EPCM(DS:RCX).W & SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := EPCM(DS:RCX).X & SCRATCH_SECINFO.FLAGS.X;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

```

### Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

### 64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

## EMODT—Change the Type of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0FH ENCLS[EMODT]	IR	V/V	SGX2	This leaf function changes the type of an existing EPC page.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

### EMODT Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

### EMODT Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

**Table 36-34. EMODT Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EMODT successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB.

### Concurrency Restrictions

**Table 36-35. Base Concurrency Restrictions of EMODT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR

**Table 36-36. Additional Concurrency Restrictions of EMODT**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

**Operation**

**Temp Variables in EMODT Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

SCRATCH\_SECINFO := DS:RBX;

(\* Check for misconfigured SECINFO flags\*)

IF ( (SCRATCH\_SECINFO reserved fields are not zero ) or  
 !(SCRATCH\_SECINFO.FLAGS.PT is PT\_TCS or SCRATCH\_SECINFO.FLAGS.PT is PT\_TRIM) )  
 THEN #GP(0); FI;

(\* Check concurrency with SGX1 instructions on the EPC page \*)

IF (other SGX1 instructions accessing EPC page)  
 THEN  
     RFLAGS.ZF := 1;  
     RAX := SGX\_EPC\_PAGE\_CONFLICT;  
     GOTO DONE;

FI;

IF (EPCM(DS:RCX).VALID is 0)  
 THEN #PF(DS:RCX); FI;

(\* Check the EPC page for concurrency \*)

IF (EPC page in use by another SGX2 instruction)  
 THEN  
     RFLAGS.ZF := 1;  
     RAX := SGX\_EPC\_PAGE\_CONFLICT;  
     GOTO DONE;



```

FI;

IF (!(EPCM(DS:RCX).PT is PT_REG or
    ((EPCM(DS:RCX).PT is PT_TCS or PT_SS_FIRST or PT_SS_REST) and SCRATCH_SECINFO.FLAGS.PT is PT_TRIM)))
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Update EPCM fields *)
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).MODIFIED := 1;
EPCM(DS:RCX).R := 0;
EPCM(DS:RCX).W := 0;
EPCM(DS:RCX).X := 0;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

```

### Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

### 64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

## EPA—Add Version Array

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0AH ENCLS[EPA]	IR	V/V	SGX1	This leaf function adds a Version Array to the EPC.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EPA (In)	PT_VA (In, Constant)	Effective address of the EPC page (In)

### Description

This leaf function creates an empty version array in the EPC page whose logical address is given by DS:RCX, and sets up EPCM attributes for that page. At the time of execution of this instruction, the register RBX must be set to PT\_VA.

The table below provides additional information on the memory parameter of EPA leaf function.

### EPA Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

### Concurrency Restrictions

Table 36-37. Base Concurrency Restrictions of EPA

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EPA	VA [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 36-38. Additional Concurrency Restrictions of EPA

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EPA	VA [DS:RCX]	Concurrent	L	Concurrent		Concurrent	

### Operation

IF (RBX ≠ PT\_VA or DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

(\* Check concurrency with other Intel SGX instructions \*)

IF (Other Intel SGX instructions accessing the page)  
THEN

IF (<<VMX non-root operation>> AND <<ENABLE\_EPC\_VIRTUALIZATION\_EXTENSIONS>>)

```

THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
    VMCS.Guest-linear_address := DS:RCX;
    Deliver VMEXIT;
ELSE
    #GP(0);
FI;
FI;

```

(\* Check EPC page must be empty \*)

```

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

```

(\* Clears EPC page \*)

```

DS:RCX[32767:0] := 0;

```

```

EPCM(DS:RCX).PT := PT_VA;
EPCM(DS:RCX).ENCLAVEADDRESS := 0;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).RWX := 0;
EPCM(DS:RCX).VALID := 1;

```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If a memory operand is not properly aligned.</p> <p>If another Intel SGX instruction is accessing the EPC page.</p> <p>If RBX is not set to PT_VA.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If a memory operand is not an EPC page.</p> <p>If the EPC page is valid.</p>

### 64-Bit Mode Exceptions

#GP(0)	<p>If a memory operand is non-canonical form.</p> <p>If a memory operand is not properly aligned.</p> <p>If another Intel SGX instruction is accessing the EPC page.</p> <p>If RBX is not set to PT_VA.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If a memory operand is not an EPC page.</p> <p>If the EPC page is valid.</p>

## ERDINFO—Read Type and Status Information About an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 10H ENCLS[ERDINFO]	IR	V/V	EAX[6]	This leaf function returns type and status information about an EPC page.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	ERDINFO (In)	Return error code (Out)	Address of a RDINFO structure (In)	Address of the destination EPC page (In)

### Description

This instruction reads type and status information about an EPC page and returns it in a RDINFO structure. The STATUS field of the structure describes the status of the page and determines the validity of the remaining fields. The FLAGS field returns the EPCM permissions of the page; the page type; and the BLOCKED, PENDING, MODIFIED, and PR status of the page. For enclave pages, the ENCLAVECONTEXT field of the structure returns the value of SECS.ENCLAVECONTEXT. For non-enclave pages (e.g., VA) ENCLAVECONTEXT returns 0.

For invalid or non-EPC pages, the instruction returns an information code indicating the page's status, in addition to populating the STATUS field.

ERDINFO returns an error code if the destination EPC page is being modified by a concurrent SGX instruction.

RBX contains the effective address of a RDINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of ERDINFO leaf function.

### ERDINFO Memory Parameter Semantics

RDINFO	EPCPAGE
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

### ERDINFO Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

The error codes are:

**Table 36-39. ERDINFO Return Value in RAX**

Error Code	Value	Description
No Error	0	ERDINFO successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD		Target page is not a valid EPC page.
SGX_PG_NONEPC		Page is not an EPC page.

## Concurrency Restrictions

Table 36-40. Base Concurrency Restrictions of ERDINFO

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERDINFO	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

Table 36-41. Additional Concurrency Restrictions of ERDINFO

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

## Operation

## Temp Variables in ERDINFO Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_RDINFO	Linear Address	64	Address of the RDINFO structure.

(\* check alignment of RDINFO structure (RBX) \*)  
 IF (DS:RBX is not 32Byte Aligned) THEN  
 #GP(0); FI;

(\* check alignment of the EPCPAGE (RCX) \*)  
 IF (DS:RCX is not 4KByte Aligned) THEN  
 #GP(0); FI;

(\* check that EPCPAGE (DS:RCX) is the address of an EPC page \*)  
 IF (DS:RCX does not resolve within EPC) THEN  
 RFLAGS.CF := 1;  
 RFLAGS.ZF := 0;  
 RAX := SGX\_PG\_NONEPC;  
 goto DONE;  
 FI;

(\* Check the EPC page for concurrency \*)  
 IF (EPC page is being modified) THEN  
 RFLAGS.ZF = 1;  
 RFLAGS.CF = 0;  
 RAX = SGX\_EPC\_PAGE\_CONFLICT;  
 goto DONE;  
 FI;

(\* check page validity \*)  
 IF (EPCM(DS:RCX).VALID = 0) THEN  
 RFLAGS.CF = 1;

## INTEL® SGX INSTRUCTION REFERENCES

```
RFLAGS.ZF = 0;
RAX = SGX_PG_INVLD;
goto DONE;
FI;

(* clear the fields of the RDINFO structure *)
TMP_RDINFO := DS:RBX;
TMP_RDINFO.STATUS := 0;
TMP_RDINFO.FLAGS := 0;
TMP_RDINFO.ENCLAVECONTEXT := 0;

(* store page info in RDINFO structure *)
TMP_RDINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_RDINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_RDINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_RDINFO.FLAGS.PR := EPCM(DS:RCX).PR;
TMP_RDINFO.FLAGS.PAGE_TYPE := EPCM(DS:RCX).PAGE_TYPE;
TMP_RDINFO.FLAGS.BLOCKED := EPCM(DS:RCX).BLOCKED;

(* read SECS.ENCLAVECONTEXT for enclave child pages *)
IF ((EPCM(DS:RCX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_REST)
    ) THEN
    TMP_SECS := Address of SECS for (DS:RCX);
    TMP_RDINFO.ENCLAVECONTEXT := SECS(TMP_SECS).ENCLAVECONTEXT;
FI;

(* populate enclave information for SECS pages *)
IF (EPCM(DS:RCX).PAGE_TYPE = PT_SECS) THEN
    IF ((VMX non-root mode) and
        (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)
        ) THEN
        TMP_RDINFO.STATUS.CHILDPRESENT :=
            ((SECS(DS:RCX).CHLDCNT ≠ 0) or
             SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
    ELSE
        TMP_RDINFO.STATUS.CHILDPRESENT := (SECS(DS:RCX).CHLDCNT ≠ 0);
        TMP_RDINFO.STATUS.VIRTCHILDPRESENT :=
            (SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
        TMP_RDINFO.ENCLAVECONTEXT := SECS(DS:RCX).ENCLAVECONTEXT;
    FI;
FI;

RAX := 0;
RFLAGS.ZF := 0;
RFLAGS.CF := 0;

DONE:
(* clear flags *)
RFLAGS.PF := 0;
RFLAGS.AF := 0;
```

RFLAGS.OF := 0;  
RFLAGS.SF := ? 0;

### Flags Affected

ZF is set if ERDINFO fails due to concurrent operation with another SGX instruction; otherwise cleared.

CF is set if page is not a valid EPC page or not an EPC page; otherwise cleared.

PF, AF, OF and SF are cleared.

### Protected Mode Exceptions

- |                 |  |
|-----------------|--|
| #GP(0)          | If a memory operand effective address is outside the DS segment limit.<br>If DS segment is unusable.<br>If a memory operand is not properly aligned. |
| #PF(error code) | If a page fault occurs in accessing memory operands.   |

### 64-Bit Mode Exceptions

- |                 |   |
|-----------------|---|
| #GP(0)          | If the memory address is in a non-canonical form.<br>If a memory operand is not properly aligned. |
| #PF(error code) | If a page fault occurs in accessing memory operands.  |

## EREMOVE—Remove a page from the EPC

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLS[EREMOVE]	IR	V/V	SGX1	This leaf function removes a page from the EPC.

### Instruction Operand Encoding

Op/En	EAX		RCX
IR	EREMOVE (In)	Return error code (Out)	Effective address of the EPC page (In)

### Description

This leaf function causes an EPC page to be un-associated with its SECS and be marked as unused. This instruction leaf can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if the operand is not properly aligned or does not refer to an EPC page or the page is in use by another thread, or other threads are running in the enclave to which the page belongs. In addition the instruction fails if the operand refers to an SECS with associations.

### EREMOVE Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

The instruction faults if any of the following:

### EREMOVE Faulting Conditions

The memory operand is not properly aligned.	The memory operand does not resolve in an EPC page.
Refers to an invalid SECS.	Refers to an EPC page that is locked by another thread.
Another Intel SGX instruction is accessing the EPC page.	RCX does not contain an effective address of an EPC page.
the EPC page refers to an SECS with associations.	

The error codes are:

**Table 36-42. EREMOVE Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EREMOVE successful.
SGX_CHILD_PRESENT	If the SECS still have enclave pages loaded into EPC.
SGX_ENCLAVE_ACT	If there are still logical processors executing inside the enclave.



## Concurrency Restrictions

Table 36-43. Base Concurrency Restrictions of EREMOVE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREMOVE	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 36-44. Additional Concurrency Restrictions of EREMOVE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREMOVE	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

## Operation

## Temp Variables in EREMOVE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.

```
IF (DS:RCX is not 4KByte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve to an EPC page)
  THEN #PF(DS:RCX); FI;
```

```
TMP_SECS := Get_SECS_ADDRESS();
```

```
(* Check the EPC page for concurrency *)
```

```
IF (EPC page being referenced by another Intel SGX instruction)
```

```
  THEN
```

```
    IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
```

```
      THEN
```

```
        VMCS.Exit_reason := SGX_CONFLICT;
```

```
        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
```

```
        VMCS.Exit_qualification.error := 0;
```

```
        VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
```

```
        VMCS.Guest-linear_address := DS:RCX;
```

```
        Deliver VMEXIT;
```

```
      ELSE
```

```
        #GP(0);
```

```
    FI;
```

```
FI;
```

```
(* if DS:RCX is already unused, nothing to do*)
```

```
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT = PT_TRIM AND EPCM(DS:RCX).MODIFIED = 0))
```

```
  THEN GOTO DONE;
```

```
FI;
```

```

IF ( (EPCM(DS:RCX).PT = PT_VA) OR
    ((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
    THEN
        EPCM(DS:RCX).VALID := 0;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
    THEN
        IF (DS:RCX has an EPC page associated with it)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_CHILD_PRESENT;
                GOTO ERROR_EXIT;

            FI;
            (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
            IF (<<in VMX non-root operation>> AND
                <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
                (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_CHILD_PRESENT
                    GOTO ERROR_EXIT

            FI;
            EPCM(DS:RCX).VALID := 0;
            GOTO DONE;
FI;

IF (Other threads active using SECS)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_ENCLAVE_ACT;
        GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        EPCM(DS:RCX).VALID := 0;
        GOTO DONE;
FI;

DONE:
RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;

```

**Flags Affected**

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

**Protected Mode Exceptions**

- #GP(0)                    If a memory operand effective address is outside the DS segment limit.  
                              If a memory operand is not properly aligned.  
                              If another Intel SGX instruction is accessing the page.
- #PF(error code)        If a page fault occurs in accessing memory operands.  
                              If the memory operand is not an EPC page.

**64-Bit Mode Exceptions**

- #GP(0)                    If the memory operand is non-canonical form.  
                              If a memory operand is not properly aligned.  
                              If another Intel SGX instruction is accessing the page.
- #PF(error code)        If a page fault occurs in accessing memory operands.  
                              If the memory operand is not an EPC page.

## ETRAK—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0CH ENCLS[ETRAK]	IR	V/V	SGX1	This leaf function activates EBLOCK checks.

### Instruction Operand Encoding

Op/En	EAX		RCX
IR	ETRAK (In)	Return error code (Out)	Pointer to the SECS of the EPC page (In)

### Description

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRAK leaf function.

### ETRAK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

**Table 36-45. ETRAK Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	ETRAK successful.
SGX_PREV_TRK_INCMPL	All processors did not complete the previous shoot-down sequence.

### Concurrency Restrictions

**Table 36-46. Base Concurrency Restrictions of ETRAK**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRAK	SECS [DS:RCX]	Shared	#GP	

**Table 36-47. Additional Concurrency Restrictions of ETRAK**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRAK, ETRAKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRAK	SECS [DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

**Operation**

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;
```

(\* Check concurrency with other Intel SGX instructions \*)

```
IF (Other Intel SGX instructions using tracking facility on this SECS)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
                VMCS.Guest-linear_address := 0;
                Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
    FI;
```

```
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;
```

```
IF (EPCM(DS:RCX).PT ≠ PT_SECS)
    THEN #PF(DS:RCX); FI;
```

(\* All processors must have completed the previous tracking cycle\*)

```
IF ( (DS:RCX).TRACKING ≠ 0 )
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
                VMCS.Guest-linear_address := 0;
                Deliver VMEXIT;
            FI;
        RFLAGS.ZF := 1;
        RAX := SGX_PREV_TRK_INCMPL;
        GOTO DONE;
    ELSE
        RAX := 0;
        RFLAGS.ZF := 0;
    FI;
```

```
DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

**Flags Affected**

Sets ZF if SECS is in use or invalid, otherwise cleared. Clears CF, PF, AF, OF, SF.

**Protected Mode Exceptions**

- #GP(0)                If a memory operand effective address is outside the DS segment limit.  
                         If a memory operand is not properly aligned.  
                         If another thread is concurrently using the tracking facility on this SECS.
- #PF(error code)    If a page fault occurs in accessing memory operands.  
                         If a memory operand is not an EPC page.

**64-Bit Mode Exceptions**

- #GP(0)                If a memory operand is non-canonical form.  
                         If a memory operand is not properly aligned.  
                         If the specified EPC resource is in use.
- #PF(error code)    If a page fault occurs in accessing memory operands.  
                         If a memory operand is not an EPC page.

## ETRACKC—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 11H ENCLS[ETRACKC]	IR	V/V	EAX[6]	This leaf function activates EBLOCK checks.

### Instruction Operand Encoding

Op/En	EAX		RCX	
IR	ETRACK (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Address of the SECS page (In, EA)

### Description

The ETRACKC instruction is thread safe variant of ETRACK leaf and can be executed concurrently with other CPU threads operating on the same SECS.

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

### ETRACKC Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

**Table 36-48. ETRACKC Return Value in RAX**

Error Code	Value	Description
No Error	0	ETRACKC successful.
SGX_EPC_PAGE_CONFLICT	7	Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD	6	Target page is not a VALID EPC page.
SGX_PREV_TRK_INCMPL	17	All processors did not complete the previous tracking sequence.
SGX_TRACK_NOT_REQUIRED	27	Target page type does not require tracking.

### Concurrency Restrictions

**Table 36-49. Base Concurrency Restrictions of ETRACKC**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACKC	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS implicit	Concurrent		

**Table 36-50. Additional Concurrency Restrictions of ETRACKC**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACKC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

**Operation**

**Temp Variables in ETRACKC Operational Flow**

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

(\* check alignment of EPCPAGE (RCX) \*)

```
IF (DS:RCX is not 4KByte Aligned) THEN
#GP(0); FI;
```

(\* check that EPCPAGE (DS:RCX) is the address of an EPC page \*)

```
IF (DS:RCX does not resolve within an EPC) THEN
#PF(DS:RCX, PFEC.SGX); FI;
```

(\* Check the EPC page for concurrency \*)

```
IF (EPC page is being modified) THEN
RFLAGS.ZF := 1;
RFLAGS.CF := 0;
RAX := SGX_EPC_PAGE_CONFLICT;
goto DONE_POST_LOCK_RELEASE;
FI;
```

(\* check to make sure the page is valid \*)

```
IF (EPCM(DS:RCX).VALID = 0) THEN
RFLAGS.ZF := 1;
RFLAGS.CF := 0;
RAX := SGX_PG_INVLD;
GOTO DONE;
FI;
```

(\* find out the target SECS page \*)

```
IF (EPCM(DS:RCX).PT is PT_REG or PT_TCS or PT_TRIM or PT_SS_FIRST or PT_SS_REST) THEN
TMP_SECS := Obtain SECS through EPCM(DS:RCX).ENCLAVESECS;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS) THEN
TMP_SECS := Obtain SECS through (DS:RCX);
ELSE
RFLAGS.ZF := 0;
RFLAGS.CF := 1;
RAX := SGX_TRACK_NOT_REQUIRED;
GOTO DONE;
FI;
```



```

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS) THEN
  IF ((VMX non-root mode) and
    (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address :=
      SECS(TMP_SECS).ENCLAVECONTEXT;
    VMCS.Guest-linear_address := 0;
    Deliver VMEXIT;
  FI;

  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_EPC_PAGE_CONFLICT;
  GOTO DONE;
FI;

(* All processors must have completed the previous tracking cycle*)
IF ( (TMP_SECS).TRACKING ≠ 0 )
THEN
  IF ((VMX non-root mode) and
    (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address :=
      SECS(TMP_SECS).ENCLAVECONTEXT;
    VMCS.Guest-linear_address := 0;
    Deliver VMEXIT;
  FI;

  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_PREV_TRK_INCMPL;
  GOTO DONE;
FI;

RFLAGS.ZF := 0;
RFLAGS.CF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.PF,AF,OF,SF := 0;

```

### Flags Affected

ZF is set if ETRACKC fails due to concurrent operations with another SGX instructions or target page is an invalid EPC page or tracking is not completed on SECS page; otherwise cleared.

CF is set if target page is not of a type that requires tracking; otherwise cleared.

PF, AF, OF and SF are cleared.

**Protected Mode Exceptions**

- #GP(0)                If the memory operand violates access-control policies of DS segment.  
                          If DS segment is unusable.
- #PF(error code)    If the memory operand is not properly aligned.  
                          If the memory operand expected to be in EPC does not resolve to an EPC page.  
                          If a page fault occurs in access memory operand.

**64-Bit Mode Exceptions**

- #GP(0)                If a memory address is in a non-canonical form.
- #PF(error code)    If a memory operand is not properly aligned.  
                          If the memory operand expected to be in EPC does not resolve to an EPC page.  
                          If a page fault occurs in access memory operand.

## EWB—Invalidate an EPC Page and Write out to Main Memory

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0BH ENCLS[EWB]	IR	V/V	SGX1	This leaf function invalidates an EPC page and writes it out to main memory.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EWB (In)	Error code (Out)	Address of an PAGEINFO (In)	Address of the EPC page (In)	Address of a VA slot (In)

### Description

This leaf function copies a page from the EPC to regular main memory. As part of the copying process, the page is cryptographically protected. This instruction can only be executed when current privilege level is 0.

The table below provides additional information on the memory parameter of EPA leaf function.

### EWB Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	EPCPAGE	VASLOT
Non-EPC R/W access	Non-EPC R/W access	Non-EPC R/W access	EPC R/W access	EPC R/W access

The error codes are:

**Table 36-51. EWB Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EWB successful.
SGX_PAGE_NOT_BLOCKED	If page is not marked as blocked.
SGX_NOT_TRACKED	If EWB is racing with ETRACK instruction.
SGX_VA_SLOT_OCCUPIED	Version array slot contained valid entry.
SGX_CHILD_PRESENT	Child page present while attempting to page out enclave.

### Concurrency Restrictions

**Table 36-52. Base Concurrency Restrictions of EWB**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EWB	Source [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	

**Table 36-53. Additional Concurrency Restrictions of EWB**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EWB	Source [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Exclusive	

Operation

Temp Variables in EWB Operational Flow

Name	Type	Size (Bytes)	Description
TMP_SRCPAGE	Memory page	4096	
TMP_PCMD	PCMD	128	
TMP_SECS	SECS	4096	
TMP_BPEPOCH	UINT64	8	
TMP_BPREFCOUNT	UINT64	8	
TMP_HEADER	MAC Header	128	
TMP_PCMD_ENCLAVEID	UINT64	8	
TMP_VER	UINT64	8	
TMP_PK	UINT128	16	

IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

IF (DS:RDX is not 8Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)  
 THEN #PF(DS:RDX); FI;

(\* EPCPAGE and VASLOT should not resolve to the same EPC page\*)  
 IF (DS:RCX and DS:RDX resolve to the same EPC page)  
 THEN #GP(0); FI;

TMP\_SRCPAGE := DS:RBX.SRCPAGE;  
 (\* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO \*)  
 TMP\_PCMD := DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)  
 THEN #GP(0); FI;

IF ( (DS:TMP\_PCMD is not 128Byte Aligned) or (DS:TMP\_SRCPAGE is not 4KByte Aligned) )  
 THEN #GP(0); FI;

(\* Check for concurrent Intel SGX instruction access to the page \*)  
 IF (Other Intel SGX instruction is accessing page)  
 THEN  
     IF (<<VMX non-root operation>> AND <<ENABLE\_EPC\_VIRTUALIZATION\_EXTENSIONS>>)  
     THEN  
         VMCS.Exit\_reason := SGX\_CONFLICT;  
         VMCS.Exit\_qualification.code := EPC\_PAGE\_CONFLICT\_EXCEPTION;  
         VMCS.Exit\_qualification.error := 0;  
         VMCS.Guest-physical\_address := << translation of DS:RCX produced by paging >>;

```

        VMCS.Guest-linear_address := DS:RCX;
    Deliver VMEXIT;
    ELSE
        #GP(0);
    FI;
FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~OFFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        TMP_SECS = Obtain SECS through EPCM(DS:RCX)
        (* Check that EBLOCK has occurred correctly *)
        IF (EBLOCK is not correct)
            THEN #GP(0); FI;
FI;

RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;

(* Zero out TMP_HEADER*)
TMP_HEADER[ sizeof(TMP_HEADER) - 1 : 0 ] := 0;

(* Perform page-type-specific checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        (* check to see if the page is evictable *)
        IF (EPCM(DS:RCX).BLOCKED = 0)
            THEN
                RAX := SGX_PAGE NOT_BLOCKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
            FI;
        (* Check if tracking done correctly *)
        IF (Tracking not correct)
            THEN
                RAX := SGX_NOT_TRACKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
            FI;
        FI;

        (* Obtain EID to establish cryptographic binding between the paged-out page and the enclave *)

```

```

    TMP_HEADER.EID := TMP_SECS.EID;

    (* Obtain EID as an enclave handle for software *)
    TMP_PCMD_ENCLAVEID := TMP_SECS.EID;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS)
    (*check that there are no child pages inside the enclave *)
    IF (DS:RCX has an EPC page associated with it)
        THEN
            RAX := SGX_CHILD_PRESENT;
            RFLAGS.ZF := 1;
            GOTO ERROR_EXIT;
    FI;
    (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
    IF (<<in VMX non-root operation>> AND
    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
    (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
        THEN
            RFLAGS.ZF := 1;
            RAX := SGX_CHILD_PRESENT;
            GOTO ERROR_EXIT;
    FI;
    TMP_HEADER.EID := 0;
    (* Obtain EID as an enclave handle for software *)
    TMP_PCMD_ENCLAVEID := (DS:RCX).EID;
ELSE IF (EPCM(DS:RCX).PT is PT_VA)
    TMP_HEADER.EID := 0; // Zero is not a special value
    (* No enclave handle for VA pages*)
    TMP_PCMD_ENCLAVEID := 0;
FI;

TMP_HEADER.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;
TMP_HEADER.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
TMP_HEADER.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_HEADER.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_HEADER.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_HEADER.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;

(* Encrypt the page, DS:RCX could be encrypted in place. AES-GCM produces 2 values, {ciphertext, MAC}. *)
(* AES-GCM input parameters: key, GCM Counter, MAC_HDR, MAC_HDR_SIZE, SRC, SRC_SIZE*)
{DS:TMP_SRCPGE, DS:TMP_PCMD.MAC} := AES_GCM_ENC(CR_BASE_PK), (TMP_VER << 32),
    TMP_HEADER, 128, DS:RCX, 4096);

(* Write the output *)
Zero out DS:TMP_PCMD.SECINFO
DS:TMP_PCMD.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
DS:TMP_PCMD.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
DS:TMP_PCMD.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
DS:TMP_PCMD.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
DS:TMP_PCMD.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;
DS:TMP_PCMD.RESERVED := 0;
DS:TMP_PCMD.ENCLAVEID := TMP_PCMD_ENCLAVEID;
DS:RBX.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;

(*Check if version array slot was empty *)

```

```

IF ([DS.RDX])
  THEN
    RAX := SGX_VA_SLOT_OCCUPIED
    RFLAGS.CF := 1;
FI;

```

```

(* Write version to Version Array slot *)
[DS.RDX] := TMP_VER;

```

```

(* Free up EPCM Entry *)
EPCM.(DS:RCX).VALID := 0;
ERROR_EXIT:

```

### Flags Affected

ZF is set if page is not blocked, not tracked, or a child is present. Otherwise cleared.

CF is set if VA slot is previously occupied, Otherwise cleared.

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If the EPC page and VASLOT resolve to the same EPC page.</li> <li>If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.</li> <li>If the tracking resource is in use.</li> <li>If the EPC page or the version array page is invalid.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If the EPC page and VASLOT resolve to the same EPC page.</li> <li>If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.</li> <li>If the tracking resource is in use.</li> <li>If the EPC page or the version array page is invalid.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> </ul>

## 36.4 INTEL® SGX USER LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.



## EACCEPT—Accept Changes to an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLU[EACCEPT]	IR	V/V	SGX2	This leaf function accepts changes made by system software to an EPC page in the running enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EACCEPT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

### EACCEPT Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

### EACCEPT Faulting Conditions

The operands are not properly aligned.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	Page type is PT_REG and MODIFIED bit is 0.
SECINFO contains an invalid request.	Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.
If security attributes of the SECINFO page make the page inaccessible.	

The error codes are:

**Table 36-54. EACCEPT Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EACCEPT successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.
SGX_NOT_TRACKED	The OS did not complete an ETRACK on the target page.

Concurrency Restrictions

**Table 36-55. Base Concurrency Restrictions of EACCEPT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target [DS:RCX]	Shared	#GP	
	SECINFO [DS:RBX]	Concurrent		

**Table 36-56. Additional Concurrency Restrictions of EACCEPT**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

**Temp Variables in EACCEPT Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operands belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RBX is not within CR\_ELRANGE)  
 THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)  
 THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or  
 (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or  
 (EPCM(DS:RBX &~FFFH).PT ≠ PT\_REG) or (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or  
 (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)))  
 THEN #PF(DS:RBX); FI;

(\* Copy 64 bytes of contents \*)  
 SCRATCH\_SECINFO := DS:RBX;

(\* Check for misconfigured SECINFO flags\*)  
 IF (SCRATCH\_SECINFO reserved fields are not zero )  
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
 THEN #GP(0); FI;

```
IF (DS:RCX is not within CR_ELRANGE)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
  THEN #PF(DS:RCX); FI;
```

(\* Check that the combination of requested PT, PENDING and MODIFIED is legal \*)

```
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 0 )
  THEN
    IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) and
      ((SCRATCH_SECINFO.FLAGS.PR is 1) or
      (SCRATCH_SECINFO.FLAGS.PENDING is 1)) and
      (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) or
      ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or PT_TRIM) and
      (SCRATCH_SECINFO.FLAGS.PR is 0) and
      (SCRATCH_SECINFO.FLAGS.PENDING is 0) and
      (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) )))
      THEN #GP(0); FI
```

```
ELSE
```

```
  IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) AND
    ((SCRATCH_SECINFO.FLAGS.PR is 1) OR
    (SCRATCH_SECINFO.FLAGS.PENDING is 1)) AND
    (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) OR
    ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS OR PT_TRIM) AND
    (SCRATCH_SECINFO.FLAGS.PENDING is 0) AND
    (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) AND
    (SCRATCH_SECINFO.FLAGS.PR is 0)) OR
    ((SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST or PT_SS_REST) AND
    (SCRATCH_SECINFO.FLAGS.PENDING is 1) AND
    (SCRATCH_SECINFO.FLAGS.MODIFIED is 0) AND
    (SCRATCH_SECINFO.FLAGS.PR is 0))))
    THEN #GP(0); FI;
```

```
FI;
```

(\* Check security attributes of the destination EPC page \*)

```
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).BLOCKED is not 0) or
  ((EPCM(DS:RCX).PT is not PT_REG) and (EPCM(DS:RCX).PT is not PT_TCS) and (EPCM(DS:RCX).PT is not PT_TRIM)
  and (EPCM(DS:RCX).PT is not PT_SS_FIRST) and (EPCM(DS:RCX).PT is not PT_SS_REST)) or
  (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
  THEN #PF(DS:RCX); FI;
```

(\* Check the destination EPC page for concurrency \*)

```
IF ( EPC page in use )
  THEN #GP(0); FI;
```

(\* Re-Check security attributes of the destination EPC page \*)

```
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
  THEN #PF(DS:RCX); FI;
```

(\* Verify that accept request matches current EPC page settings \*)

```
IF ( (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX) or (EPCM(DS:RCX).PENDING ≠ SCRATCH_SECINFO.FLAGS.PENDING) or
  (EPCM(DS:RCX).MODIFIED ≠ SCRATCH_SECINFO.FLAGS.MODIFIED) or (EPCM(DS:RCX).R ≠ SCRATCH_SECINFO.FLAGS.R) or
  (EPCM(DS:RCX).W ≠ SCRATCH_SECINFO.FLAGS.W) or (EPCM(DS:RCX).X ≠ SCRATCH_SECINFO.FLAGS.X) or
  (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) )
```

```

THEN
    RFLAGS.ZF := 1;
    RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
    GOTO DONE;
FI;
(* Check that all required threads have left enclave *)
IF (Tracking not correct)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_NOT_TRACKED;
        GOTO DONE;
FI;

(* Get pointer to the SECS to which the EPC page belongs *)
TMP_SECS = << Obtain physical address of SECS through EPCM(DS:RCX)>>
(* For TCS pages, perform additional checks *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;

        (* Check that TCS.FLAGS.DBGOPTIN, TCS stack, and TCS status are correctly initialized *)
        (* check that TCS.PREVSSP is 0 *)
        IF ( ((DS:RCX).FLAGS.DBGOPTIN is not 0) or ((DS:RCX).CSSA ≥ (DS:RCX).NSSA) or ((DS:RCX).AEP is not 0) or ((DS:RCX).STATE is not 0)
            or ((CPUID.(EAX=12H, ECX=1):EAX[6] = 1) AND ((DS:RCX).PREVSSP != 0)))
            THEN #GP(0); FI;

        (* Check consistency of FS & GS Limit *)
        IF ( (TMP_SECS.ATTRIBUTES.MODE64BIT is 0) and ((DS:RCX.FSLIMIT & FFFH ≠ FFFH) or (DS:RCX.GSLIMIT & FFFH ≠ FFFH)) )
            THEN #GP(0); FI;
FI;

(* Clear PENDING/MODIFIED flags to mark accept operation complete *)
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;

(* Clear EAX and ZF to indicate successful completion *)
RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

```

**Flags Affected**

Sets ZF if page cannot be accepted, otherwise cleared. Clears CF, PF, AF, OF, SF

**Protected Mode Exceptions**

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If EPC page has incorrect page type or security attributes.

**64-Bit Mode Exceptions**

#GP(0)	If executed outside an enclave. If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If EPC page has incorrect page type or security attributes.

## EACCEPTCOPY—Initialize a Pending Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLU[EACCEPTCOPY]	IR	V/V	SGX2	This leaf function initializes a dynamically allocated EPC page from another page in the EPC.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EACCEPTCOPY (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)	Address of the source EPC page (In)

### Description

This leaf function copies the contents of an existing EPC page into an uninitialized EPC page (created by EAUG). After initialization, the instruction may also modify the access rights associated with the destination EPC page. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX and RDX each contain the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPTCOPY leaf function.

### EACCEPTCOPY Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)	EPCPAGE (Source)
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

### EACCEPTCOPY Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	If security attributes of the source EPC page make the page inaccessible.
The EPC page is not valid.	RBX does not contain an effective address in an EPC page in the running enclave.
SECINFO contains an invalid request.	RCX/RDX does not contain an effective address of an EPC page in the running enclave.

The error codes are:

**Table 36-57. EACCEPTCOPY Return Value in RAX**

Error Code (see Table 36-4)	Description
No Error	EACCEPTCOPY successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.

## Concurrency Restrictions

Table 36-58. Base Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPTCOPY	Target [DS:RCX]	Concurrent		
	Source [DS:RDX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Table 36-59. Additional Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPTCOPY	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

## Operation

Temp Variables in EACCEPTCOPY Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
THEN #GP(0); FI;

IF ( (DS:RCX is not 4KByte Aligned) or (DS:RDX is not 4KByte Aligned) )  
THEN #GP(0); FI;

IF ((DS:RBX is not within CR\_ELRange) or (DS:RCX is not within CR\_ELRange) or (DS:RDX is not within CR\_ELRange))  
THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)  
THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

IF (DS:RDX does not resolve within an EPC)  
THEN #PF(DS:RDX); FI;

IF ( (EPCM(DS:RBX & ~FFFH).VALID = 0) or (EPCM(DS:RBX & ~FFFH).R = 0) or (EPCM(DS:RBX & ~FFFH).PENDING ≠ 0) or  
(EPCM(DS:RBX & ~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX & ~FFFH).BLOCKED ≠ 0) or (EPCM(DS:RBX & ~FFFH).PT ≠ PT\_REG) or  
(EPCM(DS:RBX & ~FFFH).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or  
(EPCM(DS:RBX & ~FFFH).ENCLAVEADDRESS ≠ DS:RBX) )  
THEN #PF(DS:RBX); FI;

(\* Copy 64 bytes of contents \*)

```
SCRATCH_SECINFO := DS:RBX;
```

(\* Check for misconfigured SECINFO flags\*)

```
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or (SCRATCH_SECINFO.FLAGS.R=0) AND(SCRATCH_SECINFO.FLAGS.W#0 ) or
  (SCRATCH_SECINFO.FLAGS.PT is not PT_REG) )
  THEN #GP(0); FI;
```

(\* Check security attributes of the source EPC page \*)

```
IF ( (EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING # 0) or (EPCM(DS:RDX).MODIFIED # 0) or
  (EPCM(DS:RDX).BLOCKED # 0) or (EPCM(DS:RDX).PT # PT_REG) or (EPCM(DS:RDX).ENCLAVESECS # CR_ACTIVE_SECS) or
  (EPCM(DS:RDX).ENCLAVEADDRESS # DS:RDX))
  THEN #PF(DS:RDX); FI;
```

(\* Check security attributes of the destination EPC page \*)

```
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING # 1) or (EPCM(DS:RCX).MODIFIED # 0) or
  (EPCM(DS:RDX).BLOCKED # 0) or (EPCM(DS:RCX).PT # PT_REG) or (EPCM(DS:RCX).ENCLAVESECS # CR_ACTIVE_SECS) )
  THEN
```

```
  RFLAGS.ZF := 1;
```

```
  RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
```

```
  GOTO DONE;
```

```
FI;
```

(\* Check the destination EPC page for concurrency \*)

```
IF (destination EPC page in use )
```

```
  THEN #GP(0); FI;
```

(\* Re-Check security attributes of the destination EPC page \*)

```
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING # 1) or (EPCM(DS:RCX).MODIFIED # 0) or
  (EPCM(DS:RCX).R # 1) or (EPCM(DS:RCX).W # 1) or (EPCM(DS:RCX).X # 0) or
  (EPCM(DS:RCX).PT # SCRATCH_SECINFO.FLAGS.PT) or (EPCM(DS:RCX).ENCLAVESECS # CR_ACTIVE_SECS) or
  (EPCM(DS:RCX).ENCLAVEADDRESS # DS:RCX))
  THEN
```

```
  RFLAGS.ZF := 1;
```

```
  RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
```

```
  GOTO DONE;
```

```
FI;
```

(\* Copy 4KBbytes form the source to destination EPC page\*)

```
DS:RCX[32767:0] := DS:RDX[32767:0];
```

(\* Update EPCM permissions \*)

```
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
```

```
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
```

```
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
```

```
EPCM(DS:RCX).PENDING := 0;
```

```
RFLAGS.ZF := 0;
```

```
RAX := 0;
```

```
DONE:
```

```
RFLAGS.CF,PF,AF,OF,SF := 0;
```



**Flags Affected**

Sets ZF if page is not modifiable, otherwise cleared. Clears CF, PF, AF, OF, SF

**Protected Mode Exceptions**

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If EPC page has incorrect page type or security attributes.

**64-Bit Mode Exceptions**

#GP(0)	If executed outside an enclave. If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If EPC page has incorrect page type or security attributes.

## EENTER—Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLU[EENTER]	IR	V/V	SGX1	This leaf function is used to enter an enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX	
IR	EENTER (In)	Content of RBX.CSSA (Out)	Address of a TCS (In)	Address of AEP (In)	Address of IP following EENTER (Out)

### Description

The ENCLU[EENTER] instruction transfers execution to an enclave. At the end of the instruction, the logical processor is executing in enclave mode at the RIP computed as EnclaveBase + TCS.OENTRY. If the target address is not within the CS segment (32-bit) or is not canonical (64-bit), a #GP(0) results.

### EENTER Memory Parameter Semantics

TCS
Enclave access

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use.	Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

The following operations are performed by EENTER:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or interrupt.
- The AEP contained in RCX is stored into the TCS for use by AEXs. FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 38.1.2):
  - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 38.2.5).
  - On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 38.2.2).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 38.2.3):

- All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED\_CTR1 and FIXED\_CTR2.
- PEBS is suppressed.
- AnyThread counting on other threads is demoted to MyThread mode and IA32\_PERF\_GLOBAL\_STATUS[60] on that thread is set
- If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32\_PERF\_GLOBAL\_STATUS[60] and IA32\_PERF\_GLOBAL\_STATUS[63].

## Concurrency Restrictions

**Table 36-60. Base Concurrency Restrictions of EENTER**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EENTER	TCS [DS:RBX]	Shared	#GP	

**Table 36-61. Additional Concurrency Restrictions of EENTER**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EENTER	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

## Operation

### Temp Variables in EENTER Operational Flow

Name	Type	Size (Bits)	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for GS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));
```

(\* Make sure DS is usable, expand up \*)

```
IF (TMP_MODE64 = 0 and (DS not usable or ((DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))
  THEN #GP(0); FI;
```

(\* Check that CS, SS, DS, ES.base is 0 \*)

```
IF (TMP_MODE64 = 0)
  THEN
    IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
    IF(ES usable and ES.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.B = 0) #GP(0); FI;
```

```

FI;

IF (DS:RBX is not 4KByte Aligned)
  THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
  THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
  THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions is operating on TCS)
  THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
  THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
  THEN #PF(DS:RBX); FI;

IF ( ( EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX ) or ( EPCM(DS:RBX).PT ≠ PT_TCS ) )
  THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
  THEN #PF(DS:RBX); FI;

IF ( ( DS:RBX).OSSA is not 4KByte Aligned)
  THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( ( DS:RBX).OFSBASE is not 4KByte Aligned) or ( ( DS:RBX).OGSBASE is not 4KByte Aligned) )
  THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
  THEN
    TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
    TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
    (* if FS wrap-around, make sure DS has no holes*)
    IF (TMP_FSLIMIT < TMP_FSBASE)
      THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
      ELSE
        IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
    (* if GS wrap-around, make sure DS has no holes*)

```

```

    IF (TMP_GSLIMIT < TMP_GSBASE)
        THEN
            IF (DS.limit < 4GB) THEN #GP(0); FI;
        ELSE
            IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
ELSE
    TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
        THEN #GP(0); FI;
FI;

(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFF7FEH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one more frame *)
IF ( (DS:RBX).CSSA ≥ (DS:RBX).NSSA)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * (DS:RBX).CSSA;
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;

    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)

```

```

    THEN #PF(DS:TMP_SSA_PAGE); FI;
  IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
    THEN #PF(DS:TMP_SSA_PAGE); FI;
  IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
    THEN #PF(DS:TMP_SSA_PAGE); FI;
  CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

```

(\* Compute address of GPR area\*)

```
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
```

If a fault occurs; release locks, abort and deliver that fault;

```

IF (DS:TMP_GPR does not resolve to EPC page)
  THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
  THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
  THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
  THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
  (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
  (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
  THEN #PF(DS:TMP_GPR); FI;

```

```

IF (TMP_MODE64 = 0)
  THEN
    IF (TMP_GPR + (GPR_SIZE - 1) is not in DS segment) THEN #GP(0); FI;
FI;

```

```
CR_GPR_PA := Physical_Address (DS: TMP_GPR);
```

(\* Validate TCS.OENTRY \*)

```
TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
```

```

IF (TMP_MODE64 = 1)
  THEN
    IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
  ELSE
    IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

```

(\* Ensure the enclave is not already active and this thread is the only one using the TCS\*)

```
IF (DS:RBX.STATE = ACTIVE)
```

```
  THEN #GP(0); FI;
```

```
TMP_IA32_U_CET := 0
```

```
TMP_SSP := 0
```

```

IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
  THEN
    IF ( CR4.CET = 0 )
      THEN

```

(\* If part does not support CET or CET has not been enabled and enclave requires CET then fail \*)  
 IF ( TMP\_SECS.CET\_ATTRIBUTES ≠ 0 OR TMP\_SECS.CET\_LEG\_BITMAP\_OFFSET ≠ 0 ) #GP(0); FI;

FI;

(\* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail EENTER \*)

IF ( TMP\_SECS.CET\_ATTRIBUTES.SH\_STK\_EN = 1 OR TMP\_SECS.CET\_ATTRIBUTES.ENDBR\_EN = 1 )  
 THEN

IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;

FI;

IF (TMP\_SECS.CET\_ATTRIBUTES.SH\_STK\_EN OR TMP\_SECS.CET\_ATTRIBUTES.ENDBR\_EN)

THEN

(\* Setup CET state from SECS, note tracker goes to IDLE \*)

TMP\_IA32\_U\_CET = TMP\_SECS.CET\_ATTRIBUTES;

IF (TMP\_IA32\_U\_CET.LEG\_IW\_EN = 1 AND TMP\_IA32\_U\_CET.ENDBR\_EN = 1 )

THEN

TMP\_IA32\_U\_CET := TMP\_IA32\_U\_CET + TMP\_SECS.BASEADDR;

TMP\_IA32\_U\_CET := TMP\_IA32\_U\_CET + TMP\_SECS.CET\_LEG\_BITMAP\_BASE;

FI;

(\* Compute linear address of what will become new CET state save area and cache its PA \*)

TMP\_CET\_SAVE\_AREA = DS:RBX.OCETSSA + TMP\_SECS.BASEADDR + (DS:RBX.CSSA) \* 16

TMP\_CET\_SAVE\_PAGE = TMP\_CET\_SAVE\_AREA & ~0xFFF;

Check the TMP\_CET\_SAVE\_PAGE page is read/write accessible

If fault occurs release locks, abort and deliver fault

(\* Read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically \*)

IF ((DS:TMP\_CET\_SAVE\_PAGE Does NOT RESOLVE TO EPC PAGE) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).VALID = 0) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).PENDING = 1) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).MODIFIED = 1) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).BLOCKED = 1) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).R = 0) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).W = 0) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).ENCLAVEADDRESS ≠ DS:TMP\_CET\_SAVE\_PAGE) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).PT ≠ PT\_SS\_REST) OR

(EPCM(DS:TMP\_CET\_SAVE\_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))

THEN

#PF(DS:TMP\_CET\_SAVE\_PAGE);

FI;

CR\_CET\_SAVE\_AREA\_PA := Physical address(DS:TMP\_CET\_SAVE\_AREA)

IF TMP\_IA32\_U\_CET.SH\_STK\_EN = 1

THEN

TMP\_SSP = TCS.PREVSSP;

FI;

FI;

FI;

CR\_ENCLAVE\_MODE := 1;

CR\_ACTIVE\_SECS := TMP\_SECS;

CR\_ELRRANGE := (TMPSECS.BASEADDR, TMP\_SECS.SIZE);

(\* Save state for possible AEXs \*)

```
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;
```

(\* Save the hidden portions of FS and GS \*)

```
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;
```

(\* If XSAVE is enabled, save XCRO and replace it with SECS.ATTRIBUTES.XFRM\*)

```
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCRO := XCRO;
    XCRO := TMP_SECS.ATTRIBUTES.XFRM;
FI;
```

```
RCX := RIP;
RIP := TMP_TARGET;
RAX := (DS:RBX).CSSA;
(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
DS:TMP_SSA.U_RSP := RSP;
DS:TMP_SSA.U_RBP := RBP;
```

(\* Do the FS/GS swap \*)

```
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;
```

```
GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS.L;
```



```

GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
  THEN
    Suppress_all_code_breakpoints_that_overlap_with_ELRANGE;
    CR_SAVE_TF := RFLAGS.TF;
    RFLAGS.TF := 0;
    Suppress_monitor_trap_flag for the source of the execution of the enclave;
    Suppress any pending debug exceptions;
    Suppress any pending MTF VM exit;
  ELSE
    IF RFLAGS.TF = 1
      THEN pend a single-step #DB at the end of EENTER; FI;
    IF the "monitor trap flag" VM-execution control is set
      THEN pend an MTF VM exit at the end of EENTER; FI;
  FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_JBT] = 1) OR (CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1))
  THEN
    (* Save enclosing application CET state into save registers *)
    CR_SAVE_IA32_U_CET := IA32_U_CET
    (* Setup enclave CET state *)
    IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
      THEN
        CR_SAVE_SSP := SSP
      FI;

    IA32_U_CET := TMP_IA32_U_CET;

  FI;

Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;

```

### Flags Affected

RFLAGS.TF is cleared on opt-out entry

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If DS:RBX is not page aligned.</li> <li>If the enclave is not initialized.</li> <li>If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.</li> <li>If the thread is not in the INACTIVE state.</li> <li>If CS, DS, ES or SS bases are not all zero.</li> <li>If executed in enclave mode.</li> <li>If any reserved field in the TCS FLAG is set.</li> <li>If the target address is not within the CS segment.</li> </ul>
--------	--

If CR4.OSFXSR = 0.  
 If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.  
 If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.  
 #PF(error code) If a page fault occurs in accessing memory.  
 If DS:RBX does not point to a valid TCS.  
 If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT\_REG EPC page.

**64-Bit Mode Exceptions**

#GP(0) If DS:RBX is not page aligned.  
 If the enclave is not initialized.  
 If the thread is not in the INACTIVE state.  
 If CS, DS, ES or SS bases are not all zero.  
 If executed in enclave mode.  
 If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.  
 If the target address is not canonical.  
 If CR4.OSFXSR = 0.  
 If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.  
 If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.  
 #PF(error code) If a page fault occurs in accessing memory operands.  
 If DS:RBX does not point to a valid TCS.  
 If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT\_REG EPC page.

## EEXIT—Exits an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EEXIT]	IR	V/V	SGX1	This leaf function is used to exit an enclave.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EEXIT (In)	Target address outside the enclave (In)	Address of the current AEP (Out)

### Description

The ENCLU[EEXIT] instruction exits the currently executing enclave and branches to the location specified in RBX. RCX receives the current AEP. If RBX is not within the CS (32-bit mode) or is not canonical (64-bit mode) a #GP(0) results.

### EEXIT Memory Parameter Semantics

Target Address
Non-Enclave read and execute access

If RBX specifies an address that is inside the enclave, the instruction will complete normally. The fetch of the next instruction will occur in non-enclave mode, but will attempt to fetch from inside the enclave. This fetch returns a fixed data pattern.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

### Concurrency Restrictions

**Table 36-62. Base Concurrency Restrictions of EEXIT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXIT		Concurrent		

**Table 36-63. Additional Concurrency Restrictions of EEXIT**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXIT		Concurrent		Concurrent		Concurrent	

## Operation

## Temp Variables in EEXIT Operational Flow

Name	Type	Size (Bits)	Description
TMP_RIP	Effective Address	32/64	Saved copy of CRIP for use when creating LBR.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));
```

```
IF (TMP_MODE64 = 1)
  THEN
    IF (RBX is not canonical) THEN #GP(0); FI;
  ELSE
    IF (RBX > CS limit) THEN #GP(0); FI;
FI;
```

```
TMP_RIP := CRIP;
RIP := RBX;
```

```
(* Return current AEP in RCX *)
RCX := CR_TCS_PA.AEP;
```

```
(* Do the FS/GS swap *)
FS.selector := CR_SAVE_FS.selector;
FS.base := CR_SAVE_FS.base;
FS.limit := CR_SAVE_FS.limit;
FS.access_rights := CR_SAVE_FS.access_rights;
GS.selector := CR_SAVE_GS.selector;
GS.base := CR_SAVE_GS.base;
GS.limit := CR_SAVE_GS.limit;
GS.access_rights := CR_SAVE_GS.access_rights;
```

```
(* Restore XCRO if needed *)
IF (CR4.OSXSAVE = 1)
  XCRO := CR_SAVE__XCRO;
FI;
```

```
Unsuppress_all_code_breakpoints_that_are_outside_ELRange;
```

```
IF (CR_DBGOPTIN = 0)
  THEN
    UnSuppress_all_code_breakpoints_that_overlap_with_ELRange;
    Restore suppressed breakpoint matches;
    RFLAGS.TF := CR_SAVE_TF;
    UnSuppress_monitor_trap_flag;
    UnSuppress_LBR_Generation;
    UnSuppress_performance_monitoring_activity;
    Restore performance monitoring counter AnyThread demotion to MyThread in enclave back to AnyThread
FI;
```

```
IF RFLAGS.TF = 1
  THEN Pend Single-Step #DB at the end of EEXIT;
FI;
```

```

IF the "monitor trap flag" VM-execution control is set
  THEN pend a MTF VM exit at the end of EEXIT;
FI;

IF (CPUID.(EAX=12H, ECX=1);EAX[6] = 1)
  THEN
    (* Record PREVSSP *)
    IF (IA32_U_CET.SH_STK_EN == 1)
      THEN CR_TCS_PA.PREVSSP = SSP; FI;
  FI;

IF ((CPUID.(EAX=7H, ECX=0);EDX[CET_IBT] = 1) OR (CPUID.(EAX=7, ECX=0);ECX[CET_SS] = 1)
  THEN
    (* Restore enclosing app's CET state from the save registers *)
    IA32_U_CET := CR_SAVE_IA32_U_CET;
    IF CPUID.(EAX=07H, ECX=00h);ECX[CET_SS] = 1
      THEN SSP := CR_SAVE_SSP; FI;

    (* Update enclosing app's TRACKER if enclosing app has indirect branch tracking enabled *)
    IF (CR4.CET = 1 AND IA32_U_CET.ENDBR_EN = 1)
      THEN
        IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
        IA32_U_CET.SUPPRESS := 0;
      FI;
  FI;

CR_ENCLAVE_MODE := 0;
CR_TCS_PA.STATE := INACTIVE;

```

```

(* Assure consistent translations *)
Flush_linear_context;

```

### Flags Affected

RFLAGS.TF is restored from the value previously saved in EENTER or ERESUME.

### Protected Mode Exceptions

#GP(0)	If executed outside an enclave. If RBX is outside the CS segment.
#PF(error code)	If a page fault occurs in accessing memory.

### 64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave. If RBX is not canonical.
#PF(error code)	If a page fault occurs in accessing memory operands.

## EGETKEY—Retrieves a Cryptographic Key

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLU[EGETKEY]	IR	V/V	SGX1	This leaf function retrieves a cryptographic key.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EGETKEY (In)	Return error code (Out)	Address to a KEYREQUEST (In)	Address of the OUTPUTDATA (In)

### Description

The ENCLU[EGETKEY] instruction returns a 128-bit secret key from the processor specific key hierarchy. The register RBX contains the effective address of a KEYREQUEST structure, which the instruction interprets to determine the key being requested. The Requesting Keys section below provides a description of the keys that can be requested. The RCX register contains the effective address where the key will be returned. Both the addresses in RBX & RCX should be locations inside the enclave.

EGETKEY derives keys using a processor unique value to create a specific key based on a number of possible inputs. This instruction leaf can only be executed inside an enclave.

### EGETKEY Memory Parameter Semantics

KEYREQUEST	OUTPUTDATA
Enclave read access	Enclave write access

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on the Table 36-64.
- Computes derived key using the derivation data and package specific value.
- Outputs the calculated key to the address in RCX.

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.{ZF, CF, AF, OF, SF, PF}. The instruction returns an error code if the user tries to request a key based on an invalid CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX\_INVALID\_SVN, SGX\_INVALID\_ATTRIBUTE, SGX\_INVALID\_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

### Requesting Keys

The KEYREQUEST structure (see Section 33.18.1) identifies the key to be provided. The Keyrequest.KeyName field identifies which type of key is requested.

### Deriving Keys

Key derivation is based on a combination of the enclave specific values (see Table 36-64) and a processor key. Depending on the key being requested a field may either be included by definition or the value may be included from the KeyRequest. A “yes” in Table 36-64 indicates the value for the field is included from its default location, identified in the source row, and a “request” indicates the values for the field is included from its corresponding KeyRequest field.

Table 36-64. Key Derivation

	Key Name	Attributes	Owner Epoch	CPU SVN	ISV SVN	ISV PRODIG	ISVEXT PRODIG	ISVFAM ILYID	MRENCLAVE	MRSIGNER	CONFIG ID	CONFIGS VN	RAND
Source	Key Dependent Constant	Y := SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;	CR_SGX_OWNER EPOCH	Y := CPUSVN Register;	R := Req.ISV SVN;	SECS.ISVID	SECS.ISVEXTPR ODID	SECS.ISVFAMIL YID	SECS.MRENCLAVE	SECS.MRSIGNER	SECS.CONFIGID	SECS.CONFIGSVN	Req. KEYID
		R := AttribMask & SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;		R := Req.CPU SVN;									
EINITTOKEN	Yes	Request	Yes	Request	Request	Yes	No	No	No	Yes	No	No	Request
Report	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	Request
Seal	Yes	Request	Yes	Request	Request	Request	Request	Request	Request	Request	Request	Request	Request
Provisioning	Yes	Request	No	Request	Request	Yes	No	No	No	Yes	No	No	Yes
Provisioning Seal	Yes	Request	No	Request	Request	Request	Request	Request	No	Yes	Request	Request	Yes

Keys that permit the specification of a CPU or ISV's code's, or enclave configuration's SVNs have additional requirements. The caller may not request a key for an SVN beyond the current CPU, ISV or enclave configuration's SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN\_KEY be set and SECS.MRSIGNER equal IA32\_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcoded PKCS padding constant (352 byte string):

```
HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;
```

```
HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH
```

```
HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403650148866009060D30313000H;
```

The error codes are:

Table 36-65. EGETKEY Return Value in RAX

Error Code (see Table 36-4)	Value	Description
No Error	0	EGETKEY successful.
SGX_INVALID_ATTRIBUTE		The KEYREQUEST contains a KEYNAME for which the enclave is not authorized.
SGX_INVALID_CPUSVN		If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value.
SGX_INVALID_ISVSVN		If KEYREQUEST software SVN (ISVSVN or CONFIGSVN) is greater than the enclave's corresponding SVN.
SGX_INVALID_KEYNAME		If KEYREQUEST.KEYNAME is an unsupported value.

## Concurrency Restrictions

Table 36-66. Base Concurrency Restrictions of EGETKEY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		
	OUTPUTDATA [DS:RCX]	Concurrent		

**Table 36-67. Additional Concurrency Restrictions of EGETKEY**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EGETKEY Operational Flow**

Name	Type	Size (Bits)	Description
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_ATTRIBUTES		128	Temp Space for the calculation of the sealable Attributes.
TMP_ISVEXTPRODID		16 bytes	Temp Space for ISVEXTPRODID.
TMP_ISVPRODID		2 bytes	Temp Space for ISVPRODID.
TMP_ISVFAMILYID		16 bytes	Temp Space for ISVFAMILYID.
TMP_CONFIGID		64 bytes	Temp Space for CONFIGID.
TMP_CONFIGSVN		2 bytes	Temp Space for CONFIGSVN.
TMP_OUTPUTKEY		128	Temp Space for the calculation of the key.

(\* Make sure KEYREQUEST is properly aligned and inside the current enclave \*)

IF ( (DS:RBX is not 512Byte aligned) or (DS:RBX is within CR\_ELRANGE) )  
 THEN #GP(0); FI;

(\* Make sure DS:RBX is an EPC address and the EPC page is valid \*)

IF ( (DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0) )  
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)

THEN #PF(DS:RBX); FI;

(\* Check page parameters for correctness \*)

IF ( (EPCM(DS:RBX).PT ≠ PT\_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or (EPCM(DS:RBX).PENDING = 1) or  
 (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )  
 THEN #PF(DS:RBX);

FI;

(\* Make sure OUTPUTDATA is properly aligned and inside the current enclave \*)

IF ( (DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR\_ELRANGE) )  
 THEN #GP(0); FI;

(\* Make sure DS:RCX is an EPC address and the EPC page is valid \*)

IF ( (DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0) )



```

THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
  THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
  (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~OFFFH) ) or (EPCM(DS:RCX).W = 0) )
  THEN #PF(DS:RCX);
FI;

(* Verify RESERVED spaces in KEYREQUEST are valid *)
IF ( (DS:RBX).RESERVED ≠ 0) or (DS:RBX.KEYPOLICY.RESERVED ≠ 0) )
  THEN #GP(0); FI;

TMP_CURRENTSECS := CR_ACTIVE_SECS;

(* Verify that CONFIGSVN & New Policy bits are not used if KSS is not enabled *)
IF ((TMP_CURRENTSECS.ATTRIBUTES.KSS == 0) AND ((DS:RBX.KEYPOLICY & 0x003C ≠ 0) OR (DS:RBX.CONFIGSVN > 0)))
  THEN #GP(0); FI;

(* Determine which enclave attributes that must be included in the key. Attributes that must always be include INIT & DEBUG *)
REQUIRED_SEALING_MASK[127:0] := 00000000 00000000 00000000 00000003H;
TMP_ATTRIBUTES := (DS:RBX.ATTRIBUTEMASK | REQUIRED_SEALING_MASK) & TMP_CURRENTSECS.ATTRIBUTES;

(* Compute MISCSELECT fields to be included *)
TMP_MISCSELECT := DS:RBX.MISCMASK & TMP_CURRENTSECS.MISCSELECT

(* Compute CET_ATTRIBUTES fields to be included *)
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
  THEN TMP_CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES_MASK & TMP_CURRENTSECS.CET_ATTRIBUTES; FI;
TMP_KEYDEPENDENCIES := 0;

CASE (DS:RBX.KEYNAME)
  SEAL_KEY:
    IF (DS:RBX.CPUSVN is beyond current CPU configuration)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
    FI;
    IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
    FI;
    IF (DS:RBX.CONFIGSVN > TMP_CURRENTSECS.CONFIGSVN)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
    FI;

  (*Include enclave identity?*)

```

```

TMP_MRENCLAVE := 0;
IF (DS:RBX.KEYPOLICY.MRENCLAVE = 1)
    THEN TMP_MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
FI;
(*Include enclave author?*)
TMP_MRSIGNER := 0;
IF (DS:RBX.KEYPOLICY.MRSIGNER = 1)
    THEN TMP_MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
FI;
(* Include enclave product family ID? *)
TMP_ISVFAMILYID := 0;
IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
    THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
FI;

(* Include enclave product ID? *)
TMP_ISVPRODID := 0;
IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
    THEN TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
FI;

(* Include enclave Config ID? *)
TMP_CONFIGID := 0;
TMP_CONFIGSVN := 0;
IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
    THEN TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;
    TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
FI;

(* Include enclave extended product ID? *)
TMP_ISVEXTPRODID := 0;
IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
    THEN TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
FI;

//Determine values key is based on
TMP_KEYDEPENDENCIES.KEYNAME := SEAL_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTESMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_MRENCLAVE;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;

```

```

TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := DS:RBX.CET_ATTRIBUTES_MASK;
FI;
BREAK;
REPORT_KEY:
//Determine values key is based on
TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := 0;
TMP_KEYDEPENDENCIES.ISVSVN := 0;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
TMP_KEYDEPENDENCIES.MRSIGNER := 0;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
EINITTOKEN_KEY:
(* Check ENCLAVE has EINITTOKEN Key capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.EINITTOKEN_KEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
FI;

```

```

(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
PROVISION_KEY:
(* Check ENCLAVE has PROVISIONING capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
FI;
(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;

```

```

TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTESMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := 0;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := 0;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
PROVISION_SEAL_KEY:
(* Check ENCLAVE has PROVISIONING capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
FI;
(* Include enclave product family ID? *)
TMP_ISVFAMILYID := 0;
IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
    THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
FI;

(* Include enclave product ID? *)
TMP_ISVPRODID := 0;
IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
    THEN TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
FI;

(* Include enclave Config ID? *)
TMP_CONFIGID := 0;
TMP_CONFIGSVN := 0;
IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
    THEN TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;

```

```

TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
FI;

(* Include enclave extended product ID? *)
TMP_ISVEXTPRODID := 0;
IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
    TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
FI;

(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_SEAL_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTESMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := 0;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
    FI;
BREAK;
DEFAULT:
    (* The value of KEYNAME is invalid *)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_KEYNAME;
    GOTO EXIT;
ESAC;

(* Calculate the final derived key and output to the address in RCX *)
TMP_OUTPUTKEY := derivekey(TMP_KEYDEPENDENCIES);
DS:RCX[15:0] := TMP_OUTPUTKEY;
RAX := 0;
RFLAGS.ZF := 0;

EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

```

**Flags Affected**

ZF is cleared if successful, otherwise ZF is set. CF, PF, AF, OF, SF are cleared.

**Protected Mode Exceptions**

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the current enclave. If an effective address is not properly aligned. If an effective address is outside the DS segment limit. If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory.

**64-Bit Mode Exceptions**

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the current enclave. If an effective address is not properly aligned. If an effective address is not canonical. If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory operands.

## EMODPE—Extend an EPC Page Permissions

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLU[EMODPE]	IR	V/V	SGX2	This leaf function extends the access rights of an existing EPC page.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EMODPE (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function extends the access rights associated with an existing EPC page in the running enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not extend the page permissions will have no effect. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPE leaf function.

### EMODPE Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

### EMODPE Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is not valid.	RCX does not contain an effective address of an EPC page in the running enclave.
SECINFO contains an invalid request.	

### Concurrency Restrictions

**Table 36-68. Base Concurrency Restrictions of EMODPE**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPE	Target [DS:RCX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

**Table 36-69. Additional Concurrency Restrictions of EMODPE**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPE	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	



## Operation

## Temp Variables in EMODPE Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF ((DS:RBX is not within CR\_ELRANGE) or (DS:RCX is not within CR\_ELRANGE) )  
THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)  
THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RBX).VALID = 0) or (EPCM(DS:RBX).R = 0) or (EPCM(DS:RBX).PENDING ≠ 0) or (EPCM(DS:RBX).MODIFIED ≠ 0) or  
(EPCM(DS:RBX).BLOCKED ≠ 0) or (EPCM(DS:RBX).PT ≠ PT\_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or  
(EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0xFFF)) )  
THEN #PF(DS:RBX); FI;

SCRATCH\_SECINFO := DS:RBX;

(\* Check for misconfigured SECINFO flags\*)  
IF (SCRATCH\_SECINFO reserved fields are not zero )  
THEN #GP(0); FI;

(\* Check security attributes of the EPC page \*)  
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or  
(EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT\_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) )  
THEN #PF(DS:RCX); FI;

(\* Check the EPC page for concurrency \*)  
IF (EPC page in use by another SGX2 instruction)  
THEN #GP(0); FI;

(\* Re-Check security attributes of the EPC page \*)  
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or  
(EPCM(DS:RCX).PT ≠ PT\_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or  
(EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))  
THEN #PF(DS:RCX); FI;

(\* Check for misconfigured SECINFO flags\*)  
IF ( (EPCM(DS:RCX).R = 0) and (SCRATCH\_SECINFO.FLAGS.R = 0) and (SCRATCH\_SECINFO.FLAGS.W ≠ 0) )  
THEN #GP(0); FI;

(\* Update EPCM permissions \*)

EPCM(DS:RCX).R := EPCM(DS:RCX).R | SCRATCH\_SECINFO.FLAGS.R;  
EPCM(DS:RCX).W := EPCM(DS:RCX).W | SCRATCH\_SECINFO.FLAGS.W;  
EPCM(DS:RCX).X := EPCM(DS:RCX).X | SCRATCH\_SECINFO.FLAGS.X;

### Flags Affected

None

### Protected Mode Exceptions

- #GP(0) If executed outside an enclave.  
If a memory operand effective address is outside the DS segment limit.  
If a memory operand is not properly aligned.  
If a memory operand is locked.
- #PF(error code) If a page fault occurs in accessing memory operands.

### 64-Bit Mode Exceptions

- #GP(0) If executed outside an enclave.  
If a memory operand is non-canonical form.  
If a memory operand is not properly aligned.  
If a memory operand is locked.
- #PF(error code) If a page fault occurs in accessing memory operands.

## EReport—Create a Cryptographic Report of the Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLU[EReport]	IR	V/V	SGX1	This leaf function creates a cryptographic report of the enclave.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX	RDX
IR	EReport (In)	Address of TARGETINFO (In)	Address of REPORTDATA (In)	Address where the REPORT is written to in an OUTPUTDATA (In)

### Description

This leaf function creates a cryptographic REPORT that describes the contents of the enclave. This instruction leaf can only be executed when inside the enclave. The cryptographic report can be used by other enclaves to determine that the enclave is running on the same platform.

RBX contains the effective address of the MRENCLAVE value of the enclave that will authenticate the REPORT output, using the REPORT key delivered by EGETKEY command for that enclave. RCX contains the effective address of a 64-byte REPORTDATA structure, which allows the caller of the instruction to associate data with the enclave from which the instruction is called. RDX contains the address where the REPORT will be output by the instruction.

### EReport Memory Parameter Semantics

TARGETINFO	REPORTDATA	OUTPUTDATA
Read access by Enclave	Read access by Enclave	Read/Write access by Enclave

This instruction leaf perform the following:

1. Validate the 3 operands (RBX, RCX, RDX) are inside the enclave.
2. Compute a report key for the target enclave, as indicated by the value located in RBX(TARGETINFO).
3. Assemble the enclave SECS data to complete the REPORT structure (including the data provided using the RCX (REPORTDATA) operand).
4. Computes a cryptographic hash over REPORT structure.
5. Add the computed hash to the REPORT structure.
6. Output the completed REPORT structure to the address in RDX (OUTPUTDATA).

The instruction fails if the operands are not properly aligned.

CR\_REPORT\_KEYID, used to provide key wearout protection, is populated with a statistically unique value on boot of the platform by a trusted entity within the SGX TCB.

The instruction faults if any of the following:

### EReport Faulting Conditions

An effective address not properly aligned.	An memory address does not resolve in an EPC page.
If accessing an invalid EPC page.	If the EPC page is blocked.
May page fault.	

Concurrency Restrictions

**Table 36-70. Base Concurrency Restrictions of EREPORT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREPORT	TARGETINFO [DS:RBX]	Concurrent		
	REPORTDATA [DS:RCX]	Concurrent		
	OUTPUTDATA [DS:RDX]	Concurrent		

**Table 36-71. Additional Concurrency Restrictions of EREPORT**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREPORT	TARGETINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RDX]	Concurrent		Concurrent		Concurrent	

Operation

**Temp Variables in EREPORT Operational Flow**

Name	Type	Size (bits)	Description
TMP_ATTRIBUTES		32	Physical address of SECS of the enclave to which source operand belongs.
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_REPORTKEY		128	REPORTKEY generated by the instruction.
TMP_REPORT		3712	

TMP\_MODE64 := ((IA32\_EFER.LMA = 1) && (CS.L = 1));

(\* Address verification for TARGETINFO (RBX) \*)

IF ( (DS:RBX is not 512Byte Aligned) or (DS:RBX is not within CR\_ELRange) )  
 THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)  
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).VALID = 0)  
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)  
 THEN #PF(DS:RBX); FI;

(\* Check page parameters for correctness \*)

IF ( (EPCM(DS:RBX).PT ≠ PT\_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or (EPCM(DS:RBX).PENDING = 1) or  
 (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~OFFFH) ) or (EPCM(DS:RBX).R = 0) )

```

THEN #PF(DS:RBX);
FI;

```

```

(* Verify RESERVED spaces in TARGETINFO are valid *)

```

```

IF (DS:RBX.RESERVED != 0)
  THEN #GP(0); FI;

```

```

(* Address verification for REPORTDATA (RCX) *)

```

```

IF ( (DS:RCX is not 128Byte Aligned) or (DS:RCX is not within CR_ELRANGE) )
  THEN #GP(0); FI;

```

```

IF (DS:RCX does not resolve within an EPC)

```

```

  THEN #PF(DS:RCX); FI;

```

```

IF (EPCM(DS:RCX).VALID = 0)

```

```

  THEN #PF(DS:RCX); FI;

```

```

IF (EPCM(DS:RCX).BLOCKED = 1)

```

```

  THEN #PF(DS:RCX); FI;

```

```

(* Check page parameters for correctness *)

```

```

IF ( (EPCM(DS:RCX).PT != PT_REG) or (EPCM(DS:RCX).ENCLAVESECS != CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
      (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS != (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).R = 0) )
  THEN #PF(DS:RCX);

```

```

FI;

```

```

(* Address verification for OUTPUTDATA (RDX) *)

```

```

IF ( (DS:RDX is not 512Byte Aligned) or (DS:RDX is not within CR_ELRANGE) )
  THEN #GP(0); FI;

```

```

IF (DS:RDX does not resolve within an EPC)

```

```

  THEN #PF(DS:RDX); FI;

```

```

IF (EPCM(DS:RDX).VALID = 0)

```

```

  THEN #PF(DS:RDX); FI;

```

```

IF (EPCM(DS:RDX).BLOCKED = 1)

```

```

  THEN #PF(DS:RDX); FI;

```

```

(* Check page parameters for correctness *)

```

```

IF ( (EPCM(DS:RDX).PT != PT_REG) or (EPCM(DS:RDX).ENCLAVESECS != CR_ACTIVE_SECS) or (EPCM(DS:RDX).PENDING = 1) or
      (EPCM(DS:RDX).MODIFIED = 1) or (EPCM(DS:RDX).ENCLAVEADDRESS != (DS:RDX & ~0FFFH) ) or (EPCM(DS:RDX).W = 0) )
  THEN #PF(DS:RDX);

```

```

FI;

```

```

(* REPORT MAC needs to be computed over data which cannot be modified *)

```

```

TMP_REPORT.CPUSVN := CR_CPUSVN;

```

```

TMP_REPORT.ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;

```

```

TMP_REPORT.ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;

```

```

TMP_REPORT.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;

```

```

TMP_REPORT.ISVSVN := TMP_CURRENTSECS.ISVSVN;

```

```

TMP_REPORT.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;

```

```

TMP_REPORT.REPORTDATA := DS:RCX[511:0];

```

```

TMP_REPORT.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;

```

```

TMP_REPORT.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_REPORT.MRRESERVED := 0;
TMP_REPORT.KEYID[255:0] := CR_REPORT_KEYID;
TMP_REPORT.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
TMP_REPORT.CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_REPORT.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN TMP_REPORT.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES; FI;

```

(\* Derive the report key \*)

```

TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := 0;
TMP_KEYDEPENDENCIES.ISVSVN := 0;
TMP_KEYDEPENDENCIES.SGXOWNERPOUCH := CR_SGXOWNERPOUCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := DS:RBX.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := DS:RBX.MEASUREMENT;
TMP_KEYDEPENDENCIES.MRSIGNER := 0;
TMP_KEYDEPENDENCIES.KEYID := TMP_REPORT.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := DS:RBX.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := DS:RBX.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := DS:RBX.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;

```

(\* Calculate the derived key\*)

```

TMP_REPORTKEY := derivekey(TMP_KEYDEPENDENCIES);

```

(\* call cryptographic CMAC function, CMAC data are not including MAC&KEYID \*)

```

TMP_REPORT.MAC := cmac(TMP_REPORTKEY, TMP_REPORT[3071:0]);
DS:RDX[3455:0] := TMP_REPORT;

```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	If executed outside an enclave. If the address in RCS is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

**64-Bit Mode Exceptions**

#GP(0)	If executed outside an enclave. If RCX is non-canonical form. If a memory operand is not properly aligned. If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

## ERESUME—Re-Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLU[ERESUME]	IR	V/V	SGX1	This leaf function is used to re-enter an enclave after an interrupt.

### Instruction Operand Encoding

Op/En	RAX	RBX	RCX
IR	ERESUME (In)	Address of a TCS (In)	Address of AEP (In)

### Description

The ENCLU[ERESUME] instruction resumes execution of an enclave that was interrupted due to an exception or interrupt, using the machine state previously stored in the SSA.

### ERESUME Memory Parameter Semantics

TCS
Enclave read/write access

The instruction faults if any of the following:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use by another enclave.	Either of TCS-specified FS and GS segment is not a subset of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
Offsets 520-535 of the XSAVE area not 0.	The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.
The SSA frame is not valid or in use.	

The following operations are performed by ERESUME:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or an asynchronous exit due to any Interrupt event.
- The AEP contained in RCX is stored into the TCS for use by AEXs. FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 38.1.2):
  - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 38.2.5).
  - On opt-in entry, a single-step debug exception is pending on the instruction boundary immediately after EENTER (see Section 38.2.3).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.



- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 38.2.3):
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED\_CTR1 and FIXED\_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32\_PERF\_GLOBAL\_STATUS[60] on that thread is set.
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32\_PERF\_GLOBAL\_STATUS[60] and IA32\_PERF\_GLOBAL\_STATUS[63].

## Concurrency Restrictions

**Table 36-72. Base Concurrency Restrictions of ERESUME**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERESUME	TCS [DS:RBX]	Shared	#GP	

**Table 36-73. Additional Concurrency Restrictions of ERESUME**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERESUME	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

## Operation

### Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for GS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));
```

(\* Make sure DS is usable, expand up \*)

```
IF (TMP_MODE64 = 0 and (DS not usable or ((DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))
  THEN #GP(0); FI;
```

(\* Check that CS, SS, DS, ES.base is 0 \*)

```
IF (TMP_MODE64 = 0)
  THEN
    IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
    IF(ES.usable and ES.base ≠ 0) #GP(0); FI;
    IF(SS.usable and SS.base ≠ 0) #GP(0); FI;
    IF(SS.usable and SS.B = 0) #GP(0); FI;
FI;
```

```
IF (DS:RBX is not 4KByte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RBX does not resolve within an EPC)
  THEN #PF(DS:RBX); FI;
```

(\* Check AEP is canonical\*)

```
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
  THEN #GP(0); FI;
```

(\* Check concurrency of TCS operation\*)

```
IF (Other Intel SGX instructions is operating on TCS)
  THEN #GP(0); FI;
```

(\* TCS verification \*)

```
IF (EPCM(DS:RBX).VALID = 0)
  THEN #PF(DS:RBX); FI;
```

```
IF (EPCM(DS:RBX).BLOCKED = 1)
  THEN #PF(DS:RBX); FI;
```

```
IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
  THEN #PF(DS:RBX); FI;
```

```
IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
  THEN #PF(DS:RBX); FI;
```

```
IF ( (DS:RBX).OSSA is not 4KByte Aligned)
  THEN #GP(0); FI;
```

(\* Check proposed FS and GS \*)

```
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
  THEN #GP(0); FI;
```

(\* Get the SECS for the enclave in which the TCS resides \*)

```
TMP_SECS := Address of SECS for TCS;
```

(\* Make sure that the FLAGS field in the TCS does not have any reserved bits set \*)

```
IF ( ( (DS:RBX).FLAGS & FFFFFFFF00000000H) ≠ 0)
  THEN #GP(0); FI;
```

(\* SECS must exist and enclave must have previously been EINITted \*)

```
IF (the enclave is not already initialized)
  THEN #GP(0); FI;
```

(\* make sure the logical processor's operating mode matches the enclave \*)

```
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
  THEN #GP(0); FI;
```

```
IF (CR4.OSFXSR = 0)
  THEN #GP(0); FI;
```

(\* Check for legal values of SECS.ATTRIBUTES.XFRM \*)

```
IF (CR4.OSXSAVE = 0)
  THEN
    IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
  ELSE
    IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCRO) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
  FI;
```

(\* Make sure the SSA contains at least one active frame \*)

```
IF ( (DS:RBX).CSSA = 0)
  THEN #GP(0); FI;
```

(\* Compute linear address of SSA frame \*)

```
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * ( (DS:RBX).CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);
```

```
FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
```

```
  (* Check page is read/write accessible *)
```

```
  Check that DS:TMP_SSA_PAGE is read/write accessible;
```

```
  If a fault occurs, release locks, abort and deliver that fault;
```

```
  IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
```

```
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
```

```
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
```

```
ENDFOR
```

(\* Compute address of GPR area\*)

```
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
```

```
Check that DS:TMP_SSA_PAGE is read/write accessible;
```

```
If a fault occurs, release locks, abort and deliver that fault;
```

```
IF (DS:TMP_GPR does not resolve to EPC page)
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).VALID = 0)
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
(EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
(EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
THEN #PF(DS:TMP_GPR); FI;
```

```
IF (TMP_MODE64 = 0)
THEN
    IF (TMP_GPR + (GPR_SIZE - 1) is not in DS segment) THEN #GP(0); FI;
FI;
```

```
CR_GPR_PA := Physical_Address (DS: TMP_GPR);
```

```
TMP_TARGET := (DS:TMP_GPR).RIP;
IF (TMP_MODE64 = 1)
THEN
    IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
ELSE
    IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;
```

(\* Check proposed FS/GS segments fall within DS \*)

```
IF (TMP_MODE64 = 0)
THEN
    TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
    TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
    (* if FS wrap-around, make sure DS has no holes*)
    IF (TMP_FSLIMIT < TMP_FSBASE)
    THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
    ELSE
        IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
    (* if GS wrap-around, make sure DS has no holes*)
    IF (TMP_GSLIMIT < TMP_GSBASE)
    THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
    ELSE
        IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
ELSE
    TMP_FSBASE := DS:TMP_GPR.FSBASE;
    TMP_GSBASE := DS:TMP_GPR.GSBASE;
    IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
    THEN #GP(0); FI;
FI;
```

(\* Ensure the enclave is not already active and this thread is the only one using the TCS\*)

```
IF (DS:RBX.STATE = ACTIVE)
THEN #GP(0); FI;
```

```
TMP_IA32_U_CET := 0
```

```
TMP_SSP := 0
```

```

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
  THEN
    IF ( CR4.CET = 0 )
      THEN
        (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
        IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 ) #GP(0); FI;
      FI;
    (* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail ERESUME *)
    IF ( TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1 )
      THEN
        IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
      FI;

```

```

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
  THEN

```

```

  (* Setup CET state from SECS, note tracker goes to IDLE *)
  TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
  IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1 )
    THEN
      TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
      TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
    FI;

```

```

  (* Compute linear address of what will become new CET state save area and cache its PA *)
  TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA - 1) * 16
  TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

```

Check the TMP\_CET\_SAVE\_PAGE page is read/write accessible  
 If fault occurs release locks, abort and deliver fault

```

  (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
  IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
    (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
    THEN
      #PF(DS:TMP_CET_SAVE_PAGE);
    FI;

```

```

  CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)

```

```

  TMP_SSP = CR_CET_SAVE_AREA_PA.SSP
  TMP_IA32_U_CET.TRACKER = CR_CET_SAVE_AREA_PA.TRACKER;
  TMP_IA32_U_CET.SUPPRESS = CR_CET_SAVE_AREA_PA.SUPPRESS;

```

```

  IF ( (TMP_MODE64 = 1 AND TMP_SSP is not canonical) OR
    (TMP_MODE64 = 0 AND (TMP_SSP & 0xFFFFFFFFF0000000) ≠ 0) OR

```

```
(TMP_SSP is not 4 byte aligned) OR
(TMP_IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH AND TMP_IA32_U_CET.SUPPRESS = 1) OR
(CR_CET_SAVE_AREA_PA.Reserved ≠ 0) ) #GP(0); FI;
FI;
```

```
(* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)
(* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)
XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);
```

```
IF (XRSTOR failed with #GP)
  THEN
    DS:RBX.STATE := INACTIVE;
    #GP(0);
```

```
FI;
```

```
CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_EL RANGE := (TMP_SECS.BASEADDR, TMP_SECS.SIZE);
```

```
(* Save state for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;
```

```
(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;
```

```
RIP := TMP_TARGET;
```

```
Restore_GPRs from DS:TMP_GPR;
```

```
(*Restore the RFLAGS values from SSA*)
RFLAGS.CF := DS:TMP_GPR.RFLAGS.CF;
RFLAGS.PF := DS:TMP_GPR.RFLAGS.PF;
RFLAGS.AF := DS:TMP_GPR.RFLAGS.AF;
RFLAGS.ZF := DS:TMP_GPR.RFLAGS.ZF;
RFLAGS.SF := DS:TMP_GPR.RFLAGS.SF;
RFLAGS.DF := DS:TMP_GPR.RFLAGS.DF;
RFLAGS.OF := DS:TMP_GPR.RFLAGS.OF;
RFLAGS.NT := DS:TMP_GPR.RFLAGS.NT;
RFLAGS.AC := DS:TMP_GPR.RFLAGS.AC;
RFLAGS.ID := DS:TMP_GPR.RFLAGS.ID;
RFLAGS.RF := DS:TMP_GPR.RFLAGS.RF;
RFLAGS.VM := 0;
IF (RFLAGS.IOPL = 3)
  THEN RFLAGS.IF := DS:TMP_GPR.RFLAGS.IF; FI;
```

```

IF (TCS.FLAGS.OPTIN = 0)
    THEN RFLAGS.TF := 0; FI;

(* If XSAVE is enabled, save XCRO and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCRO := XCRO;
    XCRO := TMP_SECS.ATTRIBUTES.XFRM;
FI;

(* Pop the SSA stack*)
(DS:RBX).CSSA := (DS:RBX).CSSA - 1;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS.L;
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress all code breakpoints that overlap with ELRANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress any MTF VM exits during execution of the enclave;
        Clear all pending debug exceptions;
        Clear any pending MTF VM exit;
    ELSE

```

Clear all pending debug exceptions;  
Clear pending MTF VM exits;

FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET\_IBT] = 1) OR (CPUID.(EAX=7, ECX=0):ECX[CET\_SS] = 1)

THEN

(\* Save enclosing application CET state into save registers \*)

CR\_SAVE\_IA32\_U\_CET := IA32\_U\_CET

(\* Setup enclave CET state \*)

IF CPUID.(EAX=07H, ECX=00h):ECX[CET\_SS] = 1

THEN

CR\_SAVE\_SSP := SSP

SSP := TMP\_SSP;

FI;

IA32\_U\_CET := TMP\_IA32\_U\_CET;

FI;

(\* Assure consistent translations \*)

Flush\_linear\_context;

Clear\_Monitor\_FSM;

Allow\_front\_end\_to\_begin\_fetch\_at\_new\_RIP;

### Flags Affected

RFLAGS.TF is cleared on opt-out entry

### Protected Mode Exceptions

#GP(0)	<p>If DS:RBX is not page aligned.</p> <p>If the enclave is not initialized.</p> <p>If the thread is not in the INACTIVE state.</p> <p>If CS, DS, ES or SS bases are not all zero.</p> <p>If executed in enclave mode.</p> <p>If part or all of the FS or GS segment specified by TCS is outside the DS segment.</p> <p>If any reserved field in the TCS FLAG is set.</p> <p>If the target address is not within the CS segment.</p> <p>If CR4.OSFXSR = 0.</p> <p>If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.</p> <p>If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory.</p> <p>If DS:RBX does not point to a valid TCS.</p> <p>If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.</p>

### 64-Bit Mode Exceptions

#GP(0)	<p>If DS:RBX is not page aligned.</p> <p>If the enclave is not initialized.</p> <p>If the thread is not in the INACTIVE state.</p> <p>If CS, DS, ES or SS bases are not all zero.</p> <p>If executed in enclave mode.</p> <p>If part or all of the FS or GS segment specified by TCS is outside the DS segment.</p> <p>If any reserved field in the TCS FLAG is set.</p>
--------	--



If the target address is not canonical.

If CR4.OSFXSR = 0.

If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.

If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

#PF(error code)

If a page fault occurs in accessing memory operands.

If DS:RBX does not point to a valid TCS.

If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT\_REG EPC page.

## 36.5 INTEL® SGX VIRTUALIZATION LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLV instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

**EDECVIRTCHILD—Decrement VIRTCHILDCNT in SECS**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLV[EDECVIRTCHILD]	IR	V/V	EAX[5]	This leaf function decrements the SECS VIRTCHILDCNT field.

**Instruction Operand Encoding**

Op/En	EAX		RBX	RCX
IR	EDECVIRTCHILD (In)	Return error code (Out)	Address of an enclave page (In)	Address of an SECS page (In)

**Description**

This instruction decrements the SECS VIRTCHILDCNT field. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

**EDECVIRTCHILD Memory Parameter Semantics**

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

**EDECVIRTCHILD Faulting Conditions**

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

**Concurrency Restrictions****Table 36-74. Base Concurrency Restrictions of EDECVIRTCHILD**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDECVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RCX]	Concurrent		

**Table 36-75. Additional Concurrency Restrictions of EDECVIRTUALCHILD**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDECVIRTUALCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EDECVIRTUALCHILD Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_VIRTUALCHILDCNT	Integer	64	Number of virtual child pages.

**EDECVIRTUALCHILD Return Value in RAX**

Error	Value	Description
No Error	0	EDECVIRTUALCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_INVALID_COUNTER		Attempt to decrement counter that is already zero.

(\* check alignment of DS:RBX \*)

```
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;
```

(\* check DS:RBX is an linear address of an EPC page \*)

```
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(\* check DS:RCX is an linear address of an EPC page \*)

```
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;
```

(\* Check the EPCPAGE for concurrency \*)

```
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;
```

(\* check that the EPC page is valid \*)

```
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(\* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent \*)

```
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
```

```

(EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
(EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
(EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
  THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS := Address_of_SECS_for (DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
    (* get the physical address of DS:RBX *)
    TMP_SECS := Physical_Address(DS:RBX);
ELSE
    (* EDECVIRTUALCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); FI;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); FI;

(* Atomically decrement virtchild counter and check for underflow *)
Locked_Decrement(SECS(TMP_SECS).VIRTCHILDCNT);
IF (There was an underflow) THEN
    Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_COUNTER;
    goto DONE;
FI;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

```

### Flags Affected

ZF is set if EDECVIRTUALCHILD fails due to concurrent operation with another SGX instruction, or if there is a VIRTCHILDCNT underflow. Otherwise cleared.

### Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If DS segment is unusable.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

**64-Bit Mode Exceptions**

- #GP(0)            If a memory address is in a non-canonical form.  
                  If a memory operand is not properly aligned.  
                  RBX does not refer to an enclave page associated with SECS referenced in RCX.
- #PF(error code)    If a page fault occurs in accessing memory operands.  
                  If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).  
                  If RCX does not refer to an SECS page.

**EINCVIRTCHILD—Increment VIRTCHILDCNT in SECS**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLV[EINCVIRTCHILD]	IR	V/V	EAX[5]	This leaf function increments the SECS VIRTCHILDCNT field.

**Instruction Operand Encoding**

Op/En	EAX		RBX	RCX
IR	EINCVIRTCHILD (In)	Return error code (Out)	Address of an enclave page (In)	Address of an SECS page (In)

**Description**

This instruction increments the SECS VIRTCHILDCNT field. This instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create a linear address. Segment override is not supported.

**EINCVIRTCHILD Memory Parameter Semantics**

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

**EINCVIRTCHILD Faulting Conditions**

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

**Concurrency Restrictions****Table 36-76. Base Concurrency Restrictions of EINCVIRTCHILD**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINCVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RCX]	Concurrent		

**Table 36-77. Additional Concurrency Restrictions of EINCVRTCHILD**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINCVRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EINCVRTCHILD Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

**EINCVRTCHILD Return Value in RAX**

Error	Value	Description
No Error	0	EINCVRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

(\* check alignment of DS:RBX \*)

```
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;
```

(\* check DS:RBX is an linear address of an EPC page \*)

```
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(\* check DS:RCX is an linear address of an EPC page \*)

```
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;
```

(\* Check the EPCPAGE for concurrency \*)

```
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;
```

(\* check that the EPC page is valid \*)

```
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(\* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent \*)

```
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
```



```

THEN
  (* get the SECS of DS:RBX *)
  TMP_SECS := Address_of_SECS_for_DS:RBX;
ELSE IF (EPCM( DS:RBX ).PAGE_TYPE = PT_SECS) THEN
  (* get the physical address of DS:RBX *)
  TMP_SECS := Physical_Address( DS:RBX );
ELSE
  (* EINCVIRTCHILD called on page of incorrect type *)
  #PF( DS:RBX, PFEC.SGX ); FI;

IF ( TMP_SECS ≠ Physical_Address( DS:RCX )) THEN
  #GP( 0 ); FI;

(* Atomically increment virtchild counter *)
Locked_Increment( SECS( TMP_SECS ).VIRTCHILDCNT );

```

```

RFLAGS.ZF := 0;
RAX := 0;

```

```

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

```

### Flags Affected

ZF is set if EINCVIRTCHILD fails due to concurrent operation with another SGX instruction; otherwise cleared.

### Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If DS segment is unusable.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

### 64-Bit Mode Exceptions

#GP(0)	<p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

## ESETCONTEXT—Set the ENCLAVECONTEXT Field in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLV[ESETCONTEXT]	IR	V/V	EAX[5]	This leaf function sets the ENCLAVECONTEXT field in SECS.

### Instruction Operand Encoding

Op/En	EAX		RCX	RDX
IR	ESETCONTEXT (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Context Value (In, EA)

### Description

The ESETCONTEXT leaf overwrites the ENCLAVECONTEXT field in the SECS. ECREATE and ELD of an SECS set the ENCLAVECONTEXT field in the SECS to the address of the SECS (for access later in ERDINFO). The ESETCONTEXT instruction allows a VMM to overwrite the default context value if necessary, for example, if the VMM is emulating ECREATE or ELD on behalf of the guest.

The content of RCX is an effective address of the SECS page to be updated, RDX contains the address pointing to the value to be stored in the SECS. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if:

- The operand is not properly aligned.
- RCX does not refer to an SECS page.

### ESETCONTEXT Memory Parameter Semantics

EPCPAGE	CONTEXT
Read access permitted by Enclave	Read/Write access permitted by Non Enclave

The instruction faults if any of the following:

### ESETCONTEXT Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

### Concurrency Restrictions

**Table 36-78. Base Concurrency Restrictions of ESETCONTEXT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ESETCONTEXT	SECS [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

Table 36-79. Additional Concurrency Restrictions of ESETCONTEXT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ESETCONTEXT	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

## Operation

### Temp Variables in ESETCONTEXT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_CONTEXT	CONTEXT	64	Data Value of CONTEXT.

### ESETCONTEXT Return Value in RAX

Error	Value	Description
No Error	0	ESETCONTEXT Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

(\* check alignment of the EPCPAGE (RCX) \*)

```
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); FI;
```

(\* check that EPCPAGE (DS:RCX) is the address of an EPC page \*)

```
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;
```

(\* check alignment of the CONTEXT field (RDX) \*)

```
IF (DS:RDX is not 8Byte Aligned) THEN
    #GP(0); FI;
```

(\* Load CONTEXT into local variable \*)

```
TMP_CONTEXT := DS:RDX
```

(\* Check the EPC page for concurrency \*)

```
IF (EPC page is being modified) THEN
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;
```

(\* check page validity \*)

```
IF (EPCM(DS:RCX).VALID = 0) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;
```

(\* check EPC page is an SECS page \*)

```
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN  
  #PF(DS:RCX, PFEC.SGX);  
FI;
```

```
(* load the context value into SECS(DS:RCX).ENCLAVECONTEXT *)  
SECS(DS:RCX).ENCLAVECONTEXT := TMP_CONTEXT;
```

```
RAX := 0;  
RFLAGS.ZF := 0;
```

```
DONE:  
(* clear flags *)  
RFLAGS.CF,PF,AF,OF,SF := 0;
```

### Flags Affected

ZF is set if ESETCONTEXT fails due to concurrent operation with another SGX instruction; otherwise cleared. CF, PF, AF, OF and SF are cleared.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If DS segment is unusable. If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

### 64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form. If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

## 28. Updates to Appendix A, Volume 3D

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: Removed reference to chapter 31 which has been removed from the manual. The removed chapter was previously titled Chapter 31, "Virtual-Machine Monitor Programming Considerations".

# APPENDIX A

## VMX CAPABILITY REPORTING FACILITY

The ability of a processor to support VMX operation and related instructions is indicated by `CPUID.1:ECX.VMX[bit 5] = 1`. A value 1 in this bit indicates support for VMX features.

Support for specific features detailed in Chapter 25 and other VMX chapters is determined by reading values from a set of capability MSR. These MSRs are indexed starting at MSR address 480H. VMX capability MSRs are read-only; an attempt to write them (with `WRMSR`) produces a general-protection exception (`#GP(0)`). They do not exist on processors that do not support VMX operation; an attempt to read them (with `RDMSR`) on such processors produces a general-protection exception (`#GP(0)`).

### A.1 BASIC VMX INFORMATION

The `IA32_VMX_BASIC` MSR (index 480H) consists of the following fields:

- Bits 30:0 contain the 31-bit VMCS revision identifier used by the processor. Processors that use the same VMCS revision identifier use the same size for VMCS regions (see subsequent item on bits 44:32).<sup>1</sup>
- Bit 31 is always 0.
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are clear).
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor's physical-address width.<sup>2</sup> If the bit is 1, these addresses are limited to 32 bits. This bit is always 0 for processors that support Intel 64 architecture.
- If bit 49 is read as 1, the logical processor supports the dual-monitor treatment of system-management interrupts and system-management mode. See Section 30.15 for details of this treatment.
- Bits 53:50 report the memory type that should be used for the VMCS, for data structures referenced by pointers in the VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions), and for the MSEG header. If software needs to access these data structures (e.g., to modify the contents of the MSR bitmaps), it can configure the paging structures to map them into the linear-address space. If it does so, it should establish mappings that use the memory type reported bits 53:50 in this MSR.<sup>3</sup>

As of this writing, all processors that support VMX operation indicate the write-back type. The values used are given in Table A-1.

**Table A-1. Memory Types Recommended for VMCS and Related Data Structures**

Value(s)	Field
0	Uncacheable (UC)
1-5	Not used
6	Write Back (WB)
7-15	Not used

1. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field in bits 31:0 of this MSR. For all processors produced prior to this change, bit 31 of this MSR was read as 0.
2. On processors that support Intel 64 architecture, the pointer must not set bits beyond the processor's physical address width.
3. Alternatively, software may map any of these regions or structures with the UC memory type. (This may be necessary for the MSEG header.) Doing so is discouraged unless necessary as it will cause the performance of software accesses to those structures to suffer.

- If bit 54 is read as 1, the processor reports information in the VM-exit instruction-information field on VM exits due to execution of the INS and OUTS instructions (see Section 26.2.5). This reporting is done only if this bit is read as 1.
- Bit 55 is read as 1 if any VMX controls that default to 1 may be cleared to 0. See Appendix A.2 for details. It also reports support for the VMX capability MSRs IA32\_VMX\_TRUE\_PINBASED\_CTLS, IA32\_VMX\_TRUE\_PROCBASED\_CTLS, IA32\_VMX\_TRUE\_EXIT\_CTLS, and IA32\_VMX\_TRUE\_ENTRY\_CTLS. See Appendix A.3.1, Appendix A.3.2, Appendix A.4, and Appendix A.5 for details.
- If bit 56 is read as 1, software can use VM entry to deliver a hardware exception with or without an error code, regardless of vector (see Section 25.2.1.3).
- The values of bits 47:45 and bits 63:57 are reserved and are read as 0.

## A.2 RESERVED CONTROLS AND DEFAULT SETTINGS

As noted in Chapter 25, “VM Entries”, certain VMX controls are reserved and must be set to a specific value (0 or 1) determined by the processor. The specific value to which a reserved control must be set is its **default setting**. Software can discover the default setting of a reserved control by consulting the appropriate VMX capability MSR (see Appendix A.3 through Appendix A.5).

Future processors may define new functionality for one or more reserved controls. Such processors would allow each newly defined control to be set either to 0 or to 1. Software that does not desire a control’s new functionality should set the control to its default setting. For that reason, it is useful for software to know the default settings of the reserved controls.

Default settings partition the various controls into the following classes:

- **Always-flexible.** These have never been reserved.
- **Default0.** These are (or have been) reserved with a default setting of 0.
- **Default1.** They are (or have been) reserved with a default setting of 1.

As noted in Appendix A.1, a logical processor uses bit 55 of the IA32\_VMX\_BASIC MSR to indicate whether any of the default1 controls may be 0:

- If bit 55 of the IA32\_VMX\_BASIC MSR is read as 0, all the default1 controls are reserved and must be 1. VM entry will fail if any of these controls are 0 (see Section 25.2.1).
- If bit 55 of the IA32\_VMX\_BASIC MSR is read as 1, not all the default1 controls are reserved, and some (but not necessarily all) may be 0. The CPU supports four (4) new VMX capability MSRs: IA32\_VMX\_TRUE\_PINBASED\_CTLS, IA32\_VMX\_TRUE\_PROCBASED\_CTLS, IA32\_VMX\_TRUE\_EXIT\_CTLS, and IA32\_VMX\_TRUE\_ENTRY\_CTLS. See Appendix A.3 through Appendix A.5 for details. (These MSRs are not supported if bit 55 of the IA32\_VMX\_BASIC MSR is read as 0.)

## A.3 VM-EXECUTION CONTROLS

There are separate capability MSRs for the pin-based VM-execution controls, the primary processor-based VM-execution controls, the secondary processor-based VM-execution controls, and the tertiary processor-based VM-execution controls. These are described in Appendix A.3.1, Appendix A.3.2, Appendix A.3.3, and Appendix A.3.4, respectively.

### A.3.1 Pin-Based VM-Execution Controls

The IA32\_VMX\_PINBASED\_CTLS MSR (index 481H) reports on the allowed settings of **most** of the pin-based VM-execution controls (see Section 23.6.1):

- Bits 31:0 indicate the **allowed 0-settings** of these controls. VM entry allows control X (bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

Exceptions are made for the pin-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 2, and 4; the corresponding bits of the IA32\_VMX\_PINBASED\_CTLMSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32\_VMX\_BASIC MSR:

- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, VM entry fails if any pin-based VM-execution control in the default1 class is 0.
- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_PINBASED\_CTLMSR (see below) reports which of the pin-based VM-execution controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the **allowed 1-settings** of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_PINBASED\_CTLMSR (index 48DH) reports on the allowed settings of **all** of the pin-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the pin-based VM-execution controls:

- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32\_VMX\_PINBASED\_CTLMSR. (The IA32\_VMX\_TRUE\_PINBASED\_CTLMSR is not supported.)
- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32\_VMX\_TRUE\_PINBASED\_CTLMSR. Assuming that software knows that the default1 class of pin-based VM-execution controls contains bits 1, 2, and 4, there is no need for software to consult the IA32\_VMX\_PINBASED\_CTLMSR.

### A.3.2 Primary Processor-Based VM-Execution Controls

The IA32\_VMX\_PROCBASED\_CTLMSR (index 482H) reports on the allowed settings of **most** of the primary processor-based VM-execution controls (see Section 23.6.2):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the primary processor-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

Exceptions are made for the primary processor-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 4–6, 8, 13–16, and 26; the corresponding bits of the IA32\_VMX\_PROCBASED\_CTLMSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32\_VMX\_BASIC MSR:

- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, VM entry fails if any of the primary processor-based VM-execution controls in the default1 class is 0.
- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_PROCBASED\_CTLMSR (see below) reports which of the primary processor-based VM-execution controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_PROCBASED\_CTLMSR (index 48EH) reports on the allowed settings of **all** of the primary processor-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the primary processor-based VM-execution controls:



- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, all information about the allowed settings of the primary processor-based VM-execution controls is contained in the IA32\_VMX\_PROCBASED\_CTLMSR. (The IA32\_VMX\_TRUE\_PROCBASED\_CTLMSR MSR is not supported.)
- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, all information about the allowed settings of the processor-based VM-execution controls is contained in the IA32\_VMX\_TRUE\_PROCBASED\_CTLMSR. Assuming that software knows that the default1 class of processor-based VM-execution controls contains bits 1, 4–6, 8, 13–16, and 26, there is no need for software to consult the IA32\_VMX\_PROCBASED\_CTLMSR.

### A.3.3 Secondary Processor-Based VM-Execution Controls

The IA32\_VMX\_PROCBASED\_CTLMSR2 (index 48BH) reports on the allowed settings of the secondary processor-based VM-execution controls (see Section 23.6.2). The following items provide details, including enforcement by VM entry:

- Bits 31:0 indicate the allowed 0-settings of these controls. These bits are always 0. This fact indicates that VM entry allows each bit of the secondary processor-based VM-execution controls to be 0 (reserved bits must be 0)
- Bits 63:32 indicate the allowed 1-settings of these controls; the 1-setting is not allowed for any reserved bit. VM entry allows control X (bit X of the secondary processor-based VM-execution controls) to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X and the “activate secondary controls” primary processor-based VM-execution control are both 1.

The IA32\_VMX\_PROCBASED\_CTLMSR2 MSR exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control (only if bit 63 of the IA32\_VMX\_PROCBASED\_CTLMSR MSR is 1).

### A.3.4 Tertiary Processor-Based VM-Execution Controls

The IA32\_VMX\_PROCBASED\_CTLMSR3 MSR (index 492H) reports on the allowed 1-settings of the tertiary processor-based VM-execution controls (see Section 23.6.2); the 1-setting is not allowed for any reserved bit.

VM entry allows control X (bit X of the tertiary processor-based VM-execution controls) to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if control X and the “activate tertiary controls” primary processor-based VM-execution control are both 1.

The IA32\_VMX\_PROCBASED\_CTLMSR3 MSR exists only on processors that support the 1-setting of the “activate tertiary controls” VM-execution control (only if bit 49 of the IA32\_VMX\_PROCBASED\_CTLMSR MSR is 1).

Notice that the organization of this MSR differs from that of IA32\_VMX\_PROCBASED\_CTLMSR2 (Appendix A.3.3). This is because there are 64 tertiary processor-based VM-execution controls, while there were only 32 secondary processor-based VM-execution controls.

## A.4 VM-EXIT CONTROLS

The IA32\_VMX\_EXIT\_CTLMSR MSR (index 483H) reports on the allowed settings of **most** of the VM-exit controls (see Section 23.7.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-exit controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. Exceptions are made for the VM-exit controls in the default1 class (see Appendix A.2). These are bits 0–8, 10, 11, 13, 14, 16, and 17; the corresponding bits of the IA32\_VMX\_EXIT\_CTLMSR MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32\_VMX\_BASIC MSR:
  - If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, VM entry fails if any VM-exit control in the default1 class is 0.
  - If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_EXIT\_CTLMSR MSR (see below) reports which of the VM-exit controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_EXIT\_CTLMS MSR (index 48FH) reports on the allowed settings of **all** of the VM-exit controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-exit controls:

- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, all information about the allowed settings of the VM-exit controls is contained in the IA32\_VMX\_EXIT\_CTLMS MSR. (The IA32\_VMX\_TRUE\_EXIT\_CTLMS MSR is not supported.)
- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, all information about the allowed settings of the VM-exit controls is contained in the IA32\_VMX\_TRUE\_EXIT\_CTLMS MSR. Assuming that software knows that the default1 class of VM-exit controls contains bits 0–8, 10, 11, 13, 14, 16, and 17, there is no need for software to consult the IA32\_VMX\_EXIT\_CTLMS MSR.

## A.5 VM-ENTRY CONTROLS

The IA32\_VMX\_ENTRY\_CTLMS MSR (index 484H) reports on the allowed settings of **most** of the VM-entry controls (see Section 23.8.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-entry controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. Exceptions are made for the VM-entry controls in the default1 class (see Appendix A.2). These are bits 0–8 and 12; the corresponding bits of the IA32\_VMX\_ENTRY\_CTLMS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32\_VMX\_BASIC MSR:
  - If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, VM entry fails if any VM-entry control in the default1 class is 0.
  - If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_ENTRY\_CTLMS MSR (see below) reports which of the VM-entry controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X is 1 in the VM-entry controls and bit 32+X is 0 in this MSR.

If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, the IA32\_VMX\_TRUE\_ENTRY\_CTLMS MSR (index 490H) reports on the allowed settings of **all** of the VM-entry controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-entry controls:

- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 0, all information about the allowed settings of the VM-entry controls is contained in the IA32\_VMX\_ENTRY\_CTLMS MSR. (The IA32\_VMX\_TRUE\_ENTRY\_CTLMS MSR is not supported.)
- If bit 55 in the IA32\_VMX\_BASIC MSR is read as 1, all information about the allowed settings of the VM-entry controls is contained in the IA32\_VMX\_TRUE\_ENTRY\_CTLMS MSR. Assuming that software knows that the default1 class of VM-entry controls contains bits 0–8 and 12, there is no need for software to consult the IA32\_VMX\_ENTRY\_CTLMS MSR.

## A.6 MISCELLANEOUS DATA

The IA32\_VMX\_MISC MSR (index 485H) consists of the following fields:

- Bits 4:0 report a value X that specifies the relationship between the rate of the VMX-preemption timer and that of the timestamp counter (TSC). Specifically, the VMX-preemption timer (if it is active) counts down by 1 every time bit X in the TSC changes due to a TSC increment.
- If bit 5 is read as 1, VM exits store the value of IA32\_EFER.LMA into the “IA-32e mode guest” VM-entry control; see Section 26.2 for more details. This bit is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.
- Bits 8:6 report, as a bitmap, the activity states supported by the implementation:
  - Bit 6 reports (if set) the support for activity state 1 (HLT).
  - Bit 7 reports (if set) the support for activity state 2 (shutdown).
  - Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI).

If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. All implementations support VM entry to activity state 0 (active).

- If bit 14 is read as 1, Intel<sup>®</sup> Processor Trace (Intel PT) can be used in VMX operation. If the processor supports Intel PT but does not allow it to be used in VMX operation, execution of VMXON clears IA32\_RTIT\_CTL.TraceEn (see “VMXON—Enter VMX Operation” in Chapter 29); any attempt to write IA32\_RTIT\_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.
- If bit 15 is read as 1, the RDMSR instruction can be used in system-management mode (SMM) to read the IA32\_SMBASE MSR (MSR address 9EH). See Section 30.15.6.3.
- Bits 24:16 indicate the number of CR3-target values supported by the processor. This number is a value between 0 and 256, inclusive (bit 24 is set if and only if bits 23:16 are clear).
- Bits 27:25 is used to compute the recommended maximum number of MSRs that should appear in the VM-exit MSR-store list, the VM-exit MSR-load list, or the VM-entry MSR-load list. Specifically, if the value bits 27:25 of IA32\_VMX\_MISC is N, then  $512 * (N + 1)$  is the recommended maximum number of MSRs to be included in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).
- If bit 28 is read as 1, bit 2 of the IA32\_SMM\_MONITOR\_CTL can be set to 1. VMXOFF unblocks SMIs unless IA32\_SMM\_MONITOR\_CTL[bit 2] is 1 (see Section 30.14.4).
- If bit 29 is read as 1, software can use VMWRITE to write to any supported field in the VMCS; otherwise, VMWRITE cannot be used to modify VM-exit information fields.
- If bit 30 is read as 1, VM entry allows injection of a software interrupt, software exception, or privileged software exception with an instruction length of 0.
- Bits 63:32 report the 32-bit MSEG revision identifier used by the processor.
- Bits 13:9 and bit 31 are reserved and are read as 0.

## A.7 VMX-FIXED BITS IN CR0

The IA32\_VMX\_CR0\_FIXED0 MSR (index 486H) and IA32\_VMX\_CR0\_FIXED1 MSR (index 487H) indicate how bits in CR0 may be set in VMX operation. They report on bits in CR0 that are allowed to be 0 and to be 1, respectively, in VMX operation. If bit X is 1 in IA32\_VMX\_CR0\_FIXED0, then that bit of CR0 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32\_VMX\_CR0\_FIXED1, then that bit of CR0 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32\_VMX\_CR0\_FIXED0, then that bit is also 1 in IA32\_VMX\_CR0\_FIXED1; if bit X is 0 in IA32\_VMX\_CR0\_FIXED1, then that bit is also 0 in IA32\_VMX\_CR0\_FIXED0. Thus, each bit in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32\_VMX\_CR0\_FIXED0 and 1 in IA32\_VMX\_CR0\_FIXED1).

## A.8 VMX-FIXED BITS IN CR4

The IA32\_VMX\_CR4\_FIXED0 MSR (index 488H) and IA32\_VMX\_CR4\_FIXED1 MSR (index 489H) indicate how bits in CR4 may be set in VMX operation. They report on bits in CR4 that are allowed to be 0 and 1, respectively, in VMX operation. If bit X is 1 in IA32\_VMX\_CR4\_FIXED0, then that bit of CR4 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32\_VMX\_CR4\_FIXED1, then that bit of CR4 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32\_VMX\_CR4\_FIXED0, then that bit is also 1 in IA32\_VMX\_CR4\_FIXED1; if bit X is 0 in IA32\_VMX\_CR4\_FIXED1, then that bit is also 0 in IA32\_VMX\_CR4\_FIXED0. Thus, each bit in CR4 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32\_VMX\_CR4\_FIXED0 and 1 in IA32\_VMX\_CR4\_FIXED1).

## A.9 VMCS ENUMERATION

The IA32\_VMX\_VMCS\_ENUM MSR (index 48AH) provides information to assist software in enumerating fields in the VMCS.

As noted in Section 23.11.2, each field in the VMCS is associated with a 32-bit encoding which is structured as follows:

- Bits 31:15 are reserved (must be 0).
- Bits 14:13 indicate the field's width.
- Bit 12 is reserved (must be 0).
- Bits 11:10 indicate the field's type.
- Bits 9:1 is an index field that distinguishes different fields with the same width and type.
- Bit 0 indicates access type.

IA32\_VMX\_VMCS\_ENUM indicates to software the highest index value used in the encoding of any field supported by the processor:

- Bits 9:1 contain the highest index value used for any VMCS encoding.
- Bit 0 and bits 63:10 are reserved and are read as 0.

## A.10 VPID AND EPT CAPABILITIES

The IA32\_VMX\_EPT\_VPID\_CAP MSR (index 48CH) reports information about the capabilities of the logical processor with regard to virtual-processor identifiers (VPIDs, Section 27.1) and extended page tables (EPT, Section 27.2):

- If bit 0 is read as 1, the processor supports execute-only translations by EPT. This support allows software to configure EPT paging-structure entries in which bits 1:0 are clear (indicating that data accesses are not allowed) and bit 2 is set (indicating that instruction fetches are allowed).<sup>1</sup>
- Bit 6 indicates support for a page-walk length of 4.
- If bit 8 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be uncacheable (UC); see Section 23.6.11.
- If bit 14 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be write-back (WB).
- If bit 16 is read as 1, the logical processor allows software to configure a EPT PDE to map a 2-Mbyte page (by setting bit 7 in the EPT PDE).
- If bit 17 is read as 1, the logical processor allows software to configure a EPT PDPTE to map a 1-Gbyte page (by setting bit 7 in the EPT PDPTE).

---

1. If the "mode-based execute control for EPT" VM-execution control is 1, setting bit 0 indicates also that software may also configure EPT paging-structure entries in which bits 1:0 are both clear and in which bit 10 is set (indicating a translation that can be used to fetch instructions from a supervisor-mode linear address or a user-mode linear address).

- Support for the INVEPT instruction (see Chapter 29 and Section 27.3.3.1).
  - If bit 20 is read as 1, the INVEPT instruction is supported.
  - If bit 25 is read as 1, the single-context INVEPT type is supported.
  - If bit 26 is read as 1, the all-context INVEPT type is supported.
- If bit 21 is read as 1, accessed and dirty flags for EPT are supported (see Section 27.2.5).
- If bit 22 is read as 1, the processor reports advanced VM-exit information for EPT violations (see Section 26.2.1). This reporting is done only if this bit is read as 1.
- If bit 23 is read as 1, supervisor shadow-stack control is supported (see Section 27.2.3.2).
- Support for the INVVPID instruction (see Chapter 29 and Section 27.3.3.1).
  - If bit 32 is read as 1, the INVVPID instruction is supported.
  - If bit 40 is read as 1, the individual-address INVVPID type is supported.
  - If bit 41 is read as 1, the single-context INVVPID type is supported.
  - If bit 42 is read as 1, the all-context INVVPID type is supported.
  - If bit 43 is read as 1, the single-context-retaining-globals INVVPID type is supported.
- Bits 5:1, bit 7, bits 13:9, bit 15, bits 19:18, bit 24, bits 31:27, bits 39:33, and bits 63:44 are reserved and are read as 0.

The IA32\_VMX\_EPT\_VPID\_CAP MSR exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control (only if bit 63 of the IA32\_VMX\_PROCBASED\_CTLMSR MSR is 1) and that support either the 1-setting of the “enable EPT” VM-execution control (only if bit 33 of the IA32\_VMX\_PROCBASED\_CTLMSR2 MSR is 1) or the 1-setting of the “enable VPID” VM-execution control (only if bit 37 of the IA32\_VMX\_PROCBASED\_CTLMSR2 MSR is 1).

## A.11 VM FUNCTIONS

The IA32\_VMX\_VMFUNC MSR (index 491H) reports on the allowed settings of the VM-function controls (see Section 23.6.14). VM entry allows bit X of the VM-function controls to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if bit X of the VM-function controls, the “activate secondary controls” primary processor-based VM-execution control, and the “enable VM functions” secondary processor-based VM-execution control are all 1.

The IA32\_VMX\_VMFUNC MSR exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control (only if bit 63 of the IA32\_VMX\_PROCBASED\_CTLMSR MSR is 1) and the 1-setting of the “enable VM functions” secondary processor-based VM-execution control (only if bit 45 of the IA32\_VMX\_PROCBASED\_CTLMSR2 MSR is 1).

## 29. Updates to Chapter 1, Volume 4

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers (MSRs)*.

-----  
Changes to chapter: Updated section 1.1 "Intel® 64 and IA-32 Processors Covered in this Manual". Removed link to outdated reference document.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (order number 253665).
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- *The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* address the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

## 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series



## ABOUT THIS MANUAL

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Core™2 Extreme QX9000 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes.
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family
- 8th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
- Intel® Xeon® E processors
- 9th generation Intel® Core™ processors
- 2nd generation Intel® Xeon® Processor Scalable Family



- 10th generation Intel® Core™ processors
- 11th generation Intel® Core™ processors
- 3rd generation Intel® Xeon® Processor Scalable Family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Airmont microarchitecture.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Atom™ processor C series, the Intel® Atom™ processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Processor Scalable Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Processor Scalable Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual's content follows:

**Chapter 1 — About This Manual.** Gives an overview of all eight volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — Model-Specific Registers (MSRs).** Lists the MSRs available in Intel processors, and describes their functions.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

## 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

### NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

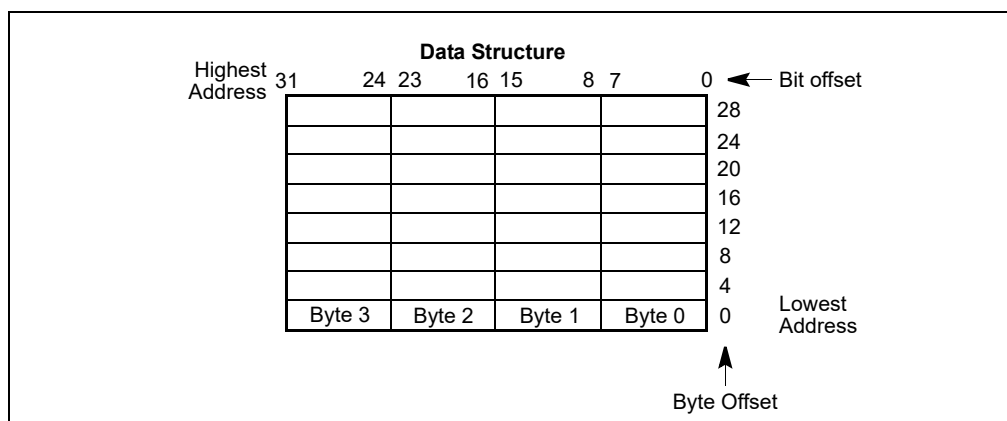


Figure 1-1. Bit and Byte Order

## 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of assembly language is used. In this subset, an instruction has the following format:

label: mnemonic argument1, argument2, argument3

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

LOADREG: MOV EAX, SUBTOTAL

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

### 1.3.6 Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a single syntax to represent this type of information. See Figure 1-2.

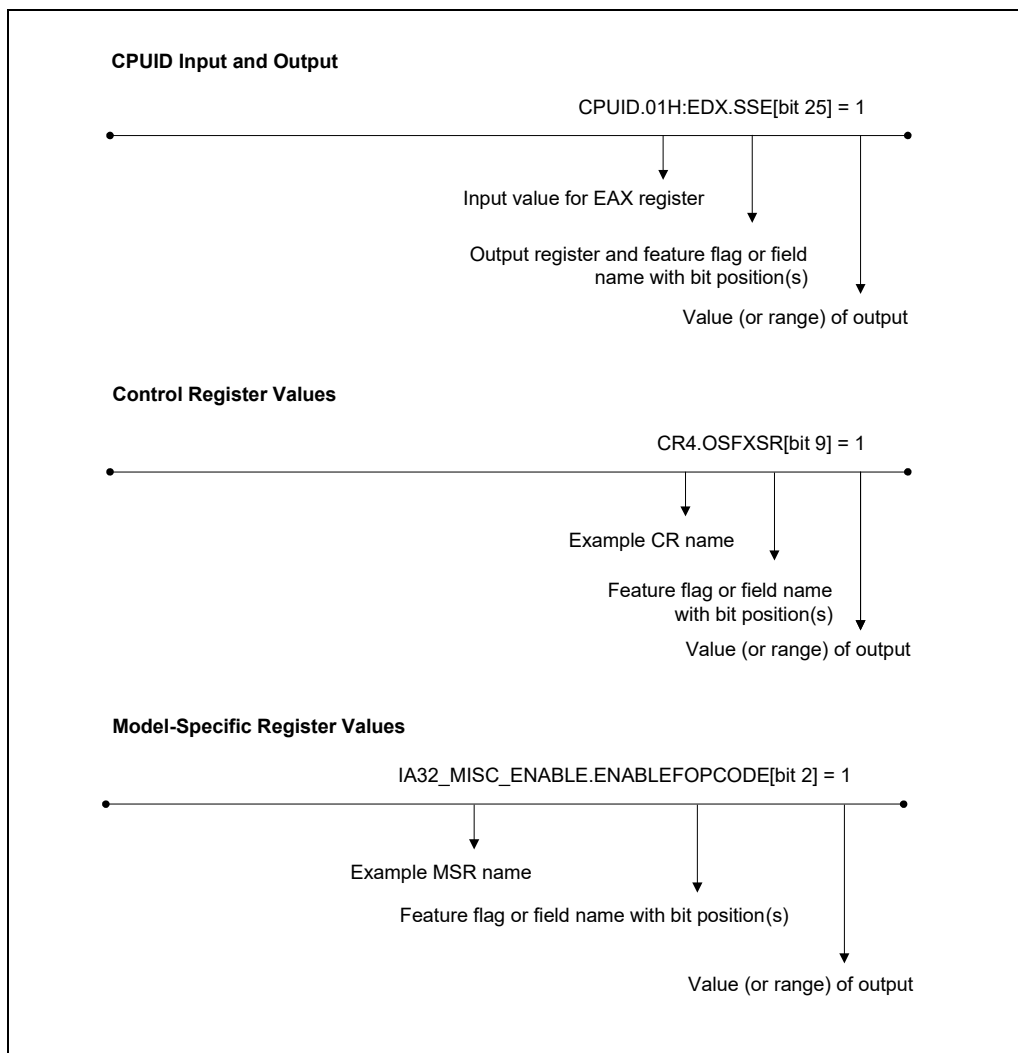


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

### 1.3.7 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

<https://software.intel.com/en-us/articles/intel-sdm>

See also:

- The latest security information on Intel® products:  
<https://www.intel.com/content/www/us/en/security-center/default.html>
- Software developer resources, guidance and insights for security advisories:  
<https://software.intel.com/security-software-guidance/>
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Software Development Tools:  
<https://software.intel.com/en-us/intel-sdp-home>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):  
<https://software.intel.com/en-us/articles/intel-sdm>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<https://software.intel.com/en-us/articles/intel-sdm#optimization>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Intel® Software Guard Extensions (Intel® SGX) Information  
<https://software.intel.com/en-us/isa-extensions/intel-sgx>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
<https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf>
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<https://software.intel.com/sites/default/files/22/30/25602>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference  
<https://software.intel.com/en-us/isa-extensions>

More relevant links are:

- Intel® Developer Zone:  
<https://software.intel.com/en-us>
- Developer centers:  
<http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>

### 30. Updates to Chapter 2, Volume 4

Change bars and green text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers (MSRs)*.

-----  
Changes to chapter: Update to IA32\_FIXED\_CTR\_CTRL MSR. Updated previous references to chapter 19 to point users to the online performance monitoring event lists (<https://perfmon-events.intel.com/>). Updated the definition of "module" for some Intel Atom® processors.

## CHAPTER 2

# MODEL-SPECIFIC REGISTERS (MSRS)

This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions. The scope of an MSR defines the set of processors that access the same MSR with RDMSR and WRMSR. Thread-scope MSRs are unique to every logical processor. Core-scope MSRs are shared by the threads in the same core; similarly for module-scope, die-scope, and package-scope.

When a processor package contains a single die, die-scope and package-scope are synonymous. When a package contains multiple die, they are distinct.

### NOTE

For information on hierarchical level types supported, refer to the CPUID Leaf 1FH definition for the actual level type numbers: "V2 Extended Topology Enumeration Leaf" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Also see Section 8.9.1, "Hierarchical Mapping of Shared Resources" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Table 2-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

**Table 2-1. CPUID Signature Values of DisplayFamily\_DisplayModel**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_85H	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture
06_8CH, 06_8DH	11th generation Intel® Core™ processors based on Tiger Lake microarchitecture
06_7DH, 06_7EH	10th generation Intel® Core™ processors based on Ice Lake microarchitecture
06_A5H, 06_A6H	10th generation Intel® Core™ processors based on Comet Lake microarchitecture
06_66H	Intel® Core™ processors based on Cannon Lake microarchitecture
06_8EH, 06_9EH	7th generation Intel® Core™ processors based on Kaby Lake microarchitecture, 8th and 9th generation Intel® Core™ processors based on Coffee Lake microarchitecture, Intel® Xeon® E processors based on Coffee Lake microarchitecture
06_6AH, 06_6CH	3rd generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture
06_55H	Intel® Xeon® Processor Scalable Family based on Skylake microarchitecture, 2nd generation Intel® Xeon® Processor Scalable Family based on Cascade Lake product, and 3rd generation Intel® Xeon® Processor Scalable Family based on Cooper Lake product
06_4EH, 06_5EH	6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture
06_56H	Intel Xeon processor D-1500 product family based on Broadwell microarchitecture
06_4FH	Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition
06_47H	5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture



**Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily\_DisplayModel (Contd.)**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_3DH	Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture
06_3FH	Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition
06_3CH, 06_45H, 06_46H	4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture
06_3EH	Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture
06_3EH	Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition
06_3AH	3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture
06_2DH	Intel Xeon processor E5 Family based on Intel microarchitecture code name Sandy Bridge, Intel Core i7-39xx Processor Extreme Edition
06_2FH	Intel Xeon Processor E7 Family
06_2AH	Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon processor 7500, 6500 series
06_25H, 06_2CH	Intel Xeon processors 3600, 5600 series, Intel Core i7, i5 and i3 Processors
06_1EH, 06_1FH	Intel Core i7 and i5 Processors
06_1AH	Intel Core i7 Processor, Intel Xeon processor 3400, 3500, 5500 series
06_1DH	Intel Xeon processor MP 7400 series
06_17H	Intel Xeon processor 3100, 3300, 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series
06_0FH	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_86H	Intel® Atom™ processors based on Tremont Microarchitecture
06_7AH	Intel Atom processors based on Goldmont Plus Microarchitecture
06_5FH	Intel Atom processors based on Goldmont Microarchitecture (code name Denverton)
06_5CH	Intel Atom processors based on Goldmont Microarchitecture
06_4CH	Intel Atom processor X7-Z8000 and X5-Z8000 series based on Airmont Microarchitecture
06_5DH	Intel Atom processor X3-C3000 based on Silvermont Microarchitecture
06_5AH	Intel Atom processor Z3500 series
06_4AH	Intel Atom processor Z3400 series
06_37H	Intel Atom processor E3000 series, Z3600 series, Z3700 series
06_4DH	Intel Atom processor C2000 series
06_36H	Intel Atom processor S1000 Series
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	Intel Atom processor family, Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon processor, Intel Xeon processor MP, Intel Pentium 4, Pentium D processors

**Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily\_DisplayModel (Contd.)**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon processor, Intel Pentium III processor
06_03H, 06_05H	Intel Pentium II Xeon processor, Intel Pentium II processor
06_01H	Intel Pentium Pro processor
05_01H, 05_02H, 05_04H	Intel Pentium processor, Intel Pentium processor with MMX Technology

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily\_DisplayModel = 05\_09H and SteppingID = 0

## 2.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these “architectural MSRs” were given the prefix “IA32\_”. Table 2-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 2-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific. Code that accesses a model-specific MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 2-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of “DF\_DM” (see Table 2-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as “MAXPHYADDR” in Table 2-2. “MAXPHYADDR” is reported by CPUID.8000\_0008H leaf.

MSR address range between 40000000H - 400000FFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

**Table 2-2. IA-32 Architectural MSRs**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
0H	0	IA32_P5_MC_ADDR (P5_MC_ADDR)	See Section 2.23, “MSRs in Pentium Processors.”	Pentium Processor (05_01H)
1H	1	IA32_P5_MC_TYPE (P5_MC_TYPE)	See Section 2.23, “MSRs in Pentium Processors.”	DF_DM = 05_01H
6H	6	IA32_MONITOR_FILTER_SIZE	See Section 8.10.5, “Monitor/Mwait Address Range Determination.”	0F_03H
10H	16	IA32_TIME_STAMP_COUNTER (TSC)	See Section 17.17, “Time-Stamp Counter.”	05_01H
17H	23	IA32_PLATFORM_ID (MSR_PLATFORM_ID)	Platform ID (RO) The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.	06_01H

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		49:0	Reserved	
		52:50	Platform Id (RO) Contains information concerning the intended platform for the processor.  52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7	
		63:53	Reserved	
1BH	27	IA32_APIC_BASE (APIC_BASE)	This register holds the APIC base address, permitting the relocation of the APIC memory map. See Section 10.4.4, "Local APIC Status and Location" and Section 10.4.5, "Relocating the Local APIC Registers".	06_01H
		7:0	Reserved	
		8	BSP flag (R/W)	
		9	Reserved	
		10	Enable x2APIC mode.	06_1AH
		11	APIC Global Enable (R/W)	
		(MAXPHYADDR - 1):12	APIC Base (R/W)	
		63: MAXPHYADDR	Reserved	
3AH	58	IA32_FEATURE_CONTROL	Control Features in Intel 64 Processor (R/W)	If any one enumeration condition for defined bit field holds.
		0	Lock bit (R/WO): (1 = locked). When set, locks this MSR from being written; writes to this bit will result in GP(0).  Note: Once the Lock bit is set, the contents of this register cannot be modified. Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS. Hence, once the Lock bit is set, the entire IA32_FEATURE_CONTROL contents are preserved across RESET when PWRGOOD is not deasserted.	If any one enumeration condition for defined bit field position greater than bit 0 holds.

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		1	Enable VMX inside SMX operation (R/WL): This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).	If CPUID.01H:ECX[5] = 1 && CPUID.01H:ECX[6] = 1
		2	Enable VMX outside SMX operation (R/WL): This bit enables VMX for a system executive that does not require SMX. BIOS must set this bit only when the CPUID function 1 returns the VMX feature flag set (ECX bit 5).	If CPUID.01H:ECX[5] = 1
		7:3	Reserved	
		14:8	SENTER Local Function Enables (R/WL): When set, each bit in the field represents an enable control for a corresponding SENTER function. This field is supported only if CPUID.1:ECX.[bit 6] is set.	If CPUID.01H:ECX[6] = 1
		15	SENTER Global Enable (R/WL): This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set.	If CPUID.01H:ECX[6] = 1
		16	Reserved	
		17	SGX Launch Control Enable (R/WL): This bit must be set to enable runtime re-configuration of SGX Launch Control via the IA32_SGXLEPUBKEYHASHn MSR.	If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1
		18	SGX Global Enable (R/WL): This bit must be set to enable SGX leaf functions.	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		19	Reserved	
		20	LMCE On (R/WL): When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.	If IA32_MCG_CAP[27] = 1
		63:21	Reserved	
3BH	59	IA32_TSC_ADJUST	Per Logical Processor TSC Adjust (R/Write to clear)	If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1
		63:0	THREAD_ADJUST: Local offset value of the IA32_TSC for a logical processor. Reset value is zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.	

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
48H	72	IA32_SPEC_CTRL	Speculation Control (R/W) The MSR bits are defined as logical processor scope. On some core implementations, the bits may impact sibling logical processors on the same core. This MSR has a value of 0 after reset and is unaffected by INIT# or SIPI#.	If any one of the enumeration conditions for defined bit field positions holds.
		0	Indirect Branch Restricted Speculation (IBRS). Restricts speculation of indirect branch.	If CPUID.(EAX=07H, ECX=0):EDX[26]=1
		1	Single Thread Indirect Branch Predictors (STIBP). Prevents indirect branch predictions on all logical processors on the core from being controlled by any sibling logical processor in the same core.	If CPUID.(EAX=07H, ECX=0):EDX[27]=1
		2	Speculative Store Bypass Disable (SSBD) delays speculative execution of a load until the addresses for all older stores are known.	If CPUID.(EAX=07H, ECX=0):EDX[31]=1
		63:3	Reserved	
49H	73	IA32_PRED_CMD	Prediction Command (W0) Gives software a way to issue commands that affect the state of predictors.	If any one of the enumeration conditions for defined bit field positions holds.
		0	Indirect Branch Prediction Barrier (IBPB).	If CPUID.(EAX=07H, ECX=0):EDX[26]=1
		63:1	Reserved	
79H	121	IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)	BIOS Update Trigger (W) Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 9.1.1.6, "Microcode Update Loader." A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
8BH	139	IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)	BIOS Update Signature (RO) Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
		31:0	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:32	It is recommended that this field be pre-loaded with zero prior to executing CPUID. If the field remains zero following the execution of CPUID, this indicates that no microcode update is loaded. Any non-zero value is the microcode update signature.	
8CH	140	IA32_SGXLEPUBKEYHASH0	IA32_SGXLEPUBKEYHASH[63:0] (R/W) Bits 63:0 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && CPUID.(EAX=07H, ECX=0H):ECX[30]=1. Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1.
8DH	141	IA32_SGXLEPUBKEYHASH1	IA32_SGXLEPUBKEYHASH[127:64] (R/W) Bits 127:64 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	
8EH	142	IA32_SGXLEPUBKEYHASH2	IA32_SGXLEPUBKEYHASH[191:128] (R/W) Bits 191:128 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	
8FH	143	IA32_SGXLEPUBKEYHASH3	IA32_SGXLEPUBKEYHASH[255:192] (R/W) Bits 255:192 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	
9BH	155	IA32_SMM_MONITOR_CTL	SMM Monitor Configuration (R/W)	If CPUID.01H: ECX[5]=1    CPUID.01H: ECX[6] = 1
		0	Valid (R/W)	
		1	Reserved	
		2	Controls SMI unblocking by VMXOFF (see Section 30.14.4).	If IA32_VMX_MISC[28]
		11:3	Reserved	
		31:12	MSEG Base (R/W)	
		63:32	Reserved	
9EH	158	IA32_SMBASE	Base address of the logical processor's SMRAM image (RO, SMM only).	If IA32_VMX_MISC[15]
C1H	193	IA32_PMC0 (PERFCTR0)	General Performance Counter 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
C2H	194	IA32_PMC1 (PERFCTR1)	General Performance Counter 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
C3H	195	IA32_PMC2	General Performance Counter 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C4H	196	IA32_PMC3	General Performance Counter 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
C5H	197	IA32_PMC4	General Performance Counter 4 (R/W)	If CPUID.0AH: EAX[15:8] > 4
C6H	198	IA32_PMC5	General Performance Counter 5 (R/W)	If CPUID.0AH: EAX[15:8] > 5
C7H	199	IA32_PMC6	General Performance Counter 6 (R/W)	If CPUID.0AH: EAX[15:8] > 6
C8H	200	IA32_PMC7	General Performance Counter 7 (R/W)	If CPUID.0AH: EAX[15:8] > 7
CFH	207	IA32_CORE_CAPABILITIES	IA32 Core Capabilities Register	If CPUID.(EAX=07H, ECX=0):EDX[30] = 1
		63:0	Reserved.	No architecturally defined bits.
E1H	225	IA32_UMWAIT_CONTROL	UMWAIT Control (R/W)	
		0	C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.	
		1	Reserved	
		31:2	Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.	
E7H	231	IA32_MPERF	TSC Frequency Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	C0_MCNT: C0 TSC Frequency Clock Count Increments at fixed interval (relative to TSC freq.) when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_APERF.	
E8H	232	IA32_APERF	Actual Performance Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	C0_ACNT: C0 Actual Frequency Clock Count Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_MPERF.	
FEH	254	IA32_MTRRCAP (MTRRcap)	MTRR Capability (RO) See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."	06_01H
		7:0	VCNT: The number of variable memory type ranges in the processor.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		8	Fixed range MTRRs are supported when set.	
		9	Reserved	
		10	wC Supported when set.	
		11	SMRR Supported when set.	
		12	PRMRR supported when set.	
		63:13	Reserved	
10AH	266	IA32_ARCH_CAPABILITIES	Enumeration of Architectural Features (RO)	If CPUID.(EAX=07H, ECX=0):EDX[29]=1
		0	RDCL_NO: The processor is not susceptible to Rogue Data Cache Load (RDCL).	
		1	IBRS_ALL: The processor supports enhanced IBRS.	
		2	RSBA: The processor supports RSB Alternate. Alternative branch predictors may be used by RET instructions when the RSB is empty. SW using retpoline may be affected by this behavior.	
		3	SKIP_L1DFL_VMENTRY: A value of 1 indicates the hypervisor need not flush the L1D on VM entry.	
		4	SSB_NO: Processor is not susceptible to Speculative Store Bypass.	
		5	MDS_NO: Processor is not susceptible to Microarchitectural Data Sampling (MDS).	
		6	IF_PCHANGE_MC_NO: The processor is not susceptible to a machine check error due to modifying the size of a code page without TLB invalidation.	
		7	TSX_CTRL: If 1, indicates presence of IA32_TSX_CTRL MSR.	
		8	TAA_NO: If 1, processor is not affected by TAA.	
		63:9	Reserved	
10BH	267	IA32_FLUSH_CMD	Flush Command (wO) Gives software a way to invalidate structures with finer granularity than other architectural methods.	If any one of the enumeration conditions for defined bit field positions holds.
		0	L1D_FLUSH: Writeback and invalidate the L1 data cache.	If CPUID.(EAX=07H, ECX=0):EDX[28]=1
		63:1	Reserved	



**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
122H	290	IA32_TSX_CTRL	IA32_TSX_CTRL	Thread scope. Not architecturally serializing. Available when CPUID.ARCH_CAP(EAX=7H, ECX = 0);EDX[29] = 1 and IA32_ARCH_CAPABILITIES. bit 7 = 1.
		0	RTM_DISABLE: When set to 1, XBEGIN will always abort with EAX code 0.	
		1	TSX_CPUID_CLEAR: When set to 1, CPUID.07H.EBX.RTM [bit 11] and CPUID.07H.EBX.HLE [bit 4] report 0. When set to 0 and the SKU supports TSX, these bits will return 1.	
		63:2	Reserved	
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR (R/W)	06_01H
		15:0	CS Selector.	
		31:16	Not used.	Can be read and written.
		63:32	Not used.	Writes ignored; reads return zero.
175H	373	IA32_SYSENTER_ESP	SYSENTER_ESP_MSR (R/W)	06_01H
176H	374	IA32_SYSENTER_EIP	SYSENTER_EIP_MSR (R/W)	06_01H
179H	377	IA32_MCG_CAP (MCG_CAP)	Global Machine Check Capability (RO)	06_01H
		7:0	Count: Number of reporting banks.	
		8	MCG_CTL_P: IA32_MCG_CTL is present if this bit is set.	
		9	MCG_EXT_P: Extended machine check state registers are present if this bit is set.	
		10	MCP_CMCI_P: Support for corrected MC error event is present.	06_01H
		11	MCG_TES_P: Threshold-based error status register are present if this bit is set.	
		15:12	Reserved	
		23:16	MCG_EXT_CNT: Number of extended machine check state registers present.	
		24	MCG_SER_P: The processor supports software error recovery if this bit is set.	
		25	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		26	MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format "Generic Error Data Entry" that augments the data included in machine check bank registers.	06_3EH
		27	MCG_LMCE_P: Indicates that the processor supports extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).	06_3EH
		63:28	Reserved	
17AH	378	IA32_MCG_STATUS (MCG_STATUS)	Global Machine Check Status (R/W0)	06_01H
		0	RIPV. Restart IP valid.	06_01H
		1	EIPV. Error IP valid.	06_01H
		2	MCIIP. Machine check in progress.	06_01H
		3	LMCE_S	If IA32_MCG_CAP.LMCE_P[27] = 1
		63:4	Reserved	
17BH	379	IA32_MCG_CTL (MCG_CTL)	Global Machine Check Control (R/W)	If IA32_MCG_CAP.CTL_P[8] = 1
180H- 185H	384- 389	Reserved		06_0EH <sup>1</sup>
186H	390	IA32_PERFEVTSELO (PERFEVTSELO)	Performance Event Select Register 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
		7:0	Event Select: Selects a performance event logic unit.	
		15:8	UMask: Qualifies the microarchitectural condition to detect on the selected event logic.	
		16	USR: Counts while in privilege level is not ring 0.	
		17	OS: Counts while in privilege level is ring 0.	
		18	Edge: Enables edge detection if set.	
		19	PC: Enables pin control.	
		20	INT: Enables interrupt on counter overflow.	

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		21	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	
		22	EN: Enables the corresponding performance counter to commence counting when this bit is set.	
		23	INV: Invert the CMASK.	
		31:24	CMASK: When CMASK is not zero, the corresponding performance counter increments each cycle if the event count is greater than or equal to the CMASK.	
		63:32	Reserved	
187H	391	IA32_PERFEVTSEL1 (PERFEVTSEL1)	Performance Event Select Register 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
188H	392	IA32_PERFEVTSEL2	Performance Event Select Register 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
189H	393	IA32_PERFEVTSEL3	Performance Event Select Register 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
18AH-197H	394-407	Reserved		06_0EH <sup>2</sup>
198H	408	IA32_PERF_STATUS	Current Performance Status (RO) See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	0F_03H
		15:0	Current performance State Value.	
		63:16	Reserved	
199H	409	IA32_PERF_CTL	Performance Control MSR (R/W) Software makes a request for a new Performance state (P-State) by writing this MSR. See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	0F_03H
		15:0	Target performance State Value.	
		31:16	Reserved	
		32	IDA Engage (R/W) When set to 1: disengages IDA.	06_0FH (Mobile only)
		63:33	Reserved	
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation Control (R/W) See Section 14.8.3, "Software Controlled Clock Modulation."	If CPUID.01H:EDX[22] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	Extended On-Demand Clock Modulation Duty Cycle.	If CPUID.06H:EAX[5] = 1
		3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation.	If CPUID.01H:EDX[22] = 1
		4	On-Demand Clock Modulation Enable: Set 1 to enable modulation.	If CPUID.01H:EDX[22] = 1
		63:5	Reserved	
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 14.8.2, "Thermal Monitor."	If CPUID.01H:EDX[22] = 1
		0	High-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		1	Low-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		2	PROCHOT# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		3	FORCEPR# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		7:5	Reserved	
		14:8	Threshold #1 Value	If CPUID.01H:EDX[22] = 1
		15	Threshold #1 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		22:16	Threshold #2 Value	If CPUID.01H:EDX[22] = 1
		23	Threshold #2 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
		63:25	Reserved	
19CH	412	IA32_THERM_STATUS	Thermal Status Information (RO) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 14.8.2, "Thermal Monitor".	If CPUID.01H:EDX[22] = 1
		0	Thermal Status (RO)	If CPUID.01H:EDX[22] = 1
		1	Thermal Status Log (R/W)	If CPUID.01H:EDX[22] = 1
		2	PROCHOT # or FORCEPR# event (RO)	If CPUID.01H:EDX[22] = 1
		3	PROCHOT # or FORCEPR# log (R/WCO)	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Status (RO)	If CPUID.01H:EDX[22] = 1
		5	Critical Temperature Status log (R/WCO)	If CPUID.01H:EDX[22] = 1
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WCO)	If CPUID.01H:ECX[8] = 1

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
		9	Thermal Threshold #2 log (R/WCO)	If CPUID.01H:ECX[8] = 1
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WCO)	If CPUID.06H:EAX[4] = 1
		12	Current Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		13	Current Limit log (R/WCO)	If CPUID.06H:EAX[7] = 1
		14	Cross Domain Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		15	Cross Domain Limit log (R/WCO)	If CPUID.06H:EAX[7] = 1
		22:16	Digital Readout (RO)	If CPUID.06H:EAX[0] = 1
		26:23	Reserved	
		30:27	Resolution in Degrees Celsius (RO)	If CPUID.06H:EAX[0] = 1
		31	Reading Valid (RO)	If CPUID.06H:EAX[0] = 1
		63:32	Reserved	
1A0H	416	IA32_MISC_ENABLE	Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.	
		0	Fast-Strings Enable When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default). When clear, fast-strings are disabled.	OF_OH
		2:1	Reserved	
		3	Automatic Thermal Control Circuit Enable (R/W) 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled. Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2 and adaptive thermal throttling will still be activated. The default value of this field varies with product . See respective tables where default value is listed.	OF_OH
		6:4	Reserved	
		7	Performance Monitoring Available (R) 1 = Performance monitoring enabled. 0 = Performance monitoring disabled.	OF_OH
		10:8	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11	Branch Trace Storage Unavailable (RO) 1 = Processor doesn't support branch trace storage (BTS). 0 = BTS is supported.	0F_0H
		12	Processor Event Based Sampling (PEBS) Unavailable (RO) 1 = PEBS is not supported. 0 = PEBS is supported.	06_0FH
		15:13	Reserved	
		16	Enhanced Intel SpeedStep Technology Enable (R/W) 0 = Enhanced Intel SpeedStep Technology disabled. 1 = Enhanced Intel SpeedStep Technology enabled.	If CPUID.01H: ECX[7] = 1
		17	Reserved	
		18	ENABLE MONITOR FSM (R/W) When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported. Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0. When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1). If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.	0F_03H
		21:19	Reserved	

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		22	Limit CPUID Maxval (R/W) When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2. BIOS should contain a setup question that allows users to specify when the installed OS does not support CPUID functions greater than 2. Before setting this bit, BIOS must execute the CPUID.0H and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 2, this bit is supported. Otherwise, this bit is not supported. Setting this bit when the maximum value is not greater than 2 may generate a #GP exception. Setting this bit may cause unexpected behavior in software that depends on the availability of CPUID leaves greater than 2.	0F_03H
		23	xTPR Message Disable (R/W) When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.	If CPUID.01H:ECX[14] = 1
		33:24	Reserved	
		34	XD Bit Disable (R/W) When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0). When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages. BIOS must not alter the contents of this bit location, if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.	If CPUID.80000001H:EDX[20] = 1
		63:35	Reserved	
1B0H	432	IA32_ENERGY_PERF_BIAS	Performance Energy Bias Hint (R/W)	If CPUID.6H:ECX[3] = 1
		3:0	Power Policy Preference: 0 indicates preference to highest performance. 15 indicates preference to maximize energy saving.	
		63:4	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package Thermal Status Information (RO) Contains status information about the package's thermal sensor. See Section 14.9, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg Thermal Status (RO)	
		1	Pkg Thermal Status Log (R/W)	
		2	Pkg PROCHOT # event (RO)	
		3	Pkg PROCHOT # log (R/WCO)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status Log (R/WCO)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 Log (R/WCO)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 Log (R/WCO)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation Log (R/WCO)	
		15:12	Reserved	
		22:16	Pkg Digital Readout (RO)	
		25:23	Reserved	
26	Hardware Feedback Interface Structure Change Status	If CPUID.06H:EAX.[19] = 1		
63:27	Reserved			
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 14.9, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg High-Temperature Interrupt Enable	
		1	Pkg Low-Temperature Interrupt Enable	
		2	Pkg PROCHOT# Interrupt Enable	
		3	Reserved	
		4	Pkg Overheat Interrupt Enable	
		7:5	Reserved	
		14:8	Pkg Threshold #1 Value	
		15	Pkg Threshold #1 Interrupt Enable	
22:16	Pkg Threshold #2 Value			



**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	
		25	Hardware Feedback Interrupt Enable	If CPUID.06H:EAX.[19] = 1
		63:26	Reserved	
1D9H	473	IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)	Trace/Profile Resource Control (R/W)	06_0EH
		0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
		1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
		5:2	Reserved	
		6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
		7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
		8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
		9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH
		10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH
		11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 38FH) on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
		14	FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.	If IA32_PERF_CAPABILITIES[12] = 1
		15	RTM_DEBUG: When set, enables DR7 debug bit on XBEGIN.	If (CPUID.(EAX=07H, ECX=0):EBX[11] = 1)
		63:16	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
1F2H	498	IA32_SMRR_PHYSBASE	SMRR Base Address (Writeable only in SMM) Base address of SMM memory range.	If IA32_MTRRCAP.SMRR[11] = 1
		7:0	Type. Specifies memory type of the range.	
		11:8	Reserved	
		31:12	PhysBase SMRR physical Base Address.	
		63:32	Reserved	
1F3H	499	IA32_SMRR_PHYSMASK	SMRR Range Mask (Writeable only in SMM) Range Mask of SMM memory range.	If IA32_MTRRCAP[SMRR] = 1
		10:0	Reserved	
		11	Valid Enable range mask.	
		31:12	PhysMask SMRR address range mask.	
		63:32	Reserved	
1F8H	504	IA32_PLATFORM_DCA_CAP	DCA Capability (R)	If CPUID.01H: ECX[18] = 1
1F9H	505	IA32_CPU_DCA_CAP	If set, CPU supports Prefetch-Hint type.	If CPUID.01H: ECX[18] = 1
1FAH	506	IA32_DCA_0_CAP	DCA type 0 Status and Control register.	If CPUID.01H: ECX[18] = 1
		0	DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set.	
		2:1	TRANSACTION	
		6:3	DCA_TYPE	
		10:7	DCA_QUEUE_SIZE	
		12:11	Reserved	
		16:13	DCA_DELAY: Writes will update the register but have no HW side-effect.	
		23:17	Reserved	
		24	SW_BLOCK: SW can request DCA block by setting this bit.	
		25	Reserved	
		26	HW_BLOCK: Set when DCA is blocked by HW (e.g. CR0.CD = 1).	
31:27	Reserved			
200H	512	IA32_MTRR_PHYSBASE0 (MTRRphysBase0)	See Section 11.11.2.3, "Variable Range MTRRs."	If IA32_MTRRCAP[7:0] > 0
201H	513	IA32_MTRR_PHYSMASK0	MTRRphysMask0	If IA32_MTRRCAP[7:0] > 0
202H	514	IA32_MTRR_PHYSBASE1	MTRRphysBase1	If IA32_MTRRCAP[7:0] > 1
203H	515	IA32_MTRR_PHYSMASK1	MTRRphysMask1	If IA32_MTRRCAP[7:0] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
204H	516	IA32_MTRR_PHYSBASE2	MTRRphysBase2	If IA32_MTRRCAP[7:0] > 2
205H	517	IA32_MTRR_PHYSMASK2	MTRRphysMask2	If IA32_MTRRCAP[7:0] > 2
206H	518	IA32_MTRR_PHYSBASE3	MTRRphysBase3	If IA32_MTRRCAP[7:0] > 3
207H	519	IA32_MTRR_PHYSMASK3	MTRRphysMask3	If IA32_MTRRCAP[7:0] > 3
208H	520	IA32_MTRR_PHYSBASE4	MTRRphysBase4	If IA32_MTRRCAP[7:0] > 4
209H	521	IA32_MTRR_PHYSMASK4	MTRRphysMask4	If IA32_MTRRCAP[7:0] > 4
20AH	522	IA32_MTRR_PHYSBASE5	MTRRphysBase5	If IA32_MTRRCAP[7:0] > 5
20BH	523	IA32_MTRR_PHYSMASK5	MTRRphysMask5	If IA32_MTRRCAP[7:0] > 5
20CH	524	IA32_MTRR_PHYSBASE6	MTRRphysBase6	If IA32_MTRRCAP[7:0] > 6
20DH	525	IA32_MTRR_PHYSMASK6	MTRRphysMask6	If IA32_MTRRCAP[7:0] > 6
20EH	526	IA32_MTRR_PHYSBASE7	MTRRphysBase7	If IA32_MTRRCAP[7:0] > 7
20FH	527	IA32_MTRR_PHYSMASK7	MTRRphysMask7	If IA32_MTRRCAP[7:0] > 7
210H	528	IA32_MTRR_PHYSBASE8	MTRRphysBase8	If IA32_MTRRCAP[7:0] > 8
211H	529	IA32_MTRR_PHYSMASK8	MTRRphysMask8	If IA32_MTRRCAP[7:0] > 8
212H	530	IA32_MTRR_PHYSBASE9	MTRRphysBase9	If IA32_MTRRCAP[7:0] > 9
213H	531	IA32_MTRR_PHYSMASK9	MTRRphysMask9	If IA32_MTRRCAP[7:0] > 9
250H	592	IA32_MTRR_FIX64K_00000	MTRRfix64K_00000	If CPUID.01H: EDX.MTRR[12] = 1
258H	600	IA32_MTRR_FIX16K_80000	MTRRfix16K_80000	If CPUID.01H: EDX.MTRR[12] = 1
259H	601	IA32_MTRR_FIX16K_A0000	MTRRfix16K_A0000	If CPUID.01H: EDX.MTRR[12] = 1
268H	616	IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000 )	See Section 11.11.2.2, "Fixed Range MTRRs."	If CPUID.01H: EDX.MTRR[12] = 1
269H	617	IA32_MTRR_FIX4K_C8000	MTRRfix4K_C8000	If CPUID.01H: EDX.MTRR[12] = 1
26AH	618	IA32_MTRR_FIX4K_D0000	MTRRfix4K_D0000	If CPUID.01H: EDX.MTRR[12] = 1
26BH	619	IA32_MTRR_FIX4K_D8000	MTRRfix4K_D8000	If CPUID.01H: EDX.MTRR[12] = 1
26CH	620	IA32_MTRR_FIX4K_E0000	MTRRfix4K_E0000	If CPUID.01H: EDX.MTRR[12] = 1
26DH	621	IA32_MTRR_FIX4K_E8000	MTRRfix4K_E8000	If CPUID.01H: EDX.MTRR[12] = 1
26EH	622	IA32_MTRR_FIX4K_F0000	MTRRfix4K_F0000	If CPUID.01H: EDX.MTRR[12] = 1
26FH	623	IA32_MTRR_FIX4K_F8000	MTRRfix4K_F8000	If CPUID.01H: EDX.MTRR[12] = 1
277H	631	IA32_PAT	IA32_PAT (R/W)	If CPUID.01H: EDX.MTRR[16] = 1
		2:0	PA0	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:3	Reserved	
		10:8	PA1	
		15:11	Reserved	
		18:16	PA2	
		23:19	Reserved	
		26:24	PA3	
		31:27	Reserved	
		34:32	PA4	
		39:35	Reserved	
		42:40	PA5	
		47:43	Reserved	
		50:48	PA6	
		55:51	Reserved	
		58:56	PA7	
		63:59	Reserved	
280H	640	IA32_MCO_CTL2	MSR to enable/disable CMCI capability for bank 0. (R/W) See Section 15.3.2.5, "IA32_MCi_CTL2 MSRs".	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 0
		14:0	Corrected error count threshold.	
		29:15	Reserved	
		30	CMCI_EN	
		63:31	Reserved	
281H	641	IA32_MC1_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
282H	642	IA32_MC2_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2
283H	643	IA32_MC3_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
284H	644	IA32_MC4_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
285H	645	IA32_MC5_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
286H	646	IA32_MC6_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
287H	647	IA32_MC7_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
288H	648	IA32_MC8_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
289H	649	IA32_MC9_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
28AH	650	IA32_MC10_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
28BH	651	IA32_MC11_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11
28CH	652	IA32_MC12_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
28DH	653	IA32_MC13_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
28EH	654	IA32_MC14_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
28FH	655	IA32_MC15_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
290H	656	IA32_MC16_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
291H	657	IA32_MC17_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17
292H	658	IA32_MC18_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
293H	659	IA32_MC19_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
294H	660	IA32_MC20_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
295H	661	IA32_MC21_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
296H	662	IA32_MC22_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
297H	663	IA32_MC23_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
298H	664	IA32_MC24_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
299H	665	IA32_MC25_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
29AH	666	IA32_MC26_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26
29BH	667	IA32_MC27_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
29CH	668	IA32_MC28_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
29DH	669	IA32_MC29_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
29EH	670	IA32_MC30_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
29FH	671	IA32_MC31_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 31
2FFH	767	IA32_MTRR_DEF_TYPE	MTRRdefType (R/W)	If CPUID.01H: EDX.MTRR[12] = 1
		2:0	Default Memory Type	
		9:3	Reserved	
		10	Fixed Range MTRR Enable	
		11	MTRR Enable	
		63:12	Reserved	
309H	777	IA32_FIXED_CTR0	Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.	If CPUID.0AH: EDX[4:0] > 0
30AH	778	IA32_FIXED_CTR1	Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core.	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2	Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref.	If CPUID.0AH: EDX[4:0] > 2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
345H	837	IA32_PERF_CAPABILITIES	Read Only MSR that enumerates the existence of performance monitoring features. (RO)	If CPUID.01H: ECX[15] = 1
		5:0	LBR format	
		6	PEBS Trap	
		7	PEBSSaveArchRegs	
		11:8	PEBS Record Format	
		12	1: Freeze while SMM is supported.	
		13	1: Full width of counter writable via IA32_A_PMCx.	
		14	PEBS_BASELINE	
		15	1: Performance metrics available.	
		16	1: PEBS output will be written into the Intel PT trace stream.	If CPUID.0x7.0.EBX[25]=1
	63:17	Reserved		
38DH	909	IA32_FIXED_CTR_CTRL	Fixed-Function Performance Counter Control (R/W) Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.	If CPUID.0AH: EAX[7:0] > 1
		0	EN0_OS: Enable Fixed Counter 0 to count while CPL = 0.	
		1	EN0_Usr: Enable Fixed Counter 0 to count while CPL > 0.	
		2	AnyThr0: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH:EAX[7:0] > 2 && CPUID.0AH:EDX[15]=0
		3	EN0_PMI: Enable PMI when fixed counter 0 overflows.	
		4	EN1_OS: Enable Fixed Counter 1 to count while CPL = 0.	
		5	EN1_Usr: Enable Fixed Counter 1 to count while CPL > 0.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		6	<i>AnyThr1</i> : When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH:EAX[7:0] > 2 && CPUID.0AH:EDX[15]=0
		7	EN1_PMI: Enable PMI when fixed counter 1 overflows.	
		8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.	
		9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0.	
		10	<i>AnyThr2</i> : When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH:EAX[7:0] > 2 && CPUID.0AH:EDX[15]=0
		11	EN2_PMI: Enable PMI when fixed counter 2 overflows.	
		12	<i>EN3_OS</i> : Enable Fixed Counter 3 to count while CPL = 0.	
		13	<i>EN3_Usr</i> : Enable Fixed Counter 3 to count while CPL > 0.	
		14	Reserved	
		15	<i>EN3_PMI</i> : Enable PMI when fixed counter 3 overflows.	
		63:16	Reserved	
38EH	910	IA32_PERF_GLOBAL_STATUS	Global Performance Counter Status (RO)	If CPUID.0AH: EAX[7:0] > 0
		0	Ovf_PMC0: Overflow status of IA32_PMC0.	If CPUID.0AH: EAX[15:8] > 0
		1	Ovf_PMC1: Overflow status of IA32_PMC1.	If CPUID.0AH: EAX[15:8] > 1
		2	Ovf_PMC2: Overflow status of IA32_PMC2.	If CPUID.0AH: EAX[15:8] > 2
		3	Ovf_PMC3: Overflow status of IA32_PMC3.	If CPUID.0AH: EAX[15:8] > 3
		31:4	Reserved	
		32	Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.	If CPUID.0AH: EAX[7:0] > 1
		33	Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.	If CPUID.0AH: EAX[7:0] > 1



**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		34	Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.	If CPUID.0AH: EAX[7:0] > 1
		47:35	Reserved	
		48	OVF_PERF_METRICS: If this bit is set, it indicates that PERF_METRIC counter has overflowed and a PMI is triggered; however, an overflow of fixed counter 3 should normally happen first. If this bit is clear no overflow occurred.	
		54:49	Reserved	
		55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer that was completely filled.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		57:56	Reserved	
		58	LBR_Frz. LBRs are frozen due to: <ul style="list-style-type: none"> <li>▪ IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1.</li> <li>▪ The LBR stack overflowed.</li> </ul>	If CPUID.0AH: EAX[7:0] > 3
		59	CTR_Frz. Performance counters in the core PMU are frozen due to: <ul style="list-style-type: none"> <li>▪ IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1.</li> <li>▪ One or more core PMU counters overflowed.</li> </ul>	If CPUID.0AH: EAX[7:0] > 3
		60	ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation Intel SGX to protect an enclave.	If CPUID.(EAX=07H, ECX=0):EBX[2] = 1
		61	Ovf_Uncore: Uncore counter overflow status.	If CPUID.0AH: EAX[7:0] > 2
		62	OvfBuf: DS SAVE area Buffer overflow status.	If CPUID.0AH: EAX[7:0] > 0
		63	CondChgd: Status bits of this register have changed.	If CPUID.0AH: EAX[7:0] > 0
		38FH	911	IA32_PERF_GLOBAL_CTRL
0	EN_PMC0			If CPUID.0AH: EAX[15:8] > 0
1	EN_PMC1			If CPUID.0AH: EAX[15:8] > 1
2	EN_PMC2			If CPUID.0AH: EAX[15:8] > 2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		n	EN_PMCn	If CPUID.0AH: EAX[15:8] > n
		31:n+1	Reserved	
		32	EN_FIXED_CTR0	If CPUID.0AH: EDX[4:0] > 0
		33	EN_FIXED_CTR1	If CPUID.0AH: EDX[4:0] > 1
		34	EN_FIXED_CTR2	If CPUID.0AH: EDX[4:0] > 2
		47:35	Reserved	
		48	EN_PERF_METRICS: If this bit is set and fixed counter 3 is effectively enabled, built-in performance metrics are enabled.	
		63:49	Reserved	
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Global Performance Counter Overflow Control (R/W)	If CPUID.0AH: EAX[7:0] > 0 && CPUID.0AH: EAX[7:0] <= 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		60:56	Reserved	
		61	Set 1 to Clear Ovf_Uncore bit.	O6_2EH
		62	Set 1 to Clear OvfBuf bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set 1 to clear CondChgd bit.	If CPUID.0AH: EAX[7:0] > 0
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Global Performance Counter Overflow Reset Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		47:35	Reserved	
		48	RESET_OVF_PERF_METRICS: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.	
		54:49	Reserved	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA[8] = 1
		57:56	Reserved	
		58	Set 1 to Clear LBR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to Clear CTR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to Clear ASCII bit.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set 1 to clear CondChgd bit.	If CPUID.0AH: EAX[7:0] > 0
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Global Performance Counter Overflow Set Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to cause Ovf_PMC0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		1	Set 1 to cause Ovf_PMC1 = 1.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to cause Ovf_PMC2 = 1.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to cause Ovf_PMCn = 1.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved	
		32	Set 1 to cause Ovf_FIXED_CTR0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		33	Set 1 to cause Ovf_FIXED_CTR1 = 1.	If CPUID.0AH: EAX[7:0] > 3
		34	Set 1 to cause Ovf_FIXED_CTR2 = 1.	If CPUID.0AH: EAX[7:0] > 3
		47:35	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		48	SET_OVF_PERF_METRICS: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.	
		54:49	Reserved	
		55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.0AH: EAX[7:0] > 3
		57:56	Reserved	
		58	Set 1 to cause LBR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to cause CTR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to cause ASCII = 1.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to cause Ovf_Uncore = 1.	If CPUID.0AH: EAX[7:0] > 3
		62	Set 1 to cause OvfBuf = 1.	If CPUID.0AH: EAX[7:0] > 3
		63	Reserved	
392H	914	IA32_PERF_GLOBAL_INUSE	Indicator that core perfmon interface is in use. (RO)	If CPUID.0AH: EAX[7:0] > 3
		0	IA32_PERFEVTSELO in use.	
		1	IA32_PERFEVTSEL1 in use.	If CPUID.0AH: EAX[15:8] > 1
		2	IA32_PERFEVTSEL2 in use.	If CPUID.0AH: EAX[15:8] > 2
		n	IA32_PERFEVTSELn in use.	If CPUID.0AH: EAX[15:8] > n
		31:n+1	Reserved	
		32	IA32_FIXED_CTR0 in use.	
		33	IA32_FIXED_CTR1 in use.	
		34	IA32_FIXED_CTR2 in use.	
		62:35	Reserved or model specific.	
		63	PMI in use.	
3F1H	1009	IA32_PEBS_ENABLE	PEBS Control (R/W)	
		0	Enable PEBS on IA32_PMC0.	06_OFH
		3:1	Reserved or model specific.	
		31:4	Reserved	
		35:32	Reserved or model specific.	
		63:36	Reserved	
400H	1024	IA32_MCO_CTL	MCO_CTL	If IA32_MCG_CAP.CNT > 0
401H	1025	IA32_MCO_STATUS	MCO_STATUS	If IA32_MCG_CAP.CNT > 0
402H	1026	IA32_MCO_ADDR <sup>1</sup>	MCO_ADDR	If IA32_MCG_CAP.CNT > 0
403H	1027	IA32_MCO_MISC	MCO_MISC	If IA32_MCG_CAP.CNT > 0
404H	1028	IA32_MC1_CTL	MC1_CTL	If IA32_MCG_CAP.CNT > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
405H	1029	IA32_MC1_STATUS	MC1_STATUS	If IA32_MCG_CAP.CNT >1
406H	1030	IA32_MC1_ADDR <sup>2</sup>	MC1_ADDR	If IA32_MCG_CAP.CNT >1
407H	1031	IA32_MC1_MISC	MC1_MISC	If IA32_MCG_CAP.CNT >1
408H	1032	IA32_MC2_CTL	MC2_CTL	If IA32_MCG_CAP.CNT >2
409H	1033	IA32_MC2_STATUS	MC2_STATUS	If IA32_MCG_CAP.CNT >2
40AH	1034	IA32_MC2_ADDR <sup>1</sup>	MC2_ADDR	If IA32_MCG_CAP.CNT >2
40BH	1035	IA32_MC2_MISC	MC2_MISC	If IA32_MCG_CAP.CNT >2
40CH	1036	IA32_MC3_CTL	MC3_CTL	If IA32_MCG_CAP.CNT >3
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	If IA32_MCG_CAP.CNT >3
40EH	1038	IA32_MC3_ADDR <sup>1</sup>	MC3_ADDR	If IA32_MCG_CAP.CNT >3
40FH	1039	IA32_MC3_MISC	MC3_MISC	If IA32_MCG_CAP.CNT >3
410H	1040	IA32_MC4_CTL	MC4_CTL	If IA32_MCG_CAP.CNT >4
411H	1041	IA32_MC4_STATUS	MC4_STATUS	If IA32_MCG_CAP.CNT >4
412H	1042	IA32_MC4_ADDR <sup>1</sup>	MC4_ADDR	If IA32_MCG_CAP.CNT >4
413H	1043	IA32_MC4_MISC	MC4_MISC	If IA32_MCG_CAP.CNT >4
414H	1044	IA32_MC5_CTL	MC5_CTL	If IA32_MCG_CAP.CNT >5
415H	1045	IA32_MC5_STATUS	MC5_STATUS	If IA32_MCG_CAP.CNT >5
416H	1046	IA32_MC5_ADDR <sup>1</sup>	MC5_ADDR	If IA32_MCG_CAP.CNT >5
417H	1047	IA32_MC5_MISC	MC5_MISC	If IA32_MCG_CAP.CNT >5
418H	1048	IA32_MC6_CTL	MC6_CTL	If IA32_MCG_CAP.CNT >6
419H	1049	IA32_MC6_STATUS	MC6_STATUS	If IA32_MCG_CAP.CNT >6
41AH	1050	IA32_MC6_ADDR <sup>1</sup>	MC6_ADDR	If IA32_MCG_CAP.CNT >6
41BH	1051	IA32_MC6_MISC	MC6_MISC	If IA32_MCG_CAP.CNT >6
41CH	1052	IA32_MC7_CTL	MC7_CTL	If IA32_MCG_CAP.CNT >7
41DH	1053	IA32_MC7_STATUS	MC7_STATUS	If IA32_MCG_CAP.CNT >7
41EH	1054	IA32_MC7_ADDR <sup>1</sup>	MC7_ADDR	If IA32_MCG_CAP.CNT >7
41FH	1055	IA32_MC7_MISC	MC7_MISC	If IA32_MCG_CAP.CNT >7
420H	1056	IA32_MC8_CTL	MC8_CTL	If IA32_MCG_CAP.CNT >8
421H	1057	IA32_MC8_STATUS	MC8_STATUS	If IA32_MCG_CAP.CNT >8
422H	1058	IA32_MC8_ADDR <sup>1</sup>	MC8_ADDR	If IA32_MCG_CAP.CNT >8
423H	1059	IA32_MC8_MISC	MC8_MISC	If IA32_MCG_CAP.CNT >8
424H	1060	IA32_MC9_CTL	MC9_CTL	If IA32_MCG_CAP.CNT >9
425H	1061	IA32_MC9_STATUS	MC9_STATUS	If IA32_MCG_CAP.CNT >9
426H	1062	IA32_MC9_ADDR <sup>1</sup>	MC9_ADDR	If IA32_MCG_CAP.CNT >9
427H	1063	IA32_MC9_MISC	MC9_MISC	If IA32_MCG_CAP.CNT >9
428H	1064	IA32_MC10_CTL	MC10_CTL	If IA32_MCG_CAP.CNT >10
429H	1065	IA32_MC10_STATUS	MC10_STATUS	If IA32_MCG_CAP.CNT >10

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
42AH	1066	IA32_MC10_ADDR <sup>1</sup>	MC10_ADDR	If IA32_MCG_CAP.CNT >10
42BH	1067	IA32_MC10_MISC	MC10_MISC	If IA32_MCG_CAP.CNT >10
42CH	1068	IA32_MC11_CTL	MC11_CTL	If IA32_MCG_CAP.CNT >11
42DH	1069	IA32_MC11_STATUS	MC11_STATUS	If IA32_MCG_CAP.CNT >11
42EH	1070	IA32_MC11_ADDR <sup>1</sup>	MC11_ADDR	If IA32_MCG_CAP.CNT >11
42FH	1071	IA32_MC11_MISC	MC11_MISC	If IA32_MCG_CAP.CNT >11
430H	1072	IA32_MC12_CTL	MC12_CTL	If IA32_MCG_CAP.CNT >12
431H	1073	IA32_MC12_STATUS	MC12_STATUS	If IA32_MCG_CAP.CNT >12
432H	1074	IA32_MC12_ADDR <sup>1</sup>	MC12_ADDR	If IA32_MCG_CAP.CNT >12
433H	1075	IA32_MC12_MISC	MC12_MISC	If IA32_MCG_CAP.CNT >12
434H	1076	IA32_MC13_CTL	MC13_CTL	If IA32_MCG_CAP.CNT >13
435H	1077	IA32_MC13_STATUS	MC13_STATUS	If IA32_MCG_CAP.CNT >13
436H	1078	IA32_MC13_ADDR <sup>1</sup>	MC13_ADDR	If IA32_MCG_CAP.CNT >13
437H	1079	IA32_MC13_MISC	MC13_MISC	If IA32_MCG_CAP.CNT >13
438H	1080	IA32_MC14_CTL	MC14_CTL	If IA32_MCG_CAP.CNT >14
439H	1081	IA32_MC14_STATUS	MC14_STATUS	If IA32_MCG_CAP.CNT >14
43AH	1082	IA32_MC14_ADDR <sup>1</sup>	MC14_ADDR	If IA32_MCG_CAP.CNT >14
43BH	1083	IA32_MC14_MISC	MC14_MISC	If IA32_MCG_CAP.CNT >14
43CH	1084	IA32_MC15_CTL	MC15_CTL	If IA32_MCG_CAP.CNT >15
43DH	1085	IA32_MC15_STATUS	MC15_STATUS	If IA32_MCG_CAP.CNT >15
43EH	1086	IA32_MC15_ADDR <sup>1</sup>	MC15_ADDR	If IA32_MCG_CAP.CNT >15
43FH	1087	IA32_MC15_MISC	MC15_MISC	If IA32_MCG_CAP.CNT >15
440H	1088	IA32_MC16_CTL	MC16_CTL	If IA32_MCG_CAP.CNT >16
441H	1089	IA32_MC16_STATUS	MC16_STATUS	If IA32_MCG_CAP.CNT >16
442H	1090	IA32_MC16_ADDR <sup>1</sup>	MC16_ADDR	If IA32_MCG_CAP.CNT >16
443H	1091	IA32_MC16_MISC	MC16_MISC	If IA32_MCG_CAP.CNT >16
444H	1092	IA32_MC17_CTL	MC17_CTL	If IA32_MCG_CAP.CNT >17
445H	1093	IA32_MC17_STATUS	MC17_STATUS	If IA32_MCG_CAP.CNT >17
446H	1094	IA32_MC17_ADDR <sup>1</sup>	MC17_ADDR	If IA32_MCG_CAP.CNT >17
447H	1095	IA32_MC17_MISC	MC17_MISC	If IA32_MCG_CAP.CNT >17
448H	1096	IA32_MC18_CTL	MC18_CTL	If IA32_MCG_CAP.CNT >18
449H	1097	IA32_MC18_STATUS	MC18_STATUS	If IA32_MCG_CAP.CNT >18
44AH	1098	IA32_MC18_ADDR <sup>1</sup>	MC18_ADDR	If IA32_MCG_CAP.CNT >18
44BH	1099	IA32_MC18_MISC	MC18_MISC	If IA32_MCG_CAP.CNT >18
44CH	1100	IA32_MC19_CTL	MC19_CTL	If IA32_MCG_CAP.CNT >19
44DH	1101	IA32_MC19_STATUS	MC19_STATUS	If IA32_MCG_CAP.CNT >19
44EH	1102	IA32_MC19_ADDR <sup>1</sup>	MC19_ADDR	If IA32_MCG_CAP.CNT >19

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
44FH	1103	IA32_MC19_MISC	MC19_MISC	If IA32_MCG_CAP.CNT >19
450H	1104	IA32_MC20_CTL	MC20_CTL	If IA32_MCG_CAP.CNT >20
451H	1105	IA32_MC20_STATUS	MC20_STATUS	If IA32_MCG_CAP.CNT >20
452H	1106	IA32_MC20_ADDR <sup>1</sup>	MC20_ADDR	If IA32_MCG_CAP.CNT >20
453H	1107	IA32_MC20_MISC	MC20_MISC	If IA32_MCG_CAP.CNT >20
454H	1108	IA32_MC21_CTL	MC21_CTL	If IA32_MCG_CAP.CNT >21
455H	1109	IA32_MC21_STATUS	MC21_STATUS	If IA32_MCG_CAP.CNT >21
456H	1110	IA32_MC21_ADDR <sup>1</sup>	MC21_ADDR	If IA32_MCG_CAP.CNT >21
457H	1111	IA32_MC21_MISC	MC21_MISC	If IA32_MCG_CAP.CNT >21
458H	1112	IA32_MC22_CTL	MC22_CTL	If IA32_MCG_CAP.CNT >22
459H	1113	IA32_MC22_STATUS	MC22_STATUS	If IA32_MCG_CAP.CNT >22
45AH	1114	IA32_MC22_ADDR <sup>1</sup>	MC22_ADDR	If IA32_MCG_CAP.CNT >22
45BH	1115	IA32_MC22_MISC	MC22_MISC	If IA32_MCG_CAP.CNT >22
45CH	1116	IA32_MC23_CTL	MC23_CTL	If IA32_MCG_CAP.CNT >23
45DH	1117	IA32_MC23_STATUS	MC23_STATUS	If IA32_MCG_CAP.CNT >23
45EH	1118	IA32_MC23_ADDR <sup>1</sup>	MC23_ADDR	If IA32_MCG_CAP.CNT >23
45FH	1119	IA32_MC23_MISC	MC23_MISC	If IA32_MCG_CAP.CNT >23
460H	1120	IA32_MC24_CTL	MC24_CTL	If IA32_MCG_CAP.CNT >24
461H	1121	IA32_MC24_STATUS	MC24_STATUS	If IA32_MCG_CAP.CNT >24
462H	1122	IA32_MC24_ADDR <sup>1</sup>	MC24_ADDR	If IA32_MCG_CAP.CNT >24
463H	1123	IA32_MC24_MISC	MC24_MISC	If IA32_MCG_CAP.CNT >24
464H	1124	IA32_MC25_CTL	MC25_CTL	If IA32_MCG_CAP.CNT >25
465H	1125	IA32_MC25_STATUS	MC25_STATUS	If IA32_MCG_CAP.CNT >25
466H	1126	IA32_MC25_ADDR <sup>1</sup>	MC25_ADDR	If IA32_MCG_CAP.CNT >25
467H	1127	IA32_MC25_MISC	MC25_MISC	If IA32_MCG_CAP.CNT >25
468H	1128	IA32_MC26_CTL	MC26_CTL	If IA32_MCG_CAP.CNT >26
469H	1129	IA32_MC26_STATUS	MC26_STATUS	If IA32_MCG_CAP.CNT >26
46AH	1130	IA32_MC26_ADDR <sup>1</sup>	MC26_ADDR	If IA32_MCG_CAP.CNT >26
46BH	1131	IA32_MC26_MISC	MC26_MISC	If IA32_MCG_CAP.CNT >26
46CH	1132	IA32_MC27_CTL	MC27_CTL	If IA32_MCG_CAP.CNT >27
46DH	1133	IA32_MC27_STATUS	MC27_STATUS	If IA32_MCG_CAP.CNT >27
46EH	1134	IA32_MC27_ADDR <sup>1</sup>	MC27_ADDR	If IA32_MCG_CAP.CNT >27
46FH	1135	IA32_MC27_MISC	MC27_MISC	If IA32_MCG_CAP.CNT >27
470H	1136	IA32_MC28_CTL	MC28_CTL	If IA32_MCG_CAP.CNT >28
471H	1137	IA32_MC28_STATUS	MC28_STATUS	If IA32_MCG_CAP.CNT >28
472H	1138	IA32_MC28_ADDR <sup>1</sup>	MC28_ADDR	If IA32_MCG_CAP.CNT >28
473H	1139	IA32_MC28_MISC	MC28_MISC	If IA32_MCG_CAP.CNT >28

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
480H	1152	IA32_VMX_BASIC	Reporting Register of Basic VMX Capabilities (R/O) See Appendix A.1, "Basic VMX Information."	If CPUID.01H:ECX.[5] = 1
481H	1153	IA32_VMX_PINBASED_CTL	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
482H	1154	IA32_VMX_PROCBASED_CTL	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
483H	1155	IA32_VMX_EXIT_CTL	Capability Reporting Register of VM-Exit Controls (R/O) See Appendix A.4, "VM-Exit Controls."	If CPUID.01H:ECX.[5] = 1
484H	1156	IA32_VMX_ENTRY_CTL	Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, "VM-Entry Controls."	If CPUID.01H:ECX.[5] = 1
485H	1157	IA32_VMX_MISC	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Appendix A.6, "Miscellaneous Data."	If CPUID.01H:ECX.[5] = 1
486H	1158	IA32_VMX_CR0_FIXED0	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
487H	1159	IA32_VMX_CR0_FIXED1	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
488H	1160	IA32_VMX_CR4_FIXED0	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
489H	1161	IA32_VMX_CR4_FIXED1	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
48AH	1162	IA32_VMX_VMCS_ENUM	Capability Reporting Register of VMCS Field Enumeration (R/O) See Appendix A.9, "VMCS Enumeration."	If CPUID.01H:ECX.[5] = 1
48BH	1163	IA32_VMX_PROCBASED_CTL2	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3.3, "Secondary Processor-Based VM-Execution Controls."	If ( CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTL[63])



**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
48CH	1164	IA32_VMX_EPT_VPID_CAP	Capability Reporting Register of EPT and VPID (R/O) See Appendix A.10, "VPID and EPT Capabilities."	If ( CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_C TLS[63] && ( IA32_VMX_PROCBASED_C TLS2[33]    IA32_VMX_PROCBASED_C TLS2[37]) )
48DH	1165	IA32_VMX_TRUE_PINBASED_CTL	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL	Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
48FH	1167	IA32_VMX_TRUE_EXIT_CTL	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Appendix A.4, "VM-Exit Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
490H	1168	IA32_VMX_TRUE_ENTRY_CTL	Capability Reporting Register of VM-Entry Flex Controls (R/O) See Appendix A.5, "VM-Entry Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
491H	1169	IA32_VMX_VMFUNC	Capability Reporting Register of VM-Function Controls (R/O)	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
4C1H	1217	IA32_A_PMC0	Full Width Writable IA32_PMC0 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 0) && IA32_PERF_CAPABILITIES[13] = 1
4C2H	1218	IA32_A_PMC1	Full Width Writable IA32_PMC1 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 1) && IA32_PERF_CAPABILITIES[13] = 1
4C3H	1219	IA32_A_PMC2	Full Width Writable IA32_PMC2 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 2) && IA32_PERF_CAPABILITIES[13] = 1
4C4H	1220	IA32_A_PMC3	Full Width Writable IA32_PMC3 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 3) && IA32_PERF_CAPABILITIES[13] = 1
4C5H	1221	IA32_A_PMC4	Full Width Writable IA32_PMC4 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 4) && IA32_PERF_CAPABILITIES[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4C6H	1222	IA32_A_PMC5	Full Width Writable IA32_PMC5 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 5) && IA32_PERF_CAPABILITIES[13] = 1
4C7H	1223	IA32_A_PMC6	Full Width Writable IA32_PMC6 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 6) && IA32_PERF_CAPABILITIES[13] = 1
4C8H	1224	IA32_A_PMC7	Full Width Writable IA32_PMC7 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 7) && IA32_PERF_CAPABILITIES[13] = 1
4D0H	1232	IA32_MCG_EXT_CTL	Allows software to signal some MCEs to only a single logical processor in the system. (R/W) See Section 15.3.1.4, "IA32_MCG_EXT_CTL MSR".	If IA32_MCG_CAP.LMCE_P = 1
		0	LMCE_EN	
		63:1	Reserved	
500H	1280	IA32_SGX_SVN_STATUS	Status and SVN Threshold of SGX Support for ACM (RO).	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		0	Lock	See Section 37.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1	Reserved	
		23:16	SGX_SVN_SINIT	See Section 37.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		63:24	Reserved	
560H	1376	IA32_RTIT_OUTPUT_BASE	Trace Output Base Register (R/W)	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H, ECX=0):ECX[0] = 1)    (CPUID.(EAX=14H, ECX=0):ECX[2] = 1)))
		6:0	Reserved	
		MAXPHYADDR <sup>3</sup> -1:7	Base physical address.	
		63:MAXPHYADDR	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Trace Output Mask Pointers Register (R/W)	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && (CPUID.(EAX=14H, ECX=0):ECX[0] = 1)    (CPUID.(EAX=14H, ECX=0):ECX[2] = 1))
		6:0	Reserved	
		31:7	MaskOrTableOffset	
		63:32	Output Offset	
570H	1392	IA32_RTIT_CTL	Trace Control Register (R/W)	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	TraceEn	
		1	CYCEn	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		2	OS	
		3	User	
		4	PwrEvtEn	If (CPUID.(EAX=07H, ECX=1):EBX[5] = 1)
		5	FUPonPTW	If (CPUID.(EAX=07H, ECX=1):EBX[4] = 1)
		6	FabricEn	If (CPUID.(EAX=07H, ECX=0):ECX[3] = 1)
		7	CR3 filter	
		8	ToPA	
		9	MTCEn	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		10	TSCEn	
		11	DisRETC	
		12	PTWEn	If (CPUID.(EAX=07H, ECX=1):EBX[4] = 1)
		13	BranchEn	
		17:14	MTCFreq	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		18	Reserved, must be zero.	
		22:19	CYCThresh	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		23	Reserved, must be zero.	
		27:24	PSBFreq	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
31:28	Reserved, must be zero.			
35:32	ADDR0_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)		

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		39:36	ADDR1_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		43:40	ADDR2_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:44	ADDR3_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		55:48	Reserved, must be zero.	
		56	InjectPsbPmiOnEnable	If (CPUID.(EAX=07H, ECX=1):EBX[6] = 1)
		63:57	Reserved, must be zero.	
571H	1393	IA32_RTIT_STATUS	Tracing Status Register (R/W)	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	FilterEn (writes ignored)	If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)
		1	ContexEn (writes ignored)	
		2	TriggerEn (writes ignored)	
		3	Reserved	
		4	Error	
		5	Stopped	
		6	PendPSB	If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)
		7	PendToPAPMI	If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)
		31:8	Reserved, must be zero.	
		48:32	PacketByteCnt	If (CPUID.(EAX=07H, ECX=0):EBX[1] > 3)
		63:49	Reserved	
572H	1394	IA32_RTIT_CR3_MATCH	Trace Filter CR3 Match Register (R/W)	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		4:0	Reserved	
		63:5	CR3[63:5] value to match.	
580H	1408	IA32_RTIT_ADDR0_A	Region 0 Start Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
581H	1409	IA32_RTIT_ADDR0_B	Region 0 End Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
582H	1410	IA32_RTIT_ADDR1_A	Region 1 Start Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		47:0	Virtual Address	
		63:48	SignExt_VA	
583H	1411	IA32_RTIT_ADDR1_B	Region 1 End Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	
584H	1412	IA32_RTIT_ADDR2_A	Region 2 Start Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
585H	1413	IA32_RTIT_ADDR2_B	Region 2 End Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
586H	1414	IA32_RTIT_ADDR3_A	Region 3 Start Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
587H	1415	IA32_RTIT_ADDR3_B	Region 3 End Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
600H	1536	IA32_DS_AREA	DS Save Area (R/W) Points to the linear address of the first byte of the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.6.3.4, "Debug Store (DS) Mechanism."	If (CPUID.01H:EDX.DS[21]=1)
		63:0	The linear address of the first byte of the DS buffer management area, if IA-32e mode is active.	
		31:0	The linear address of the first byte of the DS buffer management area, if not in IA-32e mode.	
		63:32	Reserved if not in IA-32e mode.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
6A0H	1696	IA32_U_CET	Configure User Mode CET (R/W)	Bits 1:0 are defined if CPUID.(EAX=07H, ECX=0H);ECX.CET_SS[07] = 1. Bits 5:2 and bits 63:10 are defined if CPUID.(EAX=07H, ECX=0H);EDX.CET_IBT[20] = 1.
		0	SH_STK_EN: When set to 1, enable shadow stacks at CPL3.	
		1	WR_SHSTK_EN: When set to 1, enables the WRSSD/WRSSQ instructions.	
		2	ENDBR_EN: When set to 1, enables indirect branch tracking.	
		3	LEG_IW_EN: Enable legacy compatibility treatment for indirect branch tracking.	
		4	NO_TRACK_EN: When set to 1, enables use of no-track prefix for indirect branch tracking.	
		5	SUPPRESS_DIS: When set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.	
		9:6	Reserved; must be zero.	
		10	SUPPRESS: When set to 1, indirect branch tracking is suppressed. This bit can be written to 1 only if TRACKER is written as IDLE.	
		11	TRACKER: Value of the indirect branch tracking state machine. Values: IDLE (0), WAIT_FOR_ENDBRANCH(1).	
		63:12	EB_LEG_BITMAP_BASE: Linear address bits 63:12 of a legacy code page bitmap used for legacy compatibility when indirect branch tracking is enabled.  If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	
6A2H	1698	IA32_S_CET	Configure Supervisor Mode CET (R/W)	See IA32_U_CET (6A0H) for reference; similar format.

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
6A4H	1700	IA32_PL0_SSP	Linear address to be loaded into SSP on transition to privilege level 0. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6A5H	1701	IA32_PL1_SSP	Linear address to be loaded into SSP on transition to privilege level 1. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6A6H	1702	IA32_PL2_SSP	Linear address to be loaded into SSP on transition to privilege level 2. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6A7H	1703	IA32_PL3_SSP	Linear address to be loaded into SSP on transition to privilege level 3. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
6A8H	1704	IA32_INTERRUPT_SSP_TABLE_ADDR	Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W)  This MSR is not present on processors that do not support Intel 64 architecture. This field cannot represent a non-canonical address.	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6E0H	1760	IA32_TSC_DEADLINE	TSC Target of Local APIC's TSC Deadline Mode (R/W)	If CPUID.01H:ECX.[24] = 1
6E1H	1761	IA32_PKRS	Specifies the PK permissions associated with each protection domain for supervisor pages (R/W)	If CPUID.(EAX=07H, ECX=0H):ECX.PKS [31] = 1
		31:0	For domain i (i between 0 and 15), bits 2i and 2i+1 contain the AD and WD permissions, respectively.	
		63:32	Reserved.	
770H	1904	IA32_PM_ENABLE	Enable/disable HWP (R/W)	If CPUID.06H:EAX.[7] = 1
		0	HWP_ENABLE (R/W1-Once) See Section 14.4.2, "Enabling HWP".	If CPUID.06H:EAX.[7] = 1
		63:1	Reserved	
771H	1905	IA32_HWP_CAPABILITIES	HWP Performance Range Enumeration (RO)	If CPUID.06H:EAX.[7] = 1
		7:0	Highest_Performance See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities".	If CPUID.06H:EAX.[7] = 1
		15:8	Guaranteed_Performance See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities".	If CPUID.06H:EAX.[7] = 1
		23:16	Most_Efficient_Performance See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities".	If CPUID.06H:EAX.[7] = 1
		31:24	Lowest_Performance See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities".	If CPUID.06H:EAX.[7] = 1
		63:32	Reserved	
772H	1906	IA32_HWP_REQUEST_PKG	Power Management Control Hints for All Logical Processors in a Package (R/W)	If CPUID.06H:EAX.[11] = 1
		7:0	Minimum_Performance See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[11] = 1
		15:8	Maximum_Performance See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[11] = 1
		23:16	Desired_Performance See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[11] = 1



**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		31:24	Energy_Performance_Preference See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	Activity_Window See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[9] = 1
		63:42	Reserved	
773H	1907	IA32_HWP_INTERRUPT	Control HWP Native Interrupts (R/W)	If CPUID.06H:EAX.[8] = 1
		0	EN_Guaranteed_Performance_Change See Section 14.4.6, "HWP Notifications".	If CPUID.06H:EAX.[8] = 1
		1	EN_Excursion_Minimum See Section 14.4.6, "HWP Notifications".	If CPUID.06H:EAX.[8] = 1
		63:2	Reserved	
774H	1908	IA32_HWP_REQUEST	Power Management Control Hints to a Logical Processor (R/W)	If CPUID.06H:EAX.[7] = 1
		7:0	Minimum_Performance See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[7] = 1
		15:8	Maximum_Performance See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[7] = 1
		23:16	Desired_Performance See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[7] = 1
		31:24	Energy_Performance_Preference See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	Activity_Window See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[9] = 1
		42	Package_Control See Section 14.4.4, "Managing HWP".	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[11] = 1
		63:43	Reserved	
775H	1909	IA32_PECI_HWP_REQUEST_INFO	IA32_PECI_HWP_REQUEST_INFO	
		7:0	Minimum Performance (MINIMUM_PERFORMANCE): Used by OS to read the latest value of Peci minimum performance input. Default value is 0.	
		15:8	Maximum Performance (MAXIMUM_PERFORMANCE): Used by OS to read the latest value of Peci maximum performance input. Default value is 0.	
		23:16	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		31:24	Energy Performance Preference (ENERGY_PERFORMANCE_PREFERENCE): Used by OS to read the latest value of PECC Energy Performance Preference input. Default value is 0.	
		59:32	Reserved.	
		60	EPP PECC Override (EPP_PECC_OVERRIDE): Indicates whether PECC is currently overriding the Energy Performance Preference input. If set to '1', PECC is overriding the Energy Performance Preference input. If clear (0), OS has control over Energy Performance Preference input. Default value is 0.	
		61	Reserved.	
		62	Max PECC Override (MAX_PECC_OVERRIDE): Indicates whether PECC is currently overriding the Maximum Performance input. If set to '1', PECC is overriding the Maximum Performance input. If clear (0), OS has control over Maximum Performance input. Default value is 0.	
		63	Min PECC Override (MIN_PECC_OVERRIDE): Indicates whether PECC is currently overriding the Minimum Performance input. If set to '1', PECC is overriding the Minimum Performance input. If clear (0), OS has control over Minimum Performance input. Default value is 0.	
777H	1911	IA32_HWP_STATUS	Log bits indicating changes to Guaranteed & excursions to Minimum (R/W)	If CPUID.06H:EAX.[7] = 1
		0	Guaranteed_Performance_Change (R/WCO) See Section 14.4.5, "HWP Feedback".	If CPUID.06H:EAX.[7] = 1
		1	Reserved	
		2	Excursion_To_Minimum (R/WCO) See Section 14.4.5, "HWP Feedback".	If CPUID.06H:EAX.[7] = 1
		63:3	Reserved	
802H	2050	IA32_X2APIC_APICID	x2APIC ID Register (R/O) See x2APIC Specification.	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
803H	2051	IA32_X2APIC_VERSION	x2APIC Version Register (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
808H	2056	IA32_X2APIC_TPR	x2APIC Task Priority Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
80AH	2058	IA32_X2APIC_PPR	x2APIC Processor Priority Register (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80BH	2059	IA32_X2APIC_EOI	x2APIC EOI Register (W/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80DH	2061	IA32_X2APIC_LDR	x2APIC Logical Destination Register (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80FH	2063	IA32_X2APIC_SIVR	x2APIC Spurious Interrupt Vector Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
810H	2064	IA32_X2APIC_ISR0	x2APIC In-Service Register Bits 31:0 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
811H	2065	IA32_X2APIC_ISR1	x2APIC In-Service Register Bits 63:32 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
812H	2066	IA32_X2APIC_ISR2	x2APIC In-Service Register Bits 95:64 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
813H	2067	IA32_X2APIC_ISR3	x2APIC In-Service Register Bits 127:96 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
814H	2068	IA32_X2APIC_ISR4	x2APIC In-Service Register Bits 159:128 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
815H	2069	IA32_X2APIC_ISR5	x2APIC In-Service Register Bits 191:160 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
816H	2070	IA32_X2APIC_ISR6	x2APIC In-Service Register Bits 223:192 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
817H	2071	IA32_X2APIC_ISR7	x2APIC In-Service Register Bits 255:224 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
818H	2072	IA32_X2APIC_TMR0	x2APIC Trigger Mode Register Bits 31:0 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
819H	2073	IA32_X2APIC_TMR1	x2APIC Trigger Mode Register Bits 63:32 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81AH	2074	IA32_X2APIC_TMR2	x2APIC Trigger Mode Register Bits 95:64 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
81BH	2075	IA32_X2APIC_TMR3	x2APIC Trigger Mode Register Bits 127:96 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81CH	2076	IA32_X2APIC_TMR4	x2APIC Trigger Mode Register Bits 159:128 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81DH	2077	IA32_X2APIC_TMR5	x2APIC Trigger Mode Register Bits 191:160 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81EH	2078	IA32_X2APIC_TMR6	x2APIC Trigger Mode Register Bits 223:192 (R/O)	If ( CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1)
81FH	2079	IA32_X2APIC_TMR7	x2APIC Trigger Mode Register Bits 255:224 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
820H	2080	IA32_X2APIC_IRR0	x2APIC Interrupt Request Register Bits 31:0 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
821H	2081	IA32_X2APIC_IRR1	x2APIC Interrupt Request Register Bits 63:32 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
822H	2082	IA32_X2APIC_IRR2	x2APIC Interrupt Request Register Bits 95:64 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
823H	2083	IA32_X2APIC_IRR3	x2APIC Interrupt Request Register Bits 127:96 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
824H	2084	IA32_X2APIC_IRR4	x2APIC Interrupt Request Register Bits 159:128 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
825H	2085	IA32_X2APIC_IRR5	x2APIC Interrupt Request Register Bits 191:160 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
826H	2086	IA32_X2APIC_IRR6	x2APIC Interrupt Request Register Bits 223:192 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
827H	2087	IA32_X2APIC_IRR7	x2APIC Interrupt Request Register Bits 255:224 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
828H	2088	IA32_X2APIC_ESR	x2APIC Error Status Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
82FH	2095	IA32_X2APIC_LVT_CMCI	x2APIC LVT Corrected Machine Check Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
830H	2096	IA32_X2APIC_ICR	x2APIC Interrupt Command Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
832H	2098	IA32_X2APIC_LVT_TIMER	x2APIC LVT Timer Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
833H	2099	IA32_X2APIC_LVT_THERMAL	x2APIC LVT Thermal Sensor Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
834H	2100	IA32_X2APIC_LVT_PMI	x2APIC LVT Performance Monitor Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
835H	2101	IA32_X2APIC_LVT_LINT0	x2APIC LVT LINT0 Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
836H	2102	IA32_X2APIC_LVT_LINT1	x2APIC LVT LINT1 Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
837H	2103	IA32_X2APIC_LVT_ERROR	x2APIC LVT Error Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
838H	2104	IA32_X2APIC_INIT_COUNT	x2APIC Initial Count Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
839H	2105	IA32_X2APIC_CUR_COUNT	x2APIC Current Count Register (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83EH	2110	IA32_X2APIC_DIV_CONF	x2APIC Divide Configuration Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83FH	2111	IA32_X2APIC_SELF_IPI	x2APIC Self IPI Register (W/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
981H	2433	IA32_TME_CAPABILITY	Memory Encryption Capability MSR	If CPUID.07H:ECX.[13] = 1
		0	Support for AES-XTS 128-bit encryption algorithm. (NIST standard)	
		30:1	Reserved.	
		31	TME encryption bypass supported.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		35:32	MK_TME_MAX_KEYID_BITS Number of bits which can be allocated for usage as key identifiers for multi-key memory encryption. 4 bits allow for a maximum value of 15, which could address 32K keys. Zero if MKTME is not supported.	
		50:36	MK_TME_MAX_KEYS Indicates the maximum number of keys which are available for usage. This value may not be a power of 2. KeyID 0 is specially reserved and is not accounted for in this field.	
		63:51	Reserved.	
982H	2434	IA32_TME_ACTIVATE	Memory Encryption Activation MSR This MSR is used to lock the MSRs listed below. Any write to the following MSRs will be ignored after they are locked. The lock is reset when CPU is reset. <ul style="list-style-type: none"> <li>▪ IA32_TME_ACTIVATE</li> <li>▪ IA32_TME_EXCLUDE_MASK</li> <li>▪ IA32_TME_EXCLUDE_BASE</li> </ul> Note that IA32_TME_EXCLUDE_MASK and IA32_TME_EXCLUDE_BASE must be configured before IA32_TME_ACTIVATE.	If CPUID.07H:ECX.[13] = 1
		0	Lock RO - Will be set upon successful WRMSR (or first SMI); written value ignored.	
		1	Hardware Encryption Enable This bit also enables MKTME; MKTME cannot be enabled without enabling encryption hardware.	
		2	Key Select 0: Create a new TME key (expected cold/warm boot). 1: Restore the TME key from storage (Expected when resume from standby).	
		3	Save TME Key for Standby Save key into storage to be used when resume from standby. Note: This may not be supported in all processors.	

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:4	TME Policy/Encryption Algorithm Only algorithms enumerated in IA32_TME_CAPABILITY are allowed. For example: 0000 – AES-XTS-128. Other values are invalid.	
		30:8	Reserved.	
		31	TME Encryption Bypass Enable When encryption hardware is enabled: <ul style="list-style-type: none"> <li>▪ Total Memory Encryption is enabled using a CPU generated ephemeral key based on a hardware random number generator when this bit is set to 0.</li> <li>▪ Total Memory Encryption is bypassed (no encryption/decryption for KeyID0) when this bit is set to 1.</li> </ul> Software must inspect Hardware Encryption Enable (bit 1) and TME encryption bypass Enable (bit 31) to determine if TME encryption is enabled.	
		35:32	MK_TME_KEYID_BITS Reserved if MKTME is not enumerated, otherwise: The number of key identifier bits to allocate to MKTME usage. Similar to enumeration, this is an encoded value. Writing a value greater than MK_TME_MAX_KEYID_BITS will result in #GP. Writing a non-zero value to this field will #GP if bit 1 of EAX (Hardware Encryption Enable) is not also set to '1, as encryption hardware must be enabled to use MKTME. Example: To support 255 keys, this field would be set to a value of 8.	
		47:36	Reserved.	
		63:48	MK_TME_CRYPTO_ALGS Reserved if MKTME is not enumerated, otherwise: Bit 48: AES-XTS 128 Bit 63:49: Reserved (#GP) Bitmask for BIOS to set which encryption algorithms are allowed for MKTME, would be later enforced by the key loading ISA ('1 = allowed).	
		983H	2435	IA32_TME_EXCLUDE_MASK
		10:0	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11	Enable: When set to '1', then TME_EXCLUDE_BASE and TME_EXCLUDE_MASK are used to define an exclusion region for TME/MKTME (for KeyID=0).	
		MAXPHYSADDR-1:12	TMEEMASK: This field indicates the bits that must match TMEEBASE in order to qualify as a TME/MKTME (for KeyID=0) exclusion memory range access.	
		63:MAXPHYSADDR	Reserved; must be zero.	
984H	2436	IA32_TME_EXCLUDE_BASE	Memory Encryption Exclude Base	If CPUID.07H:ECX.[13] = 1
		11:0	Reserved.	
		MAXPHYSADDR-1:12	TMEEBASE: Base physical address to be excluded for TME/MKTME (for KeyID=0) encryption.	
		63:MAXPHYSADDR	Reserved; must be zero.	
990H	2448	IA32_COPY_STATUS <sup>4</sup>	Status of Most Recent Platform to Local or Local to Platform Copies (R/O)	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] = 1))
		0	lwKEY_COPY_SUCCESSFUL: Status of most recent copy to or from lwKeyBackup	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] = 1))
		63:1	Reserved	
991H	2449	IA32_lwKEYBACKUP_STATUS <sup>4</sup>	Information about lwKeyBackup Register (R/O)	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] = 1))
		0	Backup/Restore Valid Cleared when a write to lwKeyBackup is initiated, and then set when the latest write of lwKeyBackup has been written to storage that persists across S3/S4 sleep state. If S3/S4 is entered between when an lwKeyBackup write occurs and when this bit is set, then lwKeyBackup may not be recovered after S3/S4 exit. During S3/S4 sleep state exit (system wake up), this bit is cleared. It is set again when lwKeyBackup is restored from persistent storage and thus available to be copied to lwKey using IA32_COPY_PLATFORM_TO_LOCAL MSR. Another write to lwKeyBackup (via IA32_COPY_LOCAL_TO_PLATFORM MSR) may fail if a previous write has not yet set this bit.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] = 1))
		1	Reserved	



**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		2	Backup Key Storage Read/Write Error Updated prior to backup/restore valid being set. Set when an error is encountered while backing up or restoring a key to persistent storage.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0):ECX[23] = 1))
		3	lwKeyBackup Consumed Set after the previous backup operation has been consumed by the platform. This does not indicate that the system is ready for a second lwKeyBackup write as the previous lwKeyBackup write may still need to set Backup/restore valid.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0):ECX[23] = 1))
		63:4	Reserved	
C80H	3200	IA32_DEBUG_INTERFACE	Silicon Debug Feature Control (R/W)	If CPUID.01H:ECX.[11] = 1
		0	Enable (R/W) BIOS set 1 to enable Silicon debug features. Default is 0.	If CPUID.01H:ECX.[11] = 1
		29:1	Reserved	
		30	Lock (R/W): If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.	If CPUID.01H:ECX.[11] = 1
		31	Debug Occurred (R/O): This “sticky bit” is set by hardware to indicate the status of bit 0. Default is 0.	If CPUID.01H:ECX.[11] = 1
		63:32	Reserved	
C81H	3201	IA32_L3_QOS_CFG	L3 QOS Configuration (R/W)	If ( CPUID.(EAX=10H, ECX=1):ECX.[2] = 1 )
		0	Enable (R/W) Set 1 to enable L3 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode.	
		63:1	Reserved. Attempts to write to reserved bits result in a #GP(0).	
C82H	3202	IA32_L2_QOS_CFG	L2 QOS Configuration (R/W)	If ( CPUID.(EAX=10H, ECX=2):ECX.[2] = 1 )
		0	Enable (R/W) Set 1 to enable L2 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode.	
		63:1	Reserved. Attempts to write to reserved bits result in a #GP(0).	
C8DH	3213	IA32_QM_EVTSEL	Monitoring Event Select Register (R/W)	If ( CPUID.(EAX=07H, ECX=0):EBX.[12] = 1 )

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:0	Event ID: ID of a supported monitoring event to report via IA32_QM_CTR.	
		31:8	Reserved	
		N+31:32	Resource Monitoring ID: ID for monitoring hardware to report monitored data via IA32_QM_CTR.	N = Ceil (Log <sub>2</sub> ( CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] + 1))
		63:N+32	Reserved	
C8EH	3214	IA32_QM_CTR	Monitoring Counter Register (R/O)	If ( CPUID.(EAX=07H, ECX=0);EBX.[12] = 1 )
		61:0	Resource Monitored Data	
		62	Unavailable: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.	
		63	Error: If 1, indicates an unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	
C8FH	3215	IA32_PQR_ASSOC	Resource Association Register (R/W)	If ( (CPUID.(EAX=07H, ECX=0);EBX[12] = 1) or (CPUID.(EAX=07H, ECX=0);EBX[15] = 1) )
		N-1:0	Resource Monitoring ID (R/W): ID for monitoring hardware to track internal operation, e.g., memory access.	N = Ceil (Log <sub>2</sub> ( CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] + 1))
		31:N	Reserved	
		63:32	COS (R/W): The class of service (COS) to enforce (on writes); returns the current COS when read.	If ( CPUID.(EAX=07H, ECX=0);EBX.[15] = 1 )
C90H - D8FH	3216 - 3471	Reserved MSR Address Space for CAT Mask Registers	See Section 17.19.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	
C90H	3216	IA32_L3_MASK_0	L3 CAT Mask for COS0 (R/W)	If (CPUID.(EAX=10H, ECX=0H);EBX[1] != 0)
		31:0	Capacity Bit Mask (R/W)	
		63:32	Reserved	
C90H+n	3216+n	IA32_L3_MASK_n	L3 CAT Mask for COSn (R/W)	n = CPUID.(EAX=10H, ECX=1H);EDX[15:0]
		31:0	Capacity Bit Mask (R/W)	
		63:32	Reserved	
D10H - D4FH	3344 - 3407	Reserved MSR Address Space for L2 CAT Mask Registers	See Section 17.19.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
D10H	3344	IA32_L2_MASK_0	L2 CAT Mask for COS0 (R/W)	If (CPUID.(EAX=10H, ECX=0H);EBX[2] != 0)
		31:0	Capacity Bit Mask (R/W)	
		63:32	Reserved	
D10H+n	3344+n	IA32_L2_MASK_n	L2 CAT Mask for COSn (R/W)	n = CPUID.(EAX=10H, ECX=2H);EDX[15:0]
		31:0	Capacity Bit Mask (R/W)	
		63:32	Reserved	
D90H	3472	IA32_BNDCFGS	Supervisor State of MPX Configuration (R/W)	If (CPUID.(EAX=07H, ECX=0H);EBX[14] = 1)
		0	EN: Enable Intel MPX in supervisor mode.	
		1	BNDPRESERVE: Preserve the bounds registers for near branch instructions in the absence of the BND prefix.	
		11:2	Reserved, must be zero.	
		63:12	Base Address of Bound Directory.	
D91H	3473	IA32_COPY_LOCAL_TO_PLATFORM <sup>4</sup>	Copy Local State to Platform State (w)	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H);ECX[23] = 1))
		0	lWKeyBackup Copy lWKey to lWKeyBackup	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H);ECX[23] = 1))
		63:1	Reserved	
D92H	3474	IA32_COPY_PLATFORM_TO_LOCAL <sup>4</sup>	Copy Platform State to Local State (w)	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H);ECX[23] = 1))
		0	lWKeyBackup Copy lWKeyBackup to lWKey	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H);ECX[23] = 1))
		63:1	Reserved	
DA0H	3488	IA32_XSS	Extended Supervisor State Mask (R/W)	If (CPUID.(ODH, 1);EAX.[3] = 1)
		7:0	Reserved.	
		8	Trace Packet Configuration State (R/W)	
		10:9	Reserved.	
		11	CET_U State (R/W)	
		12	CET_S State (R/W)	
		13	HDC State (R/W)	
63:14	Reserved.			

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
DB0H	3504	IA32_PKG_HDC_CTL	Package Level Enable/disable HDC (R/W)	If CPUID.06H:EAX.[13] = 1
		0	HDC_Pkg_Enable (R/W) Force HDC idling or wake up HDC-idled logical processors in the package. See Section 14.5.2, "Package level Enabling HDC".	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved	
DB1H	3505	IA32_PM_CTL1	Enable/disable HWP (R/W)	If CPUID.06H:EAX.[13] = 1
		0	HDC_Allow_Block (R/W) Allow/Block this logical processor for package level HDC control. See Section 14.5.3.	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved	
DB2H	3506	IA32_THREAD_STALL	Per-Logical_Processor HDC Idle Residency (R/O)	If CPUID.06H:EAX.[13] = 1
		63:0	Stall_Cycle_Cnt (R/W) Stalled cycles due to HDC forced idle on this logical processor. See Section 14.5.4.1.	If CPUID.06H:EAX.[13] = 1
17D0H	6096	IA32_HW_FEEDBACK_PTR	Hardware Feedback Interface Pointer	If CPUID.06H:EAX.[19] = 1
		0	Valid (R/W) When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.	
		11:1	Reserved	
		(MAXPHYADDR-1):12	ADDR (R/W) Physical address of the page frame of the first page of the hardware feedback interface structure.	
		63:MAXPHYADDR	Reserved	
17D1H	6097	IA32_HW_FEEDBACK_CONFIG	Hardware Feedback Interface Configuration	If CPUID.06H:EAX.[19] = 1
		0	Enable (R/W) When set to 1, enables the hardware feedback interface.	
		63:1	Reserved	
4000_0000H - 4000_00FFH		Reserved MSR Address Space	All existing and future processors will not implement MSRs in this range.	
C000_0080H		IA32_EFER	Extended Feature Enables	If ( CPUID.80000001H:EDX.[20]    CPUID.80000001H:EDX.[29] )

**Table 2-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	SYSCALL Enable: IA32_EFER.SCE (R/W) Enables SYSCALL/SYSRET instructions in 64-bit mode.	
		7:1	Reserved	
		8	IA-32e Mode Enable: IA32_EFER.LME (R/W) Enables IA-32e mode operation.	
		9	Reserved	
		10	IA-32e Mode Active: IA32_EFER.LMA (R) Indicates IA-32e mode is active when set.	
		11	Execute Disable Bit Enable: IA32_EFER.NXE (R/W)	
		63:12	Reserved	
C000_0081H		IA32_STAR	System Call Target Address (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0082H		IA32_LSTAR	IA-32e Mode System Call Target Address (R/W) Target RIP for the called procedure when SYSCALL is executed in 64-bit mode.	If CPUID.80000001:EDX.[29] = 1
C000_0083H		IA32_CSTAR	IA-32e Mode System Call Target Address (R/W) Not used, as the SYSCALL instruction is not recognized in compatibility mode.	If CPUID.80000001:EDX.[29] = 1
C000_0084H		IA32_FMASK	System Call Flag Mask (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0100H		IA32_FS_BASE	Map of BASE Address of FS (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0101H		IA32_GS_BASE	Map of BASE Address of GS (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0102H		IA32_KERNEL_GS_BASE	Swap Target of BASE Address of GS (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0103H		IA32_TSC_AUX	Auxiliary TSC (Rw)	If CPUID.80000001H: EDX[27] = 1 or CPUID.(EAX=7,ECX=0):ECX[bit 22] = 1
		31:0	AUX: Auxiliary signature of TSC.	
		63:32	Reserved	

**NOTES:**

1. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.

2. The \*\_ADDR MSRs may or may not be present; this depends on flag settings in IA32\_MCI\_STATUS. See Section 15.3.2.3 and Section 15.3.2.4 for more information.
3. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].
4. Further details on Key Locker and usage of this MSR can be found here:  
<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

## 2.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 2-3 lists model-specific registers (MSRs) for Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 2-3. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_0FH, see Table 2-1.

MSRs listed in Table 2-2 and Table 2-3 are also supported by processors based on the Enhanced Intel Core microarchitecture. Processors based on the Enhanced Intel Core microarchitecture have the CPUID signature DisplayFamily\_DisplayModel of 06\_17H.

The column “Shared/Unique” applies to multi-core processors based on Intel Core microarchitecture. “Unique” means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. “Shared” means the MSR or the bit field in an MSR address governs the operation of both processor cores.

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Unique	See Section 2.23, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Unique	See Section 2.23, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, “Monitor/Mwait Address Range Determination.” and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, “Time-Stamp Counter,” and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	Platform ID (R) See Table 2-2.
17H	23	MSR_PLATFORM_ID	Shared	Model Specific Platform ID (R)
		7:0		Reserved
		12:8		Maximum Qualified Ratio (R) The maximum allowed bus ratio.
		49:13		Reserved
		52:50		See Table 2-2.
		63:53		Reserved
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, “Local APIC Status and Location” and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		1		Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement R/W.
		2		Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		MCERR# Drive Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement R/W.
		4		Address Parity Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement R/W.
		5		Reserved
		6		Reserved
		7		BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement R/W.
		8		Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled
		9		Execute BIST (R/O) 1 = Enabled; 0 = Disabled
		10		MCERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled
		11		Intel TXT Capable Chipset. (R/O) 1 = Present; 0 = Not Present
		12		BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled
		13		Reserved
		14		1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		APIC Cluster ID (R/O)
		18		N/2 Non-Integer Bus Ratio (R/O) 0 = Integer ratio; 1 = Non-integer ratio
		19		Reserved
		21:20		Symmetric Arbitration ID (R/O)
		26:22		Integer Bus Frequency Ratio (R/O)

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
3AH	58	MSR_FEATURE_CONTROL	Unique	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		3	Unique	SMRR Enable (R/WL) When this bit is set and the lock bit is set, this makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.
40H	64	MSR_LASTBRANCH_0_FROM_IP	Unique	Last Branch Record 0 From IP (R/W) One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.5.</li> </ul>
41H	65	MSR_LASTBRANCH_1_FROM_IP	Unique	Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_LASTBRANCH_2_FROM_IP	Unique	Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_LASTBRANCH_3_FROM_IP	Unique	Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_LASTBRANCH_0_TO_IP	Unique	Last Branch Record 0 To IP (R/W) One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_LASTBRANCH_1_TO_IP	Unique	Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_LASTBRANCH_2_TO_IP	Unique	Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_LASTBRANCH_3_TO_IP	Unique	Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	BIOS Update Signature ID (RO) See Table 2-2.
A0H	160	MSR_SMRR_PHYSBASE	Unique	System Management Mode Base Address register (WO in SMM) Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		11:0		Reserved
		31:12		PhysBase: SMRR physical Base Address.
		63:32		Reserved



**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
A1H	161	MSR_SMRR_PHYSMASK	Unique	System Management Mode Physical Address Mask register (WO in SMM) Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		10:0		Reserved
		11		Valid: Physical address base and range mask are valid.
		31:12		PhysMask: SMRR physical address range mask.
		63:32		Reserved
C1H	193	IA32_PMC0	Unique	Performance Counter Register See Table 2-2.
C2H	194	IA32_PMC1	Unique	Performance Counter Register See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	Scaleable Bus Speed(RO) This field indicates the intended scaleable bus clock speed for processors based on Intel Core microarchitecture.
		2:0		<ul style="list-style-type: none"> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> <li>▪ 010B: 200 MHz (FSB 800)</li> <li>▪ 000B: 267 MHz (FSB 1067)</li> <li>▪ 100B: 333 MHz (FSB 1333)</li> </ul>
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B. 266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B.
		63:3		Reserved
CDH	205	MSR_FSB_FREQ	Shared	Scaleable Bus Speed(RO) This field indicates the intended scaleable bus clock speed for processors based on Enhanced Intel Core microarchitecture.
		2:0		<ul style="list-style-type: none"> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> <li>▪ 010B: 200 MHz (FSB 800)</li> <li>▪ 000B: 267 MHz (FSB 1067)</li> <li>▪ 100B: 333 MHz (FSB 1333)</li> <li>▪ 110B: 400 MHz (FSB 1600)</li> </ul>

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B. 266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 110B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 111B.
		63:3		Reserved
E7H	231	IA32_MPERF	Unique	Maximum Performance Frequency Clock Count (RW) See Table 2-2.
E8H	232	IA32_APERF	Unique	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
		11	Unique	SMRR Capability Using MSR 0A0H and 0A1H (R)
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	Global Machine Check Status
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFEVTSELO	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
198H	408	MSR_PERF_STATUS	Shared	Current performance status. See Section 14.1.1, “Software Interface For Initiating Performance State Transitions”.
		15:0		Current Performance State Value
		30:16		Reserved
		31		XE Operation (R/O). If set, XE operation is enabled. Default is cleared.
		39:32		Reserved
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		45		Reserved
		46		Non-Integer Bus Ratio (R/O) Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture.
		63:47		Reserved
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	Thermal Interrupt Control (R/W) See Table 2-2.
19CH	412	IA32_THERM_STATUS	Unique	Thermal Monitor Status (R/W) See Table 2-2.
19DH	413	MSR_THERM2_CTL	Unique	Thermal Monitor 2 Control
		15:0		Reserved
		16		TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle). 1 = Thermal Monitor 2 (thermally-initiated frequency transitions). If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:16		Reserved
1A0H	416	IA32_MISC_ENABLE		Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		0		Fast-Strings Enable See Table 2-2.
		2:1		Reserved
		3	Unique	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2.
		6:4		Reserved
		7	Shared	Performance Monitoring Available (R) See Table 2-2.
		8		Reserved
		9		Hardware Prefetcher Disable (R/W) When set, disables the hardware prefetcher operation on streams of data. When clear (default), enables the prefetch queue. Disabling of the hardware prefetcher may impact processor performance.
		10	Shared	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor. 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12	Shared	Processor Event Based Sampling Unavailable (RO) See Table 2-2.
		13	Shared	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		16	Shared	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.
		18	Shared	ENABLE MONITOR FSM (R/W) See Table 2-2.
		19	Shared	Adjacent Cache Line Prefetch Disable (R/W) When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes). Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		20	Shared	Enhanced Intel SpeedStep Technology Select Lock (R/WO) When set, this bit causes the following bits to become read-only: <ul style="list-style-type: none"> <li>▪ Enhanced Intel SpeedStep Technology Select Lock (this bit).</li> <li>▪ Enhanced Intel SpeedStep Technology Enable bit.</li> </ul> The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved
		22	Shared	Limit CPUID Maxval (R/W) See Table 2-2.
		23	Shared	xTPR Message Disable (R/W) See Table 2-2.
		33:24		Reserved
		34	Unique	XD Bit Disable (R/W) See Table 2-2.
		36:35		Reserved
		37	Unique	DCU Prefetcher Disable (R/W) When set to 1, the DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		38	Shared	<p>IDA Disable (R/W)</p> <p>When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be cleared (CPUID.06H: EAX[1]=0).</p> <p>When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled.</p> <p>Note: The power-on default value is used by BIOS to detect hardware support of IDA. If the power-on default value is 1, IDA is available in the processor. If the power-on default value is 0, IDA is not available.</p>
		39	Unique	<p>IP Prefetcher Disable (R/W)</p> <p>When set to 1, the IP prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature.</p> <p>The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.</p>
		63:40		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Unique	<p>Last Branch Record Stack TOS (R/W)</p> <p>Contains an index (bits 0-3) that points to the MSR containing the most recent branch record.</p> <p>See MSR_LASTBRANCH_0_FROM_IP (at 40H).</p>
1D9H	473	IA32_DEBUGCTL	Unique	<p>Debug Control (R/W)</p> <p>See Table 2-2.</p>
1DDH	477	MSR_LER_FROM_LIP	Unique	<p>Last Exception Record From Linear IP (R/W)</p> <p>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</p>
1DEH	478	MSR_LER_TO_LIP	Unique	<p>Last Exception Record To Linear IP (R/W)</p> <p>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</p>
200H	512	IA32_MTRR_PHYSBASE0	Unique	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Unique	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Unique	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Unique	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Unique	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Unique	See Table 2-2.

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
206H	518	IA32_MTRR_PHYSBASE3	Unique	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Unique	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Unique	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Unique	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Unique	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Unique	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Unique	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Unique	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Unique	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Unique	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Unique	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Unique	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Unique	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Unique	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Unique	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Unique	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Unique	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Unique	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Unique	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Unique	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Unique	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	Default Memory Types (R/W) See Table 2-2.
309H	777	IA32_FIXED_CTR0	Unique	Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Unique	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Unique	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Unique	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
345H	837	MSR_PERF_CAPABILITIES	Unique	RO. This applies to processors that do not support architectural perfmon version 2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format
		7		PEBSSaveArchRegs. See Table 2-2.
		63:8		Reserved
38DH	909	IA32_FIXED_CTR_CTRL	Unique	Fixed-Function-Counter Control Register (R/W) See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	MSR_PERF_GLOBAL_CTRL	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MCO_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
402H	1026	IA32_MCO_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."



**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	IA32_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	IA32_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC3_MISC	Unique	Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISC_V flag in the IA32_MCi_STATUS register is set.
414H	1044	IA32_MC5_CTL	Unique	Machine Check Error Reporting Register: Controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
415H	1045	IA32_MC5_STATUS	Unique	Machine Check Error Reporting Register: Contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
416H	1046	IA32_MC5_ADDR	Unique	Machine Check Error Reporting Register: Contains the address of the code or data memory location that produced the machine-check error if the ADDR_V flag in the IA32_MCi_STATUS register is set.
417H	1047	IA32_MC5_MISC	Unique	Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISC_V flag in the IA32_MCi_STATUS register is set.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
419H	1045	IA32_MC6_STATUS	Unique	Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 22.
480H	1152	IA32_VMX_BASIC	Unique	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	Unique	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	Unique	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTL	Unique	Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTL	Unique	Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTL52	Unique	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Unique	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
107CC H	67532	MSR_EMON_L3_CTR_CTL0	Unique	GBUSQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CD H	67533	MSR_EMON_L3_CTR_CTL1	Unique	GBUSQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CE H	67534	MSR_EMON_L3_CTR_CTL2	Unique	GSNPQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CF H	67535	MSR_EMON_L3_CTR_CTL3	Unique	GSNPQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D0 H	67536	MSR_EMON_L3_CTR_CTL4	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D1 H	67537	MSR_EMON_L3_CTR_CTL5	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D2 H	67538	MSR_EMON_L3_CTR_CTL6	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D3 H	67539	MSR_EMON_L3_CTR_CTL7	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D8 H	67544	MSR_EMON_L3_GL_CTL	Unique	L3/FSB Common Control Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
C000_ 0080H		IA32_EFER	Unique	Extended Feature Enables See Table 2-2.
C000_ 0081H		IA32_STAR	Unique	System Call Target Address (R/W) See Table 2-2.
C000_ 0082H		IA32_LSTAR	Unique	IA-32e Mode System Call Target Address (R/W) See Table 2-2.

**Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
C000_0084H		IA32_FMASK	Unique	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	Unique	Map of BASE Address of FS (R/W) See Table 2-2.
C000_0101H		IA32_GS_BASE	Unique	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Unique	Swap Target of BASE Address of GS (R/W) See Table 2-2.

## 2.3 MSRS IN THE 45 NM AND 32 NM INTEL ATOM® PROCESSOR FAMILY

Table 2-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 2-4. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_1CH, 06\_26H, 06\_27H, 06\_35H and 06\_36H; see Table 2-1.

The column “Shared/Unique” applies to logical processors sharing the same core in processors based on the Intel Atom microarchitecture. “Unique” means each logical processor has a separate MSR, or a bit field in an MSR governs only a logical processor. “Shared” means the MSR or the bit field in an MSR address governs the operation of both logical processors in the same core.

**Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Shared	See Section 2.23, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Shared	See Section 2.23, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, “Monitor/Mwait Address Range Determination.” and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, “Time-Stamp Counter,” and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	Platform ID (R) See Table 2-2.
17H	23	MSR_PLATFORM_ID	Shared	Model Specific Platform ID (R)
		7:0		Reserved
		12:8		Maximum Qualified Ratio (R) The maximum allowed bus ratio.
		63:13		Reserved
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, “Local APIC Status and Location” and Table 2-2.

**Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
2AH	42	MSR_EBL_CR_POWERON	Shared	Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved
		1		Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Always 0.
		2		Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Always 0.
		3		AERR# Drive Enable (R/W) 1 = Enabled; 0 = Disabled Always 0.
		4		BERR# Enable for initiator bus requests (R/W) 1 = Enabled; 0 = Disabled Always 0.
		5		Reserved
		6		Reserved
		7		BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled Always 0.
		8		Reserved
		9		Execute BIST (R/O) 1 = Enabled; 0 = Disabled
		10		AERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled Always 0.
		11		Reserved
		12		BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled Always 0.
		13		Reserved
		14		1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		APIC Cluster ID (R/O) Always 00B.
19:18	Reserved			
21:20	Symmetric Arbitration ID (R/O) Always 00B.			

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		26:22		Integer Bus Frequency Ratio (R/O)
3AH	58	IA32_FEATURE_CONTROL	Unique	Control Features in Intel 64Processor (R/W) See Table 2-2.
40H	64	MSR_LASTBRANCH_0_FROM_IP	Unique	Last Branch Record 0 From IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.5.</li> </ul>
41H	65	MSR_LASTBRANCH_1_FROM_IP	Unique	Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_LASTBRANCH_2_FROM_IP	Unique	Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_LASTBRANCH_3_FROM_IP	Unique	Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_LASTBRANCH_4_FROM_IP	Unique	Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_LASTBRANCH_5_FROM_IP	Unique	Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_LASTBRANCH_6_FROM_IP	Unique	Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_LASTBRANCH_7_FROM_IP	Unique	Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_LASTBRANCH_0_TO_IP	Unique	Last Branch Record 0 To IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_LASTBRANCH_1_TO_IP	Unique	Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_LASTBRANCH_2_TO_IP	Unique	Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_LASTBRANCH_3_TO_IP	Unique	Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_LASTBRANCH_4_TO_IP	Unique	Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_LASTBRANCH_5_TO_IP	Unique	Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_LASTBRANCH_6_TO_IP	Unique	Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_LASTBRANCH_7_TO_IP	Unique	Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.

**Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
79H	121	IA32_BIOS_UPDT_TRIG	Shared	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	BIOS Update Signature ID (RO) See Table 2-2.
C1H	193	IA32_PMC0	Unique	Performance counter register See Table 2-2.
C2H	194	IA32_PMC1	Unique	Performance Counter Register See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	Scaleable Bus Speed(RO) This field indicates the intended scaleable bus clock speed for processors based on Intel Atom microarchitecture.
		2:0		<ul style="list-style-type: none"> <li>▪ 111B: 083 MHz (FSB 333)</li> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> </ul>
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
		63:3		Reserved
E7H	231	IA32_MPERF	Unique	Maximum Performance Frequency Clock Count (RW) See Table 2-2.
E8H	232	IA32_APERF	Unique	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Shared	Memory Type Range Register (R) See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	Control Register 3 Used to configure the L2 Cache.
		0		L2 Hardware Enabled (RO) 1 = Indicates the L2 is hardware-enabled. 0 = Indicates the L2 is hardware-disabled.
		7:1		Reserved
		8		L2 Enabled (R/W) 1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set, the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved
		23		L2 Not Present (RO) 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	Global Machine Check Status
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFEVTSELO	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	Performance Status
		15:0		Current Performance State Value
		39:16		Reserved
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		63:45		Reserved
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	Thermal Interrupt Control (R/W) See Table 2-2.
19CH	412	IA32_THERM_STATUS	Unique	Thermal Monitor Status (R/W) See Table 2-2.
19DH	413	MSR_THERM2_CTL	Shared	Thermal Monitor 2 Control



**Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		15:0		Reserved
		16		TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle). 1 = Thermal Monitor 2 (thermally-initiated frequency transitions). If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:17		Reserved
1A0H	416	IA32_MISC_ENABLE	Unique	Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		0		Fast-Strings Enable See Table 2-2.
		2:1		Reserved
		3	Unique	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 0.
		6:4		Reserved
		7	Shared	Performance Monitoring Available (R) See Table 2-2.
		8		Reserved
		9		Reserved
		10	Shared	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor. 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12	Shared	Processor Event Based Sampling Unavailable (RO) See Table 2-2.
		13	Shared	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
				<p>When this bit is cleared (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state.</p> <p>The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location.</p> <p>The processor is operating out of specification if both this bit and the TM1 bit are set to 0.</p>
		15:14		Reserved
		16	Shared	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.
		18	Shared	ENABLE MONITOR FSM (R/W) See Table 2-2.
		19		Reserved
		20	Shared	<p>Enhanced Intel SpeedStep Technology Select Lock (R/WO)</p> <p>When set, this bit causes the following bits to become read-only:</p> <ul style="list-style-type: none"> <li>▪ Enhanced Intel SpeedStep Technology Select Lock (this bit).</li> <li>▪ Enhanced Intel SpeedStep Technology Enable bit.</li> </ul> <p>The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.</p>
		21		Reserved
		22	Unique	Limit CPUID Maxval (R/W) See Table 2-2.
		23	Shared	xTPR Message Disable (R/W) See Table 2-2.
		33:24		Reserved
		34	Unique	XD Bit Disable (R/W) See Table 2-2.
		63:35		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Unique	<p>Last Branch Record Stack TOS (R/W)</p> <p>Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).</p>
1D9H	473	IA32_DEBUGCTL	Unique	<p>Debug Control (R/W)</p> <p>See Table 2-2.</p>
1DDH	477	MSR_LER_FROM_LIP	Unique	<p>Last Exception Record From Linear IP (R)</p> <p>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</p>

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
1DEH	478	MSR_LER_TO_LIP	Unique	Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
200H	512	IA32_MTRR_PHYSBASE0	Shared	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Shared	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Shared	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Shared	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Shared	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Shared	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Shared	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Shared	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Shared	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Shared	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Shared	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Shared	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Shared	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Shared	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Shared	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Shared	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Shared	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Shared	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Shared	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Shared	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Shared	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Shared	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Shared	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Shared	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Shared	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Shared	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Shared	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.
309H	777	IA32_FIXED_CTR0	Unique	Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Unique	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
30BH	779	IA32_FIXED_CTR2	Unique	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Shared	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Unique	Fixed-Function-Counter Control Register (R/W) See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0 (R/W)
400H	1024	IA32_MCO_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
402H	1026	IA32_MCO_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
408H	1032	IA32_MC2_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40AH	1034	IA32_MC2_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	IA32_MC3_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	IA32_MC4_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Unique	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	Unique	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	Unique	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTL	Unique	Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTL	Unique	Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

**Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)**

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	Unique	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Unique	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
C000_0080H		IA32_EFER	Unique	Extended Feature Enables See Table 2-2.
C000_0081H		IA32_STAR	Unique	System Call Target Address (R/W) See Table 2-2.
C000_0082H		IA32_LSTAR	Unique	IA-32e Mode System Call Target Address (R/W) See Table 2-2.
C000_0084H		IA32_FMASK	Unique	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	Unique	Map of BASE Address of FS (R/W) See Table 2-2.
C000_0101H		IA32_GS_BASE	Unique	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Unique	Swap Target of BASE Address of GS (R/W) See Table 2-2.

Table 2-5 lists model-specific registers (MSRs) that are specific to Intel Atom® processor with the CPUID signature with DisplayFamily\_DisplayModel of 06\_27H.

**Table 2-5. MSRs Supported by Intel Atom® Processors with CPUID Signature 06\_27H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C2_RESIDENCY	Package	Package C2 Residency Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0	Package	Package C2 Residency Counter (R/O) Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.

**Table 2-5. MSRs Supported by Intel Atom® Processors (Contd.)with CPUID Signature 06\_27H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3F9H	1017	MSR_PKG_C4_RESIDENCY	Package	Package C4 Residency Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0	Package	Package C4 Residency Counter. (R/O) Time that this package is in processor-specific C4 states since last reset. Counts at 1 Mhz frequency.
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	Package C6 Residency Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0	Package	Package C6 Residency Counter. (R/O) Time that this package is in processor-specific C6 states since last reset. Counts at 1 Mhz frequency.

## 2.4 MSRS IN INTEL PROCESSORS BASED ON SILVERMONT MICROARCHITECTURE

Table 2-6 lists model-specific registers (MSRs) common to Intel processors based on the Silvermont microarchitecture. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_37H, 06\_4AH, 06\_4DH, 06\_5AH, and 06\_5DH; see Table 2-1. The MSRs listed in Table 2-6 are also common to processors based on the Airmont microarchitecture and newer microarchitectures for next generation Intel Atom processors.

Table 2-7 lists MSRs common to processors based on the Silvermont and Airmont microarchitectures, but not newer microarchitectures.

Table 2-8, Table 2-9, and Table 2-10 lists MSRs that are model-specific across processors based on the Silvermont microarchitecture.

In the Silvermont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Silvermont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. "Package" means all processor cores in the physical package share the same MSR or bit interface.

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.23, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Core	See Section 8.10.5, "Monitor/Mwait Address Range Determination." and Table 2-2.

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
10H	16	IA32_TIME_STAMP_COUNTER	Core	See Section 17.17, "Time-Stamp Counter," and Table 2-2.
1BH	27	IA32_APIC_BASE	Core	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Module	Processor Hard Power-On Configuration (R/W) Writes ignored.
		63:0		Reserved
34H	52	MSR_SMI_COUNT	Core	SMI Counter (R/O)
		31:0		SMI Count (R/O) Running count of SMI events since last RESET.
		63:32		Reserved
79H	121	IA32_BIOS_UPDT_TRIG	Core	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Core	BIOS Update Signature ID (RO) See Table 2-2.
C1H	193	IA32_PMC0	Core	Performance counter register See Table 2-2.
C2H	194	IA32_PMC1	Core	Performance Counter Register See Table 2-2.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	Power Management IO Redirection in C-state (R/W) See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		C-state Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 100b - C4 is the max C-State to include 110b - C6 is the max C-State to include 111b - C7 is the max C-State to include
		63:19		Reserved
E7H	231	IA32_MPERF	Core	Maximum Performance Frequency Clock Count (RW) See Table 2-2.



**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E8H	232	IA32_APERF	Core	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	Memory Type Range Register (R) See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction sets availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note: AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved
174H	372	IA32_SYSENTER_CS	Core	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Core	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Core	See Table 2-2.
179H	377	IA32_MCG_CAP	Core	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Core	Global Machine Check Status
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFVTSELO	Core	See Table 2-2.
		7:0		Event Select

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		15:8		UMask
		16		USR
		17		OS
		18		Edge
		19		PC
		20		INT
		21		Reserved
		22		EN
		23		INV
		31:24		CMASK
		63:32		Reserved
187H	391	IA32_PERFEVTSEL1	Core	See Table 2-2.
198H	408	IA32_PERF_STATUS	Module	See Table 2-2.
199H	409	IA32_PERF_CTL	Core	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Core	Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Core	Thermal Interrupt Control (R/W) See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R) The default thermal throttling or PROCHOT# activation temperature in degrees C. The effective temperature for thermal throttling or PROCHOT# activation is "Temperature Target" + "Target Offset".
		29:24		Target Offset (R/W) Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).
		63:30		Reserved
1A6H	422	MSR_OFFCORE_RSP_0	Module	Offcore Response Event Select Register (R/W)
1A7H	423	MSR_OFFCORE_RSP_1	Module	Offcore Response Event Select Register (R/W)
1B0H	432	IA32_ENERGY_PERF_BIAS	Core	See Table 2-2.

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Core	Debug Control (R/W) See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Core	Last Exception Record From Linear IP (R/W) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Core	Last Exception Record To Linear IP (R/W) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Core	Default Memory Types (R/W) See Table 2-2.
309H	777	IA32_FIXED_CTR0	Core	Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Core	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Core	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Core	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Core	Fixed-Function-Counter Control Register (R/W) See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.
400H	1024	IA32_MCO_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
40AH	1034	IA32_MC2_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Core	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	Core	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
482H	1154	IA32_VMX_PROCBASED_CTL5	Core	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTL5	Core	Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTL5	Core	Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Core	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Core	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Core	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Core	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Core	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Core	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTL52	Core	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	Capability Reporting Register of EPT and VPID (R/O) See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTL5	Core	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL5	Core	Capability Reporting Register of Primary Processor-based VM-Execution Flex Controls (R/O) See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTL5	Core	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTL5	Core	Capability Reporting Register of VM-Entry Flex Controls (R/O) See Table 2-2
491H	1169	IA32_VMX_FMFUNC	Core	Capability Reporting Register of VM-Function Controls (R/O) See Table 2-2
4C1H	1217	IA32_A_PMC0	Core	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Core	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C1 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C1 states. Counts at the TSC frequency.
6E0H	1760	IA32_TSC_DEADLINE	Core	TSC Target of Local APIC's TSC Deadline Mode (R/W) See Table 2-2.
C000_0080H		IA32_EFER	Core	Extended Feature Enables See Table 2-2.
C000_0081H		IA32_STAR	Core	System Call Target Address (R/W) See Table 2-2.
C000_0082H		IA32_LSTAR	Core	IA-32e Mode System Call Target Address (R/W) See Table 2-2.
C000_0084H		IA32_FMASK	Core	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	Core	Map of BASE Address of FS (R/W) See Table 2-2.
C000_0101H		IA32_GS_BASE	Core	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Core	Swap Target of BASE Address of GS (R/W) See Table 2-2.

**Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C000_0103H		IA32_TSC_AUX	Core	AUXILIARY TSC Signature (R/W) See Table 2-2

Table 2-7 lists model-specific registers (MSRs) that are common to Intel® Atom™ processors based on the Silvermont and Airmont microarchitectures but not newer microarchitectures.

**Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	Model Specific Platform ID (R)
		7:0		Reserved
		13:8		Maximum Qualified Ratio (R) The maximum allowed bus ratio.
		49:13		Reserved
		52:50		See Table 2-2.
		63:33		Reserved
3AH	58	IA32_FEATURE_CONTROL	Core	Control Features in Intel 64Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Reserved
		2		Enable VMX outside SMX operation (R/WL)
40H	64	MSR_LASTBRANCH_0_FROM_IP	Core	Last Branch Record 0 From IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.5 and record format in Section 17.4.8.1.</li> </ul>
41H	65	MSR_LASTBRANCH_1_FROM_IP	Core	Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_LASTBRANCH_2_FROM_IP	Core	Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_LASTBRANCH_3_FROM_IP	Core	Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_LASTBRANCH_4_FROM_IP	Core	Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_LASTBRANCH_5_FROM_IP	Core	Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_LASTBRANCH_6_FROM_IP	Core	Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.



**Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
47H	71	MSR_LASTBRANCH_7_FROM_IP	Core	Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_LASTBRANCH_0_TO_IP	Core	Last Branch Record 0 To IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_LASTBRANCH_1_TO_IP	Core	Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_LASTBRANCH_2_TO_IP	Core	Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_LASTBRANCH_3_TO_IP	Core	Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_LASTBRANCH_4_TO_IP	Core	Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_LASTBRANCH_5_TO_IP	Core	Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_LASTBRANCH_6_TO_IP	Core	Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_LASTBRANCH_7_TO_IP	Core	Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information: Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * Scalable Bus Frequency.
		63:16		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Module	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-state support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only).
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.
		63:16		Reserved
11EH	281	MSR_BBL_CR_CTL3	Module	Control Register 3 Used to configure the L2 Cache.
		0		L2 Hardware Enabled (RO) 1 = If the L2 is hardware-enabled. 0 = Indicates if the L2 is hardware-disabled.
		7:1		Reserved
		8		L2 Enabled (R/W) 1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved
		23		L2 Not Present (RO) 0 = L2 Present. 1 = L2 Not Present.
		63:24		Reserved
1A0H	416	IA32_MISC_ENABLE		Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		0	Core	Fast-Strings Enable See Table 2-2.

**Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2:1		Reserved
		3	Module	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 0.
		6:4		Reserved
		7	Core	Performance Monitoring Available (R) See Table 2-2.
		10:8		Reserved
		11	Core	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12	Core	Processor Event Based Sampling Unavailable (RO) See Table 2-2.
		15:13		Reserved
		16	Module	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.
		18	Core	ENABLE MONITOR FSM (R/W) See Table 2-2.
		21:19		Reserved
		22	Core	Limit CPUID Maxval (R/W) See Table 2-2.
		23	Module	xTPR Message Disable (R/W) See Table 2-2.
		33:24		Reserved
		34	Core	XD Bit Disable (R/W) See Table 2-2.
		37:35		Reserved
		38	Module	<p>Turbo Mode Disable (R/W)</p> <p>When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be cleared (CPUID.06H: EAX[1]=0).</p> <p>When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled.</p> <p>Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.</p>
		63:39		Reserved
1C8H	456	MSR_LBR_SELECT	Core	Last Branch Record Filtering Select Register (R/W) See Section 17.9.2, "Filtering of Last Branch Records."

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
		63:9		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Core	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
38EH	910	IA32_PERF_GLOBAL_STATUS	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS for precise event on IA32_PMC0 (R/W)
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter (R/O) Value since last reset that this package is in processor-specific C6 states. Counts at the TSC Frequency.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	Module C6 Residency Counter (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

### 2.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 2-8 lists model-specific registers (MSRs) that are specific to Intel Atom® processor E3000 Series (CPUID signature with DisplayFamily\_DisplayModel of 06\_37H) and Intel Atom processors (CPUID signatures with DisplayFamily\_DisplayModel of 06\_4AH, 06\_5AH, 06\_5DH).

**Table 2-8. Specific MSRs Supported by Intel Atom® Processors with CPUID Signatures 06\_37H, 06\_4AH, 06\_5AH, 06\_5DH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CDH	205	MSR_FSB_FREQ	Module	Scaleable Bus Speed(R/O) This field indicates the intended scaleable bus clock speed for processors based on Silvermont microarchitecture.
		2:0		<ul style="list-style-type: none"> <li>▪ 100B: 080.0 MHz</li> <li>▪ 000B: 083.3 MHz</li> <li>▪ 001B: 100.0 MHz</li> <li>▪ 010B: 133.3 MHz</li> <li>▪ 011B: 116.7 MHz</li> </ul>
		63:3		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers used in RAPL Interfaces (R/O) See Section 14.10.1, "RAPL Interfaces."
		3:0		Power Units Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	PKG RAPL Power Limit Control (R/W)
		14:0		Package Power Limit #1 (R/W) See Section 14.10.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1 (R/W) See Section 14.10.3, "Package RAPL Domain."
		16		Package Clamping Limitation #1 (R/W) See Section 14.10.3, "Package RAPL Domain."
		23:17		Time Window for Power Limit #1 (R/W) In unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.
		63:24		Reserved
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.

**Table 2-8. Specific MSRs Supported by Intel Atom® Processors with CPUID Signatures 06\_37H, 06\_4AH, 06\_5AH, 06\_5DH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8.

Table 2-9 lists model-specific registers (MSRs) that are specific to Intel Atom® processor E3000 Series (CPUID signature with DisplayFamily\_DisplayModel of 06\_37H).

**Table 2-9. Specific MSRs Supported by Intel Atom® Processor E3000 Series with CPUID Signature 06\_37H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
668H	1640	MSR_CC6_DEMOTION_POLICY_CONFIG	Package	Core C6 Demotion Policy Config MSR
		63:0		Controls per-core C6 demotion policy. Writing a value of 0 disables core level HW demotion policy.
669H	1641	MSR_MC6_DEMOTION_POLICY_CONFIG	Package	Module C6 Demotion Policy Config MSR
		63:0		Controls module (i.e., two cores sharing the second-level cache) C6 demotion policy. Writing a value of 0 disables module level HW demotion policy.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	Module C6 Residency Counter (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

Table 2-10 lists model-specific registers (MSRs) that are specific to Intel Atom® processor C2000 Series (CPUID signature with DisplayFamily\_DisplayModel of 06\_4DH).

**Table 2-10. Specific MSRs Supported by Intel Atom® Processor C2000 Series with CPUID Signature 06\_4DH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1A4H	420	MSR_MISC_FEATURE_CONTROL		Miscellaneous Feature Control (R/W)
		0	Core	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		Reserved
		2	Core	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved

**Table 2-10. Specific MSRs Supported by Intel Atom® Processor C2000 Series (Contd.)with CPUID Signature**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode (RW)
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		39:32	Package	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 core active.
		47:40	Package	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 core active.
		55:48	Package	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 core active.
		63:56	Package	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 core active.
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers used in RAPL Interfaces (R/O) See Section 14.10.1, "RAPL Interfaces."
		3:0		Power Units Power related information (in milliWatts) is based on the multiplier, $2^{PU}$ ; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, $2^{ESU}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
		610H	1552	MSR_PKG_POWER_LIMIT
66EH	1646	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameter (R/O)

**Table 2-10. Specific MSRs Supported by Intel Atom® Processor C2000 Series (Contd.)with CPUID Signature**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:0		Thermal Spec Power (R/O) The unsigned integer value is the equivalent of the thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
		63:15		Reserved

## 2.4.2 MSRs In Intel Atom® Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 2-6, Table 2-7, Table 2-8, and Table 2-11. These processors have a CPUID signature with DisplayFamily\_DisplayModel including 06\_4CH; see Table 2-1.

**Table 2-11. MSRs in Intel Atom® Processors Based on the Airmont Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CDH	205	MSR_FSB_FREQ	Module	Scaleable Bus Speed(RO) This field indicates the intended scaleable bus clock speed for processors based on Airmont microarchitecture.
		3:0		<ul style="list-style-type: none"> <li>▪ 0000B: 083.3 MHz</li> <li>▪ 0001B: 100.0 MHz</li> <li>▪ 0010B: 133.3 MHz</li> <li>▪ 0011B: 116.7 MHz</li> <li>▪ 0100B: 080.0 MHz</li> <li>▪ 0101B: 093.3 MHz</li> <li>▪ 0110B: 090.0 MHz</li> <li>▪ 0111B: 088.9 MHz</li> <li>▪ 1000B: 087.5 MHz</li> </ul>
		63:5		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Module	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .



**Table 2-11. MSRs in Intel Atom® Processors Based on the Airmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.
		63:16		Reserved
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	Power Management IO Redirection in C-state (R/W) See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		C-state Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - Deep Power Down Technology is the max C-State. 010b - C7 is the max C-State to include.
		63:19		Reserved
638H	1592	MSR_PPO_POWER_LIMIT	Package	PPO RAPL Power Limit Control (R/W)

**Table 2-11. MSRs in Intel Atom® Processors Based on the Airmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:0		PPO Power Limit #1 (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains" and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1 (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
		16		Reserved
		23:17		Time Window for Power Limit #1 (R/W) Specifies the time duration over which the average power must remain below PPO_POWER_LIMIT #1(14:0). Supported Encodings: 0x0: 1 second time duration. 0x1: 5 second time duration (Default). 0x2: 10 second time duration. 0x3: 15 second time duration. 0x4: 20 second time duration. 0x5: 25 second time duration. 0x6: 30 second time duration. 0x7: 35 second time duration. 0x8: 40 second time duration. 0x9: 45 second time duration. 0xA: 50 second time duration. 0xB-0x7F - reserved.
		63:24		Reserved

## 2.5 MSRS IN INTEL ATOM® PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Intel Atom processors based on the Goldmont microarchitecture support MSRs listed in Table 2-6 and Table 2-12. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_5CH; see Table 2-1.

In the Goldmont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Goldmont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. "Package" means all processor cores in the physical package share the same MSR or bit interface.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	Model Specific Platform ID (R)

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		49:0		Reserved
		52:50		See Table 2-2.
		63:33		Reserved
3AH	58	IA32_FEATURE_CONTROL	Core	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Enable VMX inside SMX operation (R/WL)
		2		Enable VMX outside SMX operation (R/WL)
		14:8		SENTER local functions enables (R/WL)
		15		SENTER global functions enable (R/WL)
		18		SGX global functions enable (R/WL)
		63:19		Reserved
3BH	59	IA32_TSC_ADJUST	Core	Per-Core TSC ADJUST (R/W) See Table 2-2.
C3H	195	IA32_PMC2	Core	Performance Counter Register See Table 2-2.
C4H	196	IA32_PMC3	Core	Performance Counter Register See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * 100 MHz.
		27:16		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	Programmable TJ OFFSET (R/O) When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify a temperature offset.
		39:31		Reserved

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		3:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.
		63:16		Reserved
17DH	381	MSR_SMM_MCA_CAP	Core	Enhanced SMM Capabilities (SMM-RO) Reports SMM capability enhancement. Accessible only while in SMM.
		57:0		Reserved
		58		SMM_Code_Access_Chk (SMM-RO) If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		59		Long_Flow_Indication (SMM-RO) If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
188H	392	IA32_PERFEVTSEL2	Core	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Core	See Table 2-2.
1A0H	416	IA32_MISC_ENABLE		Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		0	Core	Fast-Strings Enable See Table 2-2.
		2:1		Reserved
		3	Package	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 1.
		6:4		Reserved
		7	Core	Performance Monitoring Available (R) See Table 2-2.
		10:8		Reserved
		11	Core	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12	Core	Processor Event Based Sampling Unavailable (RO) See Table 2-2.
		15:13		Reserved
		16	Package	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.
		18	Core	ENABLE MONITOR FSM (R/W) See Table 2-2.
		21:19		Reserved
		22	Core	Limit CPUID Maxval (R/W) See Table 2-2.
		23	Package	xTPR Message Disable (R/W) See Table 2-2.
		33:24		Reserved
34	Core	XD Bit Disable (R/W) See Table 2-2.		
37:35		Reserved		

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		38	Package	<p>Turbo Mode Disable (R/W)</p> <p>When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0).</p> <p>When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled.</p> <p>Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.</p>
		63:39		Reserved
1A4H	420	MSR_MISC_FEATURE_CONTROL		Miscellaneous Feature Control (R/W)
		0	Core	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		Reserved
		2	Core	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved
1AAH	426	MSR_MISC_PWR_MGMT	Package	Miscellaneous Power Management Control Various model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		EIST Hardware Coordination Disable (R/W) When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores. When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		21:1		Reserved
		22		Thermal Interrupt Coordination Enable (R/W) If set, then thermal interrupt on one core is routed to all cores.
		63:23		Reserved

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode by Core Groups (RW) Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically. For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.
		7:0	Package	Maximum Ratio Limit for Active Cores in Group 0 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 0 threshold.
		15:8	Package	Maximum Ratio Limit for Active Cores in Group 1 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 1 threshold, and greater than the Group 0 threshold.
		23:16	Package	Maximum Ratio Limit for Active Cores in Group 2 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 2 threshold, and greater than the Group 1 threshold.
		31:24	Package	Maximum Ratio Limit for Active Cores in Group 3 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 3 threshold, and greater than the Group 2 threshold.
		39:32	Package	Maximum Ratio Limit for Active Cores in Group 4 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 4 threshold, and greater than the Group 3 threshold.
		47:40	Package	Maximum Ratio Limit for Active Cores in Group 5 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 5 threshold, and greater than the Group 4 threshold.
		55:48	Package	Maximum Ratio Limit for Active Cores in Group 6 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 6 threshold, and greater than the Group 5 threshold.
		63:56	Package	Maximum Ratio Limit for Active Cores in Group 7 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 7 threshold, and greater than the Group 6 threshold.
1AEH	430	MSR_TURBO_GROUP_CORECNT	Package	Group Size of Active Cores for Turbo Mode Operation (RW) Writes of 0 threshold is ignored.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0	Package	Group 0 Core Count Threshold Maximum number of active cores to operate under the Group 0 Max Turbo Ratio limit.
		15:8	Package	Group 1 Core Count Threshold Maximum number of active cores to operate under the Group 1 Max Turbo Ratio limit. Must be greater than the Group 0 Core Count.
		23:16	Package	Group 2 Core Count Threshold Maximum number of active cores to operate under the Group 2 Max Turbo Ratio limit. Must be greater than the Group 1 Core Count.
		31:24	Package	Group 3 Core Count Threshold Maximum number of active cores to operate under the Group 3 Max Turbo Ratio limit. Must be greater than the Group 2 Core Count.
		39:32	Package	Group 4 Core Count Threshold Maximum number of active cores to operate under the Group 4 Max Turbo Ratio limit. Must be greater than the Group 3 Core Count.
		47:40	Package	Group 5 Core Count Threshold Maximum number of active cores to operate under the Group 5 Max Turbo Ratio limit. Must be greater than the Group 4 Core Count.
		55:48	Package	Group 6 Core Count Threshold Maximum number of active cores to operate under the Group 6 Max Turbo Ratio limit. Must be greater than the Group 5 Core Count.
		63:56	Package	Group 7 Core Count Threshold Maximum number of active cores to operate under the Group 7 Max Turbo Ratio limit. Must be greater than the Group 6 Core Count, and not less than the total number of processor cores in the package. E.g., specify 255.
1C8H	456	MSR_LBR_SELECT	Core	Last Branch Record Filtering Select Register (R/W) See Section 17.9.2, "Filtering of Last Branch Records."
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP



**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		8		FAR_BRANCH
		9		EN_CALL_STACK
		63:10		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Core	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-4) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		Reserved
		1	Package	C1E Enable (R/W) When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved
210H	528	IA32_MTRR_PHYSBASE8	Core	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Core	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Core	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Core	See Table 2-2.
280H	640	IA32_MC0_CTL2	Module	See Table 2-2.
281H	641	IA32_MC1_CTL2	Module	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Module	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	See Table 2-2.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
300H	768	MSR_SGXOWNEREPOCH0	Package	Lower 64 Bit CR_SGXOWNEREPOCH (W) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.
301H	769	MSR_SGXOWNEREPOCH1	Package	Upper 64 Bit CR_SGXOWNEREPOCH (W) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.
38EH	910	IA32_PERF_GLOBAL_STATUS	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0		Ovf_PMCO

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		Ovf_PMC1
		2		Ovf_PMC2
		3		Ovf_PMC3
		31:4		Reserved
		32		Ovf_FixedCtr0
		33		Ovf_FixedCtr1
		34		Ovf_FixedCtr2
		54:35		Reserved
		55		Trace_ToPA_PMI
		57:56		Reserved
		58		LBR_Frz.
		59		CTR_Frz.
		60		ASCI
		61		Ovf_Uncore
		62		Ovf_BufDSSAVE
63		CondChgd		
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0		Set 1 to clear Ovf_PMC0.
		1		Set 1 to clear Ovf_PMC1.
		2		Set 1 to clear Ovf_PMC2.
		3		Set 1 to clear Ovf_PMC3.
		31:4		Reserved
		32		Set 1 to clear Ovf_FixedCtr0.
		33		Set 1 to clear Ovf_FixedCtr1.
		34		Set 1 to clear Ovf_FixedCtr2.
		54:35		Reserved
		55		Set 1 to clear Trace_ToPA_PMI.
		57:56		Reserved
		58		Set 1 to clear LBR_Frz.
		59		Set 1 to clear CTR_Frz.
		60		Set 1 to clear ASCI.
61		Set 1 to clear Ovf_Uncore.		
62		Set 1 to clear Ovf_BufDSSAVE.		
63		Set 1 to clear CondChgd.		
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		Set 1 to cause Ovf_PMC0 = 1.
		1		Set 1 to cause Ovf_PMC1 = 1.
		2		Set 1 to cause Ovf_PMC2 = 1.
		3		Set 1 to cause Ovf_PMC3 = 1.
		31:4		Reserved
		32		Set 1 to cause Ovf_FixedCtr0 = 1.
		33		Set 1 to cause Ovf_FixedCtr1 = 1.
		34		Set 1 to cause Ovf_FixedCtr2 = 1.
		54:35		Reserved
		55		Set 1 to cause Trace_ToPA_PMI = 1.
		57:56		Reserved
		58		Set 1 to cause LBR_Frz = 1.
		59		Set 1 to cause CTR_Frz = 1.
		60		Set 1 to cause ASCI = 1.
		61		Set 1 to cause Ovf_Uncore.
		62		Set 1 to cause Ovf_BufDSSAVE.
		63		Reserved
392H	914	IA32_PERF_GLOBAL_INUSE	Core	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:0		CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
406H	1030	IA32_MC1_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
4C3H	1219	IA32_A_PMC2	Core	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Core	See Table 2-2.
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	Enhanced SMM Feature Control (SMM-RW) Reports SMM capability Enhancement. Accessible only while in SMM.
		0		Lock (SMM-RW0) When set to '1' locks this register from further changes.
		1		Reserved
		2		SMM_Code_Chk_En (SMM-RW) This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	SMM Delayed (SMM-RO) Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		N-1:0		LOG_PROC_STATE (SMM-RO) Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	SMM Blocked (SMM-RO) Reports the blocked state of all logical processors in the package. Available only while in SMM.
		N-1:0		LOG_PROC_STATE (SMM-RO) Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is OFFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
500H	1280	IA32_SGX_SVN_STATUS	Core	Status and SVN Threshold of SGX Support for ACM (RO)
		0		Lock See Section 37.1.1.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1		Reserved
		23:16		SGX_SVN_SINIT See Section 37.1.1.3, "Interactions with Authenticated Code Modules (ACMs)".
		63:24		Reserved
560H	1376	IA32_RTIT_OUTPUT_BASE	Core	Trace Output Base Register (R/W) See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Core	Trace Output Mask Pointers Register (R/W) See Table 2-2.
570H	1392	IA32_RTIT_CTL	Core	Trace Control Register (R/W)
		0		TraceEn
		1		CYCEn
		2		OS
		3		User
		6:4		Reserved, must be zero.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		MTCEn
		10		TSCEn
		11		DisRETC
		12		Reserved, must be zero.
		13		BranchEn
		17:14		MTCFreq
		18		Reserved, must be zero.
		22:19		CYCThresh
		23		Reserved, must be zero.
		27:24		PSBFreq
		31:28		Reserved, must be zero.
		35:32		ADDRO_CFG
		39:36		ADDR1_CFG
		63:40		Reserved, must be zero.
571H	1393	IA32_RTIT_STATUS	Core	Tracing Status Register (R/W)
		0		FilterEn Writes ignored.
		1		ContexEn Writes ignored.
		2		TriggerEn Writes ignored.
		3		Reserved
		4		Error (R/W)
		5		Stopped
		31:6		Reserved, must be zero.
		48:32		PacketByteCnt
		63:49		Reserved, must be zero.
572H	1394	IA32_RTIT_CR3_MATCH	Core	Trace Filter CR3 Match Register (R/W)
		4:0		Reserved
		63:5		CR3[63:5] value to match.
580H	1408	IA32_RTIT_ADDRO_A	Core	Region 0 Start Address (R/W)
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDRO_B	Core	Region 0 End Address (R/W)
		63:0		See Table 2-2.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
582H	1410	IA32_RTIT_ADDR1_A	Core	Region 1 Start Address (R/W)
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Core	Region 1 End Address (R/W)
		63:0		See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers used in RAPL Interfaces (R/O) See Section 14.10.1, "RAPL Interfaces."
		3:0		Power Units Power related information (in Watts) is in unit of $1W/2^{PU}$ ; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units Energy related information (in Joules) is in unit of $1Joule/2^{ESU}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.
		15:13		Reserved
		19:16		Time Unit Time related information (in seconds) is in unit of $1S/2^{TU}$ ; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.
		63:20		Reserved
60AH	1546	MSR_PKGC3_IRTL	Package	Package C3 Interrupt Response Limit (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
60BH	1547	MSR_PKG_C2_IRTL1	Package	Package C6/C7S Interrupt Response Limit 1 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7S state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved
60CH	1548	MSR_PKG_C2_IRTL2	Package	Package C7 Interrupt Response Limit 2 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.



**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:0		Package C2 Residency Counter (R/O) Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	PKG RAPL Power Limit Control (R/W) See Section 14.10.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	PKG Perf Status (R/O) See Section 14.10.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameters (R/W)
		14:0		Thermal Spec Power (R/W) See Section 14.10.3, "Package RAPL Domain."
		15		Reserved
		30:16		Minimum Power (R/W) See Section 14.10.3, "Package RAPL Domain."
		31		Reserved
		46:32		Maximum Power (R/W) See Section 14.10.3, "Package RAPL Domain."
		47		Reserved
		54:48		Maximum Time Window (R/W) Specified by $2^Y * (1.0 + Z/4.0) * \text{Time\_Unit}$ , where "Y" is the unsigned integer value represented by bits 52:48, "Z" is an unsigned integer represented by bits 54:53. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
		63:55		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
632H	1586		Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		63:0		Package C10 Residency Counter (R/O) Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	PP1 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	ConfigTDP Control (R/W)
		7:0		MAX_NON_TURBO_RATIO (R/W/L) System BIOS can program this field.
		30:8		Reserved
		31		TURBO_ACTIVATION_RATIO_Lock (R/W/L) When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)
		0		PROCHOT Status (R0) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.
		2		Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		3		Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		8:4		Reserved
		9		Core Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
11		Max Turbo Limit Status (R0) When set, frequency is reduced below the operating system request due to multi-core turbo limits.		

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12		Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).
		13		Turbo Transition Attenuation Status (R0) When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		14		Maximum Efficiency Frequency Status (R0) When set, frequency is reduced below the maximum efficiency frequency.
		15		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24:20		Reserved
		25		Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		26		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		Maximum Efficiency Frequency Log When set, indicates that the Maximum Efficiency Frequency Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:31		Reserved
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Core	Last Branch Record 0 From IP (R/W) One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.6 and record format in Section 17.4.8.1.</li> </ul>
		0:47		From Linear Address (R/W)
		62:48		Signed extension of bits 47:0.
		63		Mispred
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Core	Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Core	Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Core	Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Core	Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Core	Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Core	Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Core	Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Core	Last Branch Record 8 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Core	Last Branch Record 9 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Core	Last Branch Record 10 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	Core	Last Branch Record 11 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	Core	Last Branch Record 12 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	Core	Last Branch Record 13 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	Core	Last Branch Record 14 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	Core	Last Branch Record 15 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
690H	1680	MSR_LASTBRANCH_16_FROM_IP	Core	Last Branch Record 16 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
691H	1681	MSR_LASTBRANCH_17_FROM_IP	Core	Last Branch Record 17 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_LASTBRANCH_18_FROM_IP	Core	Last Branch Record 18 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_LASTBRANCH_19_FROM_IP	Core	Last Branch Record 19 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_LASTBRANCH_20_FROM_IP	Core	Last Branch Record 20 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_LASTBRANCH_21_FROM_IP	Core	Last Branch Record 21 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_LASTBRANCH_22_FROM_IP	Core	Last Branch Record 22 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
697H	1687	MSR_LASTBRANCH_23_FROM_IP	Core	Last Branch Record 23 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_LASTBRANCH_24_FROM_IP	Core	Last Branch Record 24 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_LASTBRANCH_25_FROM_IP	Core	Last Branch Record 25 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_LASTBRANCH_26_FROM_IP	Core	Last Branch Record 26 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_LASTBRANCH_27_FROM_IP	Core	Last Branch Record 27 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_LASTBRANCH_28_FROM_IP	Core	Last Branch Record 28 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_LASTBRANCH_29_FROM_IP	Core	Last Branch Record 29 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_LASTBRANCH_30_FROM_IP	Core	Last Branch Record 30 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_LASTBRANCH_31_FROM_IP	Core	Last Branch Record 31 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Core	Last Branch Record 0 To IP (R/W) One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the Destination instruction and elapsed cycles from last LBR update. See Section 17.6.
		0:47		Target Linear Address (R/W)
		63:48		Elapsed cycles from last update to the LBR.
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	Core	Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	Core	Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	Core	Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	Core	Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	Core	Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	Core	Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	Core	Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	Core	Last Branch Record 8 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	Core	Last Branch Record 9 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	Core	Last Branch Record 10 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	Core	Last Branch Record 11 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	Core	Last Branch Record 12 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	Core	Last Branch Record 13 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	Core	Last Branch Record 14 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	Core	Last Branch Record 15 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DOH	1744	MSR_LASTBRANCH_16_TO_IP	Core	Last Branch Record 16 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D1H	1745	MSR_LASTBRANCH_17_TO_IP	Core	Last Branch Record 17 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_LASTBRANCH_18_TO_IP	Core	Last Branch Record 18 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_LASTBRANCH_19_TO_IP	Core	Last Branch Record 19 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_LASTBRANCH_20_TO_IP	Core	Last Branch Record 20 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_LASTBRANCH_21_TO_IP	Core	Last Branch Record 21 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_LASTBRANCH_22_TO_IP	Core	Last Branch Record 22 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_LASTBRANCH_23_TO_IP	Core	Last Branch Record 23 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_LASTBRANCH_24_TO_IP	Core	Last Branch Record 24 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_LASTBRANCH_25_TO_IP	Core	Last Branch Record 25 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_LASTBRANCH_26_TO_IP	Core	Last Branch Record 26 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
6DBH	1755	MSR_LASTBRANCH_27_TO_IP	Core	Last Branch Record 27 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_LASTBRANCH_28_TO_IP	Core	Last Branch Record 28 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_LASTBRANCH_29_TO_IP	Core	Last Branch Record 29 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_LASTBRANCH_30_TO_IP	Core	Last Branch Record 30 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_LASTBRANCH_31_TO_IP	Core	Last Branch Record 31 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Core	x2APIC ID register (R/O)
803H	2051	IA32_X2APIC_VERSION	Core	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Core	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Core	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Core	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Core	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Core	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Core	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Core	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Core	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Core	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Core	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Core	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Core	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Core	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Core	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Core	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Core	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Core	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Core	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Core	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Core	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Core	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Core	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Core	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Core	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Core	x2APIC Interrupt Request register bits [127:96] (R/O)



**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
824H	2084	IA32_X2APIC_IRR4	Core	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Core	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Core	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Core	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Core	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Core	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Core	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Core	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Core	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Core	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Core	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Core	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Core	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Core	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Core	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Core	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Core	x2APIC Self IPI register (W/O)
C8FH	3215	IA32_PQR_ASSOC	Core	Resource Association Register (R/W)
		31:0		Reserved
		33:32		COS (R/W)
		63: 34		Reserved
D10H	3344	IA32_L2_QOS_MASK_0	Module	L2 Class Of Service Mask - COS 0 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0.
		0:7		CBM: Bit vector of available L2 ways for COS 0 enforcement.
		63:8		Reserved
D11H	3345	IA32_L2_QOS_MASK_1	Module	L2 Class Of Service Mask - COS 1 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1.
		0:7		CBM: Bit vector of available L2 ways for COS 0 enforcement.
		63:8		Reserved
D12H	3346	IA32_L2_QOS_MASK_2	Module	L2 Class Of Service Mask - COS 2 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2.
		0:7		CBM: Bit vector of available L2 ways for COS 0 enforcement.

**Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:8		Reserved
D13H	3347	IA32_L2_QOS_MASK_3	Package	L2 Class Of Service Mask - COS 3 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3.
		0:19		CBM: Bit vector of available L2 ways for COS 3 enforcement.
		63:20		Reserved
D90H	3472	IA32_BNDCFGS	Core	See Table 2-2.
DA0H	3488	IA32_XSS	Core	See Table 2-2.

See Table 2-6, and Table 2-12 for MSR definitions applicable to processors with CPUID signature 06\_5CH.

## 2.6 MSRS IN INTEL ATOM® PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Intel Atom processors based on the Goldmont Plus microarchitecture support MSRs listed in Table 2-6, Table 2-12 and Table 2-13. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_7AH; see Table 2-1. For an MSR listed in Table 2-13 that also appears in the model-specific tables of prior generations, Table 2-13 supersedes prior generation tables.

In the Goldmont Plus microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Goldmont Plus microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. “Package” means all processor cores in the physical package share the same MSR or bit interface.

**Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Core	Control Features in Intel 64Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Enable VMX inside SMX operation (R/WL)
		2		Enable VMX outside SMX operation (R/WL)
		14:8		SENTER local functions enables (R/WL)
		15		SENTER global functions enable (R/WL)
		17		SGX Launch Control Enable (R/WL) This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Valid if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.
		18		SGX global functions enable (R/WL)

**Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:19		Reserved
8CH	140	IA32_SGXLEPUBKEYHASH0	Core	See Table 2-2.
8DH	141	IA32_SGXLEPUBKEYHASH1	Core	See Table 2-2.
8EH	142	IA32_SGXLEPUBKEYHASH2	Core	See Table 2-2.
8FH	143	IA32_SGXLEPUBKEYHASH3	Core	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	(R/W) See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0.
		1		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC1.
		2		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC2.
		3		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC3.
		31:4		Reserved
		32		Enable PEBS trigger and recording for IA32_FIXED_CTRO.
		33		Enable PEBS trigger and recording for IA32_FIXED_CTR1.
		34		Enable PEBS trigger and recording for IA32_FIXED_CTR2.
		63:35		Reserved
570H	1392	IA32_RTIT_CTL	Core	Trace Control Register (R/W)
		0		TraceEn
		1		CYCEn
		2		OS
		3		User
		4		PwrEvtEn
		5		FUPonPTW
		6		FabricEn
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		MTCEn
		10		TSCEn
11		DisRETC		

Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12		PTWEn
		13		BranchEn
		17:14		MTCFreq
		18		Reserved, must be zero.
		22:19		CYCThresh
		23		Reserved, must be zero.
		27:24		PSBFreq
		31:28		Reserved, must be zero.
		35:32		ADDR0_CFG
		39:36		ADDR1_CFG
		63:40		Reserved, must be zero.
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Core	Last Branch Record 0 From IP (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. The From_IP part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>Last Branch Record Stack TOS at 1C9H.</li> <li>Section 17.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."</li> </ul>
681H - 69FH	1665 - 1695	MSR_LASTBRANCH_1_FROM_IP	Core	Last Branch Record <i>i</i> From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP; <i>i</i> = 1-31.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Core	Last Branch Record 0 To IP (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. The To_IP part of the stack contains pointers to the Destination instruction. See also: <ul style="list-style-type: none"> <li>Section 17.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."</li> </ul>
6C1H - 6DFH	1729 - 1759	MSR_LASTBRANCH_1_TO_IP	Core	Last Branch Record <i>i</i> To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP; <i>i</i> = 1-31.
DC0H	3520	MSR_LASTBRANCH_INFO_0	Core	Last Branch Record 0 Additional Information (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. This part of the stack contains flag and elapsed cycle information. See also: <ul style="list-style-type: none"> <li>Last Branch Record Stack TOS at 1C9H.</li> <li>Section 17.9.1, "LBR Stack."</li> </ul>
DC1H	3521	MSR_LASTBRANCH_INFO_1	Core	Last Branch Record 1 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC2H	3522	MSR_LASTBRANCH_INFO_2	Core	Last Branch Record 2 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.

**Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DC3H	3523	MSR_LASTBRANCH_INFO_3	Core	Last Branch Record 3 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC4H	3524	MSR_LASTBRANCH_INFO_4	Core	Last Branch Record 4 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC5H	3525	MSR_LASTBRANCH_INFO_5	Core	Last Branch Record 5 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC6H	3526	MSR_LASTBRANCH_INFO_6	Core	Last Branch Record 6 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC7H	3527	MSR_LASTBRANCH_INFO_7	Core	Last Branch Record 7 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC8H	3528	MSR_LASTBRANCH_INFO_8	Core	Last Branch Record 8 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC9H	3529	MSR_LASTBRANCH_INFO_9	Core	Last Branch Record 9 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCAH	3530	MSR_LASTBRANCH_INFO_10	Core	Last Branch Record 10 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCBH	3531	MSR_LASTBRANCH_INFO_11	Core	Last Branch Record 11 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCCH	3532	MSR_LASTBRANCH_INFO_12	Core	Last Branch Record 12 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCDH	3533	MSR_LASTBRANCH_INFO_13	Core	Last Branch Record 13 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCEH	3534	MSR_LASTBRANCH_INFO_14	Core	Last Branch Record 14 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCFH	3535	MSR_LASTBRANCH_INFO_15	Core	Last Branch Record 15 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD0H	3536	MSR_LASTBRANCH_INFO_16	Core	Last Branch Record 16 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD1H	3537	MSR_LASTBRANCH_INFO_17	Core	Last Branch Record 17 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD2H	3538	MSR_LASTBRANCH_INFO_18	Core	Last Branch Record 18 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD3H	3539	MSR_LASTBRANCH_INFO_19	Core	Last Branch Record 19 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD4H	3520	MSR_LASTBRANCH_INFO_20	Core	Last Branch Record 20 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD5H	3521	MSR_LASTBRANCH_INFO_21	Core	Last Branch Record 21 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.

**Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DD6H	3522	MSR_LASTBRANCH_INFO_22	Core	Last Branch Record 22 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD7H	3523	MSR_LASTBRANCH_INFO_23	Core	Last Branch Record 23 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD8H	3524	MSR_LASTBRANCH_INFO_24	Core	Last Branch Record 24 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD9H	3525	MSR_LASTBRANCH_INFO_25	Core	Last Branch Record 25 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDAH	3526	MSR_LASTBRANCH_INFO_26	Core	Last Branch Record 26 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDBH	3527	MSR_LASTBRANCH_INFO_27	Core	Last Branch Record 27 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDCH	3528	MSR_LASTBRANCH_INFO_28	Core	Last Branch Record 28 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDDH	3529	MSR_LASTBRANCH_INFO_29	Core	Last Branch Record 29 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDEH	3530	MSR_LASTBRANCH_INFO_30	Core	Last Branch Record 30 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDFH	3531	MSR_LASTBRANCH_INFO_31	Core	Last Branch Record 31 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.

See Table 2-6, Table 2-12 and Table 2-13 for MSR definitions applicable to processors with CPUID signature 06\_7AH.

## 2.7 MSRS IN INTEL ATOM® PROCESSORS BASED ON TREMONT MICROARCHITECTURE

Intel Atom processors based on the Tremont microarchitecture support MSRs listed in Table 2-6, Table 2-12, Table 2-13 and Table 2-14. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_86H; see Table 2-1. For an MSR listed in Table 2-14 that also appears in the model-specific tables of prior generations, Table 2-14 supersedes prior generation tables.

In the Tremont microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Tremont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. “Package” means all processor cores in the physical package share the same MSR or bit interface.

**Table 2-14. MSRs in Intel Atom® Processors Based on the Tremont Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
33H	51	MSR_TEST_CTRL	Core	Test Control Register
		28:0		Reserved.
		29		Enable #AC(0) exception for split locked accesses: Cause #AC(0) exception for split locked access at all CPL irrespective of CR0.AM or EFLAGS.AC. If bits 29 and 31 are both set, bit 29 takes precedence.
		30		Reserved.
		31		Reserved.
CFH	207	IA32_CORE_CAPABILITIES	Core	IA32 Core Capabilities Register If CPUID.(EAX=07H, ECX=0):EDX[30] = 1.
		4:0		Reserved.
		5		Bit 29 of MSR_TEST_CTRL (address 33H) supported.
		63:6		Reserved.
3F1H	1009	MSR_PEBS_ENABLE	Core	(R/W) See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)".
		<i>n</i> :0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMCx. The maximum value <i>n</i> can be determined from CPUID.0AH:EAX[15:8].
		31: <i>n</i> +1		Reserved.
		32+ <i>m</i> :32		Enable PEBS trigger and recording for IA32_FIXED_CTRx. The maximum value <i>m</i> can be determined from CPUID.0AH:EDX[4:0].
		59:33+ <i>m</i>		Reserved.
		60		Pend a PerfMon Interrupt (PMI) after each PEBS event.
		62:61		Specifies PEBS output destination. Encodings: 00B: DS Save Area 01B: Intel PT trace output. Supported if IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] and CPUID.07H.0.EBX[25] are set. 10B: Reserved 11B: Reserved
		63		Reserved.
1309H - 130BH	4873 - 4875	MSR_RELOAD_FIXED_CTRx		Reload value for IA32_FIXED_CTRx (R/W)
		47:0		Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and FIXED_CTRx is overflowed.
		63:48		Reserved.

**Table 2-14. MSRs in Intel Atom® Processors Based on the Tremont Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
14C1H	5313	MSR_RELOAD_PMCx		Reload value for IA32_PMCx (R/W)
-	-	47:0		Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and PMCx is overflowed.
14C8H	5320	63:48		Reserved.

See Table 2-6, Table 2-12, Table 2-13 and Table 2-14 for MSR definitions applicable to processors with CPUID signature 06\_86H.

## 2.8 MSRS IN THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table 2-15 lists model-specific registers (MSRs) that are common for Intel® microarchitecture code name Nehalem. These include Intel Core i7 and i5 processor family. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_1AH, 06\_1EH, 06\_1FH, 06\_2EH, see Table 2-1. Additional MSRs specific to 06\_1AH, 06\_1EH, 06\_1FH are listed in Table 2-16. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at <http://biosbits.org>.

The column "Scope" represents the package/core/thread scope of individual bit field of an MSR. "Thread" means this bit field must be programmed on each logical processor independently. "Core" means the bit field must be programmed on each processor core independently, logical processors in the same core will be affected by change of this bit on the other logical processor in the same core. "Package" means the bit field must be programmed once for each physical package. Change of a bit filed with a package scope will affect all logical processors in that physical package.

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.23, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination" and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	Platform ID (R) See Table 2-2.
17H	23	MSR_PLATFORM_ID	Package	Model Specific Platform ID (R)
		49:0		Reserved
		52:50		See Table 2-2.
		63:53		Reserved
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.



**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
34H	52	MSR_SMI_COUNT	Thread	SMI Counter (R/O)
		31:0		SMI Count (R/O) Running count of SMI events since last RESET.
		63:32		Reserved
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64Processor (R/W) See Table 2-2.
79H	121	IA32_BIOS_UPDT_TRIG	Core	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Thread	BIOS Update Signature ID (RO) See Table 2-2.
C1H	193	IA32_PMC0	Thread	Performance Counter Register See Table 2-2.
C2H	194	IA32_PMC1	Thread	Performance Counter Register See Table 2-2.
C3H	195	IA32_PMC2	Thread	Performance Counter Register See Table 2-2.
C4H	196	IA32_PMC3	Thread	Performance Counter Register See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.
		27:16		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDC-TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDC and TDP Limits for Turbo mode are programmable. When set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 133.33MHz.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.  The following C-state code name encodings are supported: 000b: C0 (no package C-state support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.
		23:16		Reserved
		24		Interrupt filtering enable (R/W) When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.
		25		C3 state auto demotion enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		26		C1 state auto demotion enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		29		Package C State Demotion Enable (R/W)
		30		Package C State UnDemotion Enable (R/W)
		63:31		Reserved
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	Power Management IO Redirection in C-state (R/W) See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		C-state Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - C6 is the max C-State to include. 010b - C7 is the max C-State to include.
		63:19		Reserved
E7H	231	IA32_MPERF	Thread	Maximum Performance Frequency Clock Count (RW) See Table 2-2.
E8H	232	IA32_APERF	Thread	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFEVTSELO	Thread	See Table 2-2.
		7:0		Event Select
		15:8		UMask
		16		USR
		17		OS
		18		Edge
		19		PC
		20		INT
		21		AnyThread
		22		EN
		23		INV
		31:24		CMASK
		63:32		Reserved
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Core	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
19AH	410	IA32_CLOCK_MODULATION	Thread	Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		0		Reserved
		3:1		On demand Clock Modulation Duty Cycle (R/W)
		4		On demand Clock Modulation Enable (R/W)
		63:5		Reserved
19BH	411	IA32_THERM_INTERRUPT	Core	Thermal Interrupt Control (R/W) See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
1A0H	416	IA32_MISC_ENABLE		Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		0	Thread	Fast-Strings Enable See Table 2-2.
		2:1		Reserved
		3	Thread	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 1.
		6:4		Reserved
		7	Thread	Performance Monitoring Available (R) See Table 2-2.
		10:8		Reserved
		11	Thread	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12	Thread	Processor Event Based Sampling Unavailable (RO) See Table 2-2.
		15:13		Reserved
		16	Package	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 2-2.
		21:19		Reserved
		22	Thread	Limit CPUID Maxval (R/W) See Table 2-2.
23	Thread	xTPR Message Disable (R/W) See Table 2-2.		
33:24		Reserved		

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		34	Thread	XD Bit Disable (R/W) See Table 2-2.
		37:35		Reserved
		38	Package	Turbo Mode Disable (R/W) When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.
		63:39		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Thread	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R) The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.
		63:24		Reserved
1A4H	420	MSR_MISC_FEATURE_CONTROL		Miscellaneous Feature Control (R/W)
		0	Core	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	L2 Adjacent Cache Line Prefetcher Disable (R/W) If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	DCU IP Prefetcher Disable (R/W) If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved
1A6H	422	MSR_OFFCORE_RSP_0	Thread	Offcore Response Event Select Register (R/W)

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1AAH	426	MSR_MISC_PWR_MGMT		Miscellaneous Power Management Control Various model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		0	Package	EIST Hardware Coordination Disable (R/W) When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores. When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		1	Thread	Energy/Performance Bias Enable (R/W) This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].
		63:2		Reserved
1ACH	428	MSR_TURBO_POWER_CURRENT_LIMIT		See <a href="http://biosbits.org">http://biosbits.org</a> .
		14:0	Package	TDP Limit (R/W) TDP limit in 1/8 Watt granularity.
		15	Package	TDP Limit Override Enable (R/W) A value = 0 indicates override is not active; a value = 1 indicates override is active.
		30:16	Package	TDC Limit (R/W) TDC limit in 1/8 Amp granularity.
		31	Package	TDC Limit Override Enable (R/W) A value = 0 indicates override is not active; a value = 1 indicates override is active.
		63:32		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		63:32		Reserved
1C8H	456	MSR_LBR_SELECT	Core	Last Branch Record Filtering Select Register (R/W) See Section 17.9.2, "Filtering of Last Branch Records."

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
		63:9		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Thread	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	Debug Control (R/W) See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Thread	Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		Reserved
		1	Package	C1E Enable (R/W) When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.



**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Thread	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Package	See Table 2-2.
281H	641	IA32_MC1_CTL2	Package	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Core	See Table 2-2.
285H	645	IA32_MC5_CTL2	Core	See Table 2-2.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	Default Memory Types (R/W) See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format See Table 2-2.
		6		PEBS Record Format
		7		PEBSSaveArchRegs See Table 2-2.
		11:8		PEBS_REC_FORMAT See Table 2-2.
		12		SMM_FREEZE See Table 2-2.
		63:13		Reserved
38DH	909	IA32_FIXED_CTR_CTRL	Thread	Fixed-Function-Counter Control Register (R/W) See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Thread	Provides single-bit status used by software to query the overflow condition of each performance counter. (RO)
		61		UNC_Ovf Uncore overflowed if 1.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities." Allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Thread	(R/W)

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		61		CLR_UNC_Ovf Set 1 to clear UNC_Ovf.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0 (R/W)
		1		Enable PEBS on IA32_PMC1 (R/W)
		2		Enable PEBS on IA32_PMC2 (R/W)
		3		Enable PEBS on IA32_PMC3 (R/W)
		31:4		Reserved
		32		Enable Load Latency on IA32_PMC0 (R/W)
		33		Enable Load Latency on IA32_PMC1 (R/W)
		34		Enable Load Latency on IA32_PMC2 (R/W)
		35		Enable Load Latency on IA32_PMC3 (R/W)
		63:36		Reserved
3F6H	1014	MSR_PEBS_LD_LAT	Thread	See Section 18.3.1.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
400H	1024	IA32_MCO_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
402H	1026	IA32_MCO_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MCO_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
406H	1030	IA32_MC1_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC4_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
414H	1044	IA32_MC5_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
416H	1046	IA32_MC5_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
480H	1152	IA32_VMX_BASIC	Thread	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	Thread	Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	Thread	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTL	Thread	Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTL	Thread	Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CRO_FIXED0	Thread	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
487H	1159	IA32_VMX_CRO_FIXED1	Thread	Capability Reporting Register of CRO Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Thread	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTL2	Thread	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Thread	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	Last Branch Record 0 From IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.9.1 and record format in Section 17.4.8.1.</li> </ul>
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Thread	Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Thread	Last Branch Record 8 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Thread	Last Branch Record 9 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Thread	Last Branch Record 10 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	Thread	Last Branch Record 11 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	Thread	Last Branch Record 12 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	Thread	Last Branch Record 13 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	Thread	Last Branch Record 14 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	Thread	Last Branch Record 15 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Thread	Last Branch Record 0 To IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	Thread	Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	Thread	Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	Thread	Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	Thread	Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	Thread	Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	Thread	Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	Thread	Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	Thread	Last Branch Record 8 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	Thread	Last Branch Record 9 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	Thread	Last Branch Record 10 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	Thread	Last Branch Record 11 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	Thread	Last Branch Record 12 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	Thread	Last Branch Record 13 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	Thread	Last Branch Record 14 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	Thread	Last Branch Record 15 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID Register (R/O)



**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version Register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority Register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority Register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI Register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination Register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector Register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service Register Bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service Register Bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service Register Bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service Register Bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service Register Bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service Register Bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service Register Bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service Register Bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode Register Bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode Register Bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode Register Bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode Register Bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode Register Bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode Register Bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode Register Bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode Register Bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request Register Bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request Register Bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request Register Bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request Register Bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request Register Bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request Register Bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request Register Bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request Register Bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status Register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt Register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command Register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt Register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt Register (R/W)

**Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor Register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 Register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 Register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error Register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count Register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count Register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration Register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI Register (W/O)
C000_0080H		IA32_EFER	Thread	Extended Feature Enables See Table 2-2.
C000_0081H		IA32_STAR	Thread	System Call Target Address (R/W) See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	IA-32e Mode System Call Target Address (R/W) See Table 2-2.
C000_0084H		IA32_FMASK	Thread	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	Map of BASE Address of FS (R/W) See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	Swap Target of BASE Address of GS (R/W) See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	AUXILIARY TSC Signature (R/W) See Table 2-2 and Section 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

### 2.8.1 Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

Intel Xeon Processor 5500 and 3400 series support additional model-specific registers listed in Table 2-16. These MSRs also apply to Intel Core i7 and i5 processor family CPUID signature with DisplayFamily\_DisplayModel of 06\_1AH, 06\_1EH and 06\_1FH, see Table 2-1.

**Table 2-16. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Actual maximum turbo frequency is multiplied by 133.33MHz. (Not available in model 06_2EH.)

**Table 2-16. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0		Maximum Turbo Ratio Limit 1C (R/O) Maximum Turbo mode ratio limit with 1 core active.
		15:8		Maximum Turbo Ratio Limit 2C (R/O) Maximum Turbo mode ratio limit with 2 cores active.
		23:16		Maximum Turbo Ratio Limit 3C (R/O) Maximum Turbo mode ratio limit with 3 cores active.
		31:24		Maximum Turbo Ratio Limit 4C (R/O) Maximum Turbo mode ratio limit with 4 cores active.
		63:32		Reserved
301H	769	MSR_GQ_SNOOP_MESF	Package	
		0		From M to S (R/W)
		1		From E to S (R/W)
		2		From S to S (R/W)
		3		From F to S (R/W)
		4		From M to I (R/W)
		5		From E to I (R/W)
		6		From S to I (R/W)
		7		From F to I (R/W)
63:8		Reserved		
391H	913	MSR_UNCORE_PERF_GLOBAL_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
392H	914	MSR_UNCORE_PERF_GLOBAL_STATUS	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
393H	915	MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
394H	916	MSR_UNCORE_FIXED_CTR0	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
395H	917	MSR_UNCORE_FIXED_CTR_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
396H	918	MSR_UNCORE_ADDR_OPCODE_MATCH	Package	See Section 18.3.1.2.3, "Uncore Address/Opcode Match MSR."
3B0H	960	MSR_UNCORE_PMC0	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B1H	961	MSR_UNCORE_PMC1	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B2H	962	MSR_UNCORE_PMC2	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B3H	963	MSR_UNCORE_PMC3	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B4H	964	MSR_UNCORE_PMC4	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

**Table 2-16. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3B6H	966	MSR_UNCORE_PMC6	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3B7H	967	MSR_UNCORE_PMC7	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C0H	944	MSR_UNCORE_PERFEVTSELO	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C1H	945	MSR_UNCORE_PERFEVTSEL1	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C2H	946	MSR_UNCORE_PERFEVTSEL2	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C3H	947	MSR_UNCORE_PERFEVTSEL3	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C4H	948	MSR_UNCORE_PERFEVTSEL4	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C5H	949	MSR_UNCORE_PERFEVTSEL5	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C6H	950	MSR_UNCORE_PERFEVTSEL6	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C7H	951	MSR_UNCORE_PERFEVTSEL7	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”

## 2.8.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table 2-15 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-17. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2EH.

**Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Reserved Attempt to read/write will cause #UD.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
394H	816	MSR_W_PMON_FIXED_CTR	Package	Uncore W-box perfmon fixed counter.
395H	817	MSR_W_PMON_FIXED_CTR_CTL	Package	Uncore U-box perfmon fixed counter control MSR.
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRS."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRS."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRS."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRS."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRS."
451H	1105	IA32_MC20_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
452H	1106	IA32_MC20_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
453H	1107	IA32_MC20_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRS."
455H	1109	IA32_MC21_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
456H	1110	IA32_MC21_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS."
457H	1111	IA32_MC21_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRS."
C00H	3072	MSR_U_PMON_GLOBAL_CTRL	Package	Uncore U-box perfmon global control MSR.
C01H	3073	MSR_U_PMON_GLOBAL_STATUS	Package	Uncore U-box perfmon global status MSR.
C02H	3074	MSR_U_PMON_GLOBAL_OVF_CTRL	Package	Uncore U-box perfmon global overflow control MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C10H	3088	MSR_U_PMON_EVNT_SEL	Package	Uncore U-box perfmon event select MSR.
C11H	3089	MSR_U_PMON_CTR	Package	Uncore U-box perfmon counter MSR.
C20H	3104	MSR_B0_PMON_BOX_CTRL	Package	Uncore B-box 0 perfmon local box control MSR.
C21H	3105	MSR_B0_PMON_BOX_STATUS	Package	Uncore B-box 0 perfmon local box status MSR.
C22H	3106	MSR_B0_PMON_BOX_OVF_CTRL	Package	Uncore B-box 0 perfmon local box overflow control MSR.
C30H	3120	MSR_B0_PMON_EVNT_SELO	Package	Uncore B-box 0 perfmon event select MSR.
C31H	3121	MSR_B0_PMON_CTR0	Package	Uncore B-box 0 perfmon counter MSR.
C32H	3122	MSR_B0_PMON_EVNT_SEL1	Package	Uncore B-box 0 perfmon event select MSR.
C33H	3123	MSR_B0_PMON_CTR1	Package	Uncore B-box 0 perfmon counter MSR.
C34H	3124	MSR_B0_PMON_EVNT_SEL2	Package	Uncore B-box 0 perfmon event select MSR.
C35H	3125	MSR_B0_PMON_CTR2	Package	Uncore B-box 0 perfmon counter MSR.
C36H	3126	MSR_B0_PMON_EVNT_SEL3	Package	Uncore B-box 0 perfmon event select MSR.
C37H	3127	MSR_B0_PMON_CTR3	Package	Uncore B-box 0 perfmon counter MSR.
C40H	3136	MSR_S0_PMON_BOX_CTRL	Package	Uncore S-box 0 perfmon local box control MSR.
C41H	3137	MSR_S0_PMON_BOX_STATUS	Package	Uncore S-box 0 perfmon local box status MSR.
C42H	3138	MSR_S0_PMON_BOX_OVF_CTRL	Package	Uncore S-box 0 perfmon local box overflow control MSR.
C50H	3152	MSR_S0_PMON_EVNT_SELO	Package	Uncore S-box 0 perfmon event select MSR.
C51H	3153	MSR_S0_PMON_CTR0	Package	Uncore S-box 0 perfmon counter MSR.
C52H	3154	MSR_S0_PMON_EVNT_SEL1	Package	Uncore S-box 0 perfmon event select MSR.
C53H	3155	MSR_S0_PMON_CTR1	Package	Uncore S-box 0 perfmon counter MSR.
C54H	3156	MSR_S0_PMON_EVNT_SEL2	Package	Uncore S-box 0 perfmon event select MSR.
C55H	3157	MSR_S0_PMON_CTR2	Package	Uncore S-box 0 perfmon counter MSR.
C56H	3158	MSR_S0_PMON_EVNT_SEL3	Package	Uncore S-box 0 perfmon event select MSR.
C57H	3159	MSR_S0_PMON_CTR3	Package	Uncore S-box 0 perfmon counter MSR.
C60H	3168	MSR_B1_PMON_BOX_CTRL	Package	Uncore B-box 1 perfmon local box control MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C61H	3169	MSR_B1_PMON_BOX_STATUS	Package	Uncore B-box 1 perfmon local box status MSR.
C62H	3170	MSR_B1_PMON_BOX_OVF_CTRL	Package	Uncore B-box 1 perfmon local box overflow control MSR.
C70H	3184	MSR_B1_PMON_EVNT_SELO	Package	Uncore B-box 1 perfmon event select MSR.
C71H	3185	MSR_B1_PMON_CTR0	Package	Uncore B-box 1 perfmon counter MSR.
C72H	3186	MSR_B1_PMON_EVNT_SEL1	Package	Uncore B-box 1 perfmon event select MSR.
C73H	3187	MSR_B1_PMON_CTR1	Package	Uncore B-box 1 perfmon counter MSR.
C74H	3188	MSR_B1_PMON_EVNT_SEL2	Package	Uncore B-box 1 perfmon event select MSR.
C75H	3189	MSR_B1_PMON_CTR2	Package	Uncore B-box 1 perfmon counter MSR.
C76H	3190	MSR_B1_PMON_EVNT_SEL3	Package	Uncore B-box 1 vperfmon event select MSR.
C77H	3191	MSR_B1_PMON_CTR3	Package	Uncore B-box 1 perfmon counter MSR.
C80H	3120	MSR_W_PMON_BOX_CTRL	Package	Uncore W-box perfmon local box control MSR.
C81H	3121	MSR_W_PMON_BOX_STATUS	Package	Uncore W-box perfmon local box status MSR.
C82H	3122	MSR_W_PMON_BOX_OVF_CTRL	Package	Uncore W-box perfmon local box overflow control MSR.
C90H	3136	MSR_W_PMON_EVNT_SELO	Package	Uncore W-box perfmon event select MSR.
C91H	3137	MSR_W_PMON_CTR0	Package	Uncore W-box perfmon counter MSR.
C92H	3138	MSR_W_PMON_EVNT_SEL1	Package	Uncore W-box perfmon event select MSR.
C93H	3139	MSR_W_PMON_CTR1	Package	Uncore W-box perfmon counter MSR.
C94H	3140	MSR_W_PMON_EVNT_SEL2	Package	Uncore W-box perfmon event select MSR.
C95H	3141	MSR_W_PMON_CTR2	Package	Uncore W-box perfmon counter MSR.
C96H	3142	MSR_W_PMON_EVNT_SEL3	Package	Uncore W-box perfmon event select MSR.
C97H	3143	MSR_W_PMON_CTR3	Package	Uncore W-box perfmon counter MSR.
CA0H	3232	MSR_M0_PMON_BOX_CTRL	Package	Uncore M-box 0 perfmon local box control MSR.
CA1H	3233	MSR_M0_PMON_BOX_STATUS	Package	Uncore M-box 0 perfmon local box status MSR.
CA2H	3234	MSR_M0_PMON_BOX_OVF_CTRL	Package	Uncore M-box 0 perfmon local box overflow control MSR.
CA4H	3236	MSR_M0_PMON_TIMESTAMP	Package	Uncore M-box 0 perfmon time stamp unit select MSR.
CA5H	3237	MSR_M0_PMON_DSP	Package	Uncore M-box 0 perfmon DSP unit select MSR.



Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CA6H	3238	MSR_M0_PMON_ISS	Package	Uncore M-box 0 perfmon ISS unit select MSR.
CA7H	3239	MSR_M0_PMON_MAP	Package	Uncore M-box 0 perfmon MAP unit select MSR.
CA8H	3240	MSR_M0_PMON_MSC_THR	Package	Uncore M-box 0 perfmon MIC THR select MSR.
CA9H	3241	MSR_M0_PMON_PGT	Package	Uncore M-box 0 perfmon PGT unit select MSR.
CAAH	3242	MSR_M0_PMON_PLD	Package	Uncore M-box 0 perfmon PLD unit select MSR.
CABH	3243	MSR_M0_PMON_ZDP	Package	Uncore M-box 0 perfmon ZDP unit select MSR.
CBOH	3248	MSR_M0_PMON_EVNT_SELO	Package	Uncore M-box 0 perfmon event select MSR.
CB1H	3249	MSR_M0_PMON_CTRL0	Package	Uncore M-box 0 perfmon counter MSR.
CB2H	3250	MSR_M0_PMON_EVNT_SEL1	Package	Uncore M-box 0 perfmon event select MSR.
CB3H	3251	MSR_M0_PMON_CTRL1	Package	Uncore M-box 0 perfmon counter MSR.
CB4H	3252	MSR_M0_PMON_EVNT_SEL2	Package	Uncore M-box 0 perfmon event select MSR.
CB5H	3253	MSR_M0_PMON_CTRL2	Package	Uncore M-box 0 perfmon counter MSR.
CB6H	3254	MSR_M0_PMON_EVNT_SEL3	Package	Uncore M-box 0 perfmon event select MSR.
CB7H	3255	MSR_M0_PMON_CTRL3	Package	Uncore M-box 0 perfmon counter MSR.
CB8H	3256	MSR_M0_PMON_EVNT_SEL4	Package	Uncore M-box 0 perfmon event select MSR.
CB9H	3257	MSR_M0_PMON_CTRL4	Package	Uncore M-box 0 perfmon counter MSR.
CBAH	3258	MSR_M0_PMON_EVNT_SEL5	Package	Uncore M-box 0 perfmon event select MSR.
CBBH	3259	MSR_M0_PMON_CTRL5	Package	Uncore M-box 0 perfmon counter MSR.
CCOH	3264	MSR_S1_PMON_BOX_CTRL	Package	Uncore S-box 1 perfmon local box control MSR.
CC1H	3265	MSR_S1_PMON_BOX_STATUS	Package	Uncore S-box 1 perfmon local box status MSR.
CC2H	3266	MSR_S1_PMON_BOX_OVF_CTRL	Package	Uncore S-box 1 perfmon local box overflow control MSR.
CDOH	3280	MSR_S1_PMON_EVNT_SELO	Package	Uncore S-box 1 perfmon event select MSR.
CD1H	3281	MSR_S1_PMON_CTRL0	Package	Uncore S-box 1 perfmon counter MSR.
CD2H	3282	MSR_S1_PMON_EVNT_SEL1	Package	Uncore S-box 1 perfmon event select MSR.
CD3H	3283	MSR_S1_PMON_CTRL1	Package	Uncore S-box 1 perfmon counter MSR.
CD4H	3284	MSR_S1_PMON_EVNT_SEL2	Package	Uncore S-box 1 perfmon event select MSR.
CD5H	3285	MSR_S1_PMON_CTRL2	Package	Uncore S-box 1 perfmon counter MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CD6H	3286	MSR_S1_PMON_EVNT_SEL3	Package	Uncore S-box 1 perfmon event select MSR.
CD7H	3287	MSR_S1_PMON_CTR3	Package	Uncore S-box 1 perfmon counter MSR.
CE0H	3296	MSR_M1_PMON_BOX_CTRL	Package	Uncore M-box 1 perfmon local box control MSR.
CE1H	3297	MSR_M1_PMON_BOX_STATUS	Package	Uncore M-box 1 perfmon local box status MSR.
CE2H	3298	MSR_M1_PMON_BOX_OVF_CTRL	Package	Uncore M-box 1 perfmon local box overflow control MSR.
CE4H	3300	MSR_M1_PMON_TIMESTAMP	Package	Uncore M-box 1 perfmon time stamp unit select MSR.
CE5H	3301	MSR_M1_PMON_DSP	Package	Uncore M-box 1 perfmon DSP unit select MSR.
CE6H	3302	MSR_M1_PMON_ISS	Package	Uncore M-box 1 perfmon ISS unit select MSR.
CE7H	3303	MSR_M1_PMON_MAP	Package	Uncore M-box 1 perfmon MAP unit select MSR.
CE8H	3304	MSR_M1_PMON_MSC_THR	Package	Uncore M-box 1 perfmon MIC THR select MSR.
CE9H	3305	MSR_M1_PMON_PGT	Package	Uncore M-box 1 perfmon PGT unit select MSR.
CEAH	3306	MSR_M1_PMON_PLD	Package	Uncore M-box 1 perfmon PLD unit select MSR.
CEBH	3307	MSR_M1_PMON_ZDP	Package	Uncore M-box 1 perfmon ZDP unit select MSR.
CF0H	3312	MSR_M1_PMON_EVNT_SEL0	Package	Uncore M-box 1 perfmon event select MSR.
CF1H	3313	MSR_M1_PMON_CTR0	Package	Uncore M-box 1 perfmon counter MSR.
CF2H	3314	MSR_M1_PMON_EVNT_SEL1	Package	Uncore M-box 1 perfmon event select MSR.
CF3H	3315	MSR_M1_PMON_CTR1	Package	Uncore M-box 1 perfmon counter MSR.
CF4H	3316	MSR_M1_PMON_EVNT_SEL2	Package	Uncore M-box 1 perfmon event select MSR.
CF5H	3317	MSR_M1_PMON_CTR2	Package	Uncore M-box 1 perfmon counter MSR.
CF6H	3318	MSR_M1_PMON_EVNT_SEL3	Package	Uncore M-box 1 perfmon event select MSR.
CF7H	3319	MSR_M1_PMON_CTR3	Package	Uncore M-box 1 perfmon counter MSR.
CF8H	3320	MSR_M1_PMON_EVNT_SEL4	Package	Uncore M-box 1 perfmon event select MSR.
CF9H	3321	MSR_M1_PMON_CTR4	Package	Uncore M-box 1 perfmon counter MSR.
CFAH	3322	MSR_M1_PMON_EVNT_SEL5	Package	Uncore M-box 1 perfmon event select MSR.
CFBH	3323	MSR_M1_PMON_CTR5	Package	Uncore M-box 1 perfmon counter MSR.
D00H	3328	MSR_C0_PMON_BOX_CTRL	Package	Uncore C-box 0 perfmon local box control MSR.
D01H	3329	MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon local box status MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
D02H	3330	MSR_C0_PMON_BOX_OVF_CTRL	Package	Uncore C-box 0 perfmon local box overflow control MSR.
D10H	3344	MSR_C0_PMON_EVNT_SELO	Package	Uncore C-box 0 perfmon event select MSR.
D11H	3345	MSR_C0_PMON_CTRL0	Package	Uncore C-box 0 perfmon counter MSR.
D12H	3346	MSR_C0_PMON_EVNT_SEL1	Package	Uncore C-box 0 perfmon event select MSR.
D13H	3347	MSR_C0_PMON_CTRL1	Package	Uncore C-box 0 perfmon counter MSR.
D14H	3348	MSR_C0_PMON_EVNT_SEL2	Package	Uncore C-box 0 perfmon event select MSR.
D15H	3349	MSR_C0_PMON_CTRL2	Package	Uncore C-box 0 perfmon counter MSR.
D16H	3350	MSR_C0_PMON_EVNT_SEL3	Package	Uncore C-box 0 perfmon event select MSR.
D17H	3351	MSR_C0_PMON_CTRL3	Package	Uncore C-box 0 perfmon counter MSR.
D18H	3352	MSR_C0_PMON_EVNT_SEL4	Package	Uncore C-box 0 perfmon event select MSR.
D19H	3353	MSR_C0_PMON_CTRL4	Package	Uncore C-box 0 perfmon counter MSR.
D1AH	3354	MSR_C0_PMON_EVNT_SEL5	Package	Uncore C-box 0 perfmon event select MSR.
D1BH	3355	MSR_C0_PMON_CTRL5	Package	Uncore C-box 0 perfmon counter MSR.
D20H	3360	MSR_C4_PMON_BOX_CTRL	Package	Uncore C-box 4 perfmon local box control MSR.
D21H	3361	MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon local box status MSR.
D22H	3362	MSR_C4_PMON_BOX_OVF_CTRL	Package	Uncore C-box 4 perfmon local box overflow control MSR.
D30H	3376	MSR_C4_PMON_EVNT_SELO	Package	Uncore C-box 4 perfmon event select MSR.
D31H	3377	MSR_C4_PMON_CTRL0	Package	Uncore C-box 4 perfmon counter MSR.
D32H	3378	MSR_C4_PMON_EVNT_SEL1	Package	Uncore C-box 4 perfmon event select MSR.
D33H	3379	MSR_C4_PMON_CTRL1	Package	Uncore C-box 4 perfmon counter MSR.
D34H	3380	MSR_C4_PMON_EVNT_SEL2	Package	Uncore C-box 4 perfmon event select MSR.
D35H	3381	MSR_C4_PMON_CTRL2	Package	Uncore C-box 4 perfmon counter MSR.
D36H	3382	MSR_C4_PMON_EVNT_SEL3	Package	Uncore C-box 4 perfmon event select MSR.
D37H	3383	MSR_C4_PMON_CTRL3	Package	Uncore C-box 4 perfmon counter MSR.
D38H	3384	MSR_C4_PMON_EVNT_SEL4	Package	Uncore C-box 4 perfmon event select MSR.
D39H	3385	MSR_C4_PMON_CTRL4	Package	Uncore C-box 4 perfmon counter MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
D3AH	3386	MSR_C4_PMON_EVNT_SEL5	Package	Uncore C-box 4 perfmon event select MSR.
D3BH	3387	MSR_C4_PMON_CTR5	Package	Uncore C-box 4 perfmon counter MSR.
D40H	3392	MSR_C2_PMON_BOX_CTRL	Package	Uncore C-box 2 perfmon local box control MSR.
D41H	3393	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon local box status MSR.
D42H	3394	MSR_C2_PMON_BOX_OVF_CTRL	Package	Uncore C-box 2 perfmon local box overflow control MSR.
D50H	3408	MSR_C2_PMON_EVNT_SELO	Package	Uncore C-box 2 perfmon event select MSR.
D51H	3409	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter MSR.
D52H	3410	MSR_C2_PMON_EVNT_SEL1	Package	Uncore C-box 2 perfmon event select MSR.
D53H	3411	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter MSR.
D54H	3412	MSR_C2_PMON_EVNT_SEL2	Package	Uncore C-box 2 perfmon event select MSR.
D55H	3413	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter MSR.
D56H	3414	MSR_C2_PMON_EVNT_SEL3	Package	Uncore C-box 2 perfmon event select MSR.
D57H	3415	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter MSR.
D58H	3416	MSR_C2_PMON_EVNT_SEL4	Package	Uncore C-box 2 perfmon event select MSR.
D59H	3417	MSR_C2_PMON_CTR4	Package	Uncore C-box 2 perfmon counter MSR.
D5AH	3418	MSR_C2_PMON_EVNT_SEL5	Package	Uncore C-box 2 perfmon event select MSR.
D5BH	3419	MSR_C2_PMON_CTR5	Package	Uncore C-box 2 perfmon counter MSR.
D60H	3424	MSR_C6_PMON_BOX_CTRL	Package	Uncore C-box 6 perfmon local box control MSR.
D61H	3425	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon local box status MSR.
D62H	3426	MSR_C6_PMON_BOX_OVF_CTRL	Package	Uncore C-box 6 perfmon local box overflow control MSR.
D70H	3440	MSR_C6_PMON_EVNT_SELO	Package	Uncore C-box 6 perfmon event select MSR.
D71H	3441	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter MSR.
D72H	3442	MSR_C6_PMON_EVNT_SEL1	Package	Uncore C-box 6 perfmon event select MSR.
D73H	3443	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter MSR.
D74H	3444	MSR_C6_PMON_EVNT_SEL2	Package	Uncore C-box 6 perfmon event select MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
D75H	3445	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter MSR.
D76H	3446	MSR_C6_PMON_EVNT_SEL3	Package	Uncore C-box 6 perfmon event select MSR.
D77H	3447	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter MSR.
D78H	3448	MSR_C6_PMON_EVNT_SEL4	Package	Uncore C-box 6 perfmon event select MSR.
D79H	3449	MSR_C6_PMON_CTR4	Package	Uncore C-box 6 perfmon counter MSR.
D7AH	3450	MSR_C6_PMON_EVNT_SEL5	Package	Uncore C-box 6 perfmon event select MSR.
D7BH	3451	MSR_C6_PMON_CTR5	Package	Uncore C-box 6 perfmon counter MSR.
D80H	3456	MSR_C1_PMON_BOX_CTRL	Package	Uncore C-box 1 perfmon local box control MSR.
D81H	3457	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon local box status MSR.
D82H	3458	MSR_C1_PMON_BOX_OVF_CTRL	Package	Uncore C-box 1 perfmon local box overflow control MSR.
D90H	3472	MSR_C1_PMON_EVNT_SELO	Package	Uncore C-box 1 perfmon event select MSR.
D91H	3473	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter MSR.
D92H	3474	MSR_C1_PMON_EVNT_SEL1	Package	Uncore C-box 1 perfmon event select MSR.
D93H	3475	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter MSR.
D94H	3476	MSR_C1_PMON_EVNT_SEL2	Package	Uncore C-box 1 perfmon event select MSR.
D95H	3477	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter MSR.
D96H	3478	MSR_C1_PMON_EVNT_SEL3	Package	Uncore C-box 1 perfmon event select MSR.
D97H	3479	MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter MSR.
D98H	3480	MSR_C1_PMON_EVNT_SEL4	Package	Uncore C-box 1 perfmon event select MSR.
D99H	3481	MSR_C1_PMON_CTR4	Package	Uncore C-box 1 perfmon counter MSR.
D9AH	3482	MSR_C1_PMON_EVNT_SEL5	Package	Uncore C-box 1 perfmon event select MSR.
D9BH	3483	MSR_C1_PMON_CTR5	Package	Uncore C-box 1 perfmon counter MSR.
DA0H	3488	MSR_C5_PMON_BOX_CTRL	Package	Uncore C-box 5 perfmon local box control MSR.
DA1H	3489	MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon local box status MSR.
DA2H	3490	MSR_C5_PMON_BOX_OVF_CTRL	Package	Uncore C-box 5 perfmon local box overflow control MSR.
DB0H	3504	MSR_C5_PMON_EVNT_SELO	Package	Uncore C-box 5 perfmon event select MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DB1H	3505	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter MSR.
DB2H	3506	MSR_C5_PMON_EVNT_SEL1	Package	Uncore C-box 5 perfmon event select MSR.
DB3H	3507	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter MSR.
DB4H	3508	MSR_C5_PMON_EVNT_SEL2	Package	Uncore C-box 5 perfmon event select MSR.
DB5H	3509	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter MSR.
DB6H	3510	MSR_C5_PMON_EVNT_SEL3	Package	Uncore C-box 5 perfmon event select MSR.
DB7H	3511	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter MSR.
DB8H	3512	MSR_C5_PMON_EVNT_SEL4	Package	Uncore C-box 5 perfmon event select MSR.
DB9H	3513	MSR_C5_PMON_CTR4	Package	Uncore C-box 5 perfmon counter MSR.
DBAH	3514	MSR_C5_PMON_EVNT_SEL5	Package	Uncore C-box 5 perfmon event select MSR.
DBBH	3515	MSR_C5_PMON_CTR5	Package	Uncore C-box 5 perfmon counter MSR.
DC0H	3520	MSR_C3_PMON_BOX_CTRL	Package	Uncore C-box 3 perfmon local box control MSR.
DC1H	3521	MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon local box status MSR.
DC2H	3522	MSR_C3_PMON_BOX_OVF_CTRL	Package	Uncore C-box 3 perfmon local box overflow control MSR.
DD0H	3536	MSR_C3_PMON_EVNT_SELO	Package	Uncore C-box 3 perfmon event select MSR.
DD1H	3537	MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter MSR.
DD2H	3538	MSR_C3_PMON_EVNT_SEL1	Package	Uncore C-box 3 perfmon event select MSR.
DD3H	3539	MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter MSR.
DD4H	3540	MSR_C3_PMON_EVNT_SEL2	Package	Uncore C-box 3 perfmon event select MSR.
DD5H	3541	MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter MSR.
DD6H	3542	MSR_C3_PMON_EVNT_SEL3	Package	Uncore C-box 3 perfmon event select MSR.
DD7H	3543	MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter MSR.
DD8H	3544	MSR_C3_PMON_EVNT_SEL4	Package	Uncore C-box 3 perfmon event select MSR.
DD9H	3545	MSR_C3_PMON_CTR4	Package	Uncore C-box 3 perfmon counter MSR.
DDAH	3546	MSR_C3_PMON_EVNT_SEL5	Package	Uncore C-box 3 perfmon event select MSR.
DDBH	3547	MSR_C3_PMON_CTR5	Package	Uncore C-box 3 perfmon counter MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DE0H	3552	MSR_C7_PMON_BOX_CTRL	Package	Uncore C-box 7 perfmon local box control MSR.
DE1H	3553	MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon local box status MSR.
DE2H	3554	MSR_C7_PMON_BOX_OVF_CTRL	Package	Uncore C-box 7 perfmon local box overflow control MSR.
DF0H	3568	MSR_C7_PMON_EVNT_SELO	Package	Uncore C-box 7 perfmon event select MSR.
DF1H	3569	MSR_C7_PMON_CTRL0	Package	Uncore C-box 7 perfmon counter MSR.
DF2H	3570	MSR_C7_PMON_EVNT_SEL1	Package	Uncore C-box 7 perfmon event select MSR.
DF3H	3571	MSR_C7_PMON_CTRL1	Package	Uncore C-box 7 perfmon counter MSR.
DF4H	3572	MSR_C7_PMON_EVNT_SEL2	Package	Uncore C-box 7 perfmon event select MSR.
DF5H	3573	MSR_C7_PMON_CTRL2	Package	Uncore C-box 7 perfmon counter MSR.
DF6H	3574	MSR_C7_PMON_EVNT_SEL3	Package	Uncore C-box 7 perfmon event select MSR.
DF7H	3575	MSR_C7_PMON_CTRL3	Package	Uncore C-box 7 perfmon counter MSR.
DF8H	3576	MSR_C7_PMON_EVNT_SEL4	Package	Uncore C-box 7 perfmon event select MSR.
DF9H	3577	MSR_C7_PMON_CTRL4	Package	Uncore C-box 7 perfmon counter MSR.
DFAH	3578	MSR_C7_PMON_EVNT_SEL5	Package	Uncore C-box 7 perfmon event select MSR.
DFBH	3579	MSR_C7_PMON_CTRL5	Package	Uncore C-box 7 perfmon counter MSR.
E00H	3584	MSR_R0_PMON_BOX_CTRL	Package	Uncore R-box 0 perfmon local box control MSR.
E01H	3585	MSR_R0_PMON_BOX_STATUS	Package	Uncore R-box 0 perfmon local box status MSR.
E02H	3586	MSR_R0_PMON_BOX_OVF_CTRL	Package	Uncore R-box 0 perfmon local box overflow control MSR.
E04H	3588	MSR_R0_PMON_IPERFO_P0	Package	Uncore R-box 0 perfmon IPERFO unit Port 0 select MSR.
E05H	3589	MSR_R0_PMON_IPERFO_P1	Package	Uncore R-box 0 perfmon IPERFO unit Port 1 select MSR.
E06H	3590	MSR_R0_PMON_IPERFO_P2	Package	Uncore R-box 0 perfmon IPERFO unit Port 2 select MSR.
E07H	3591	MSR_R0_PMON_IPERFO_P3	Package	Uncore R-box 0 perfmon IPERFO unit Port 3 select MSR.
E08H	3592	MSR_R0_PMON_IPERFO_P4	Package	Uncore R-box 0 perfmon IPERFO unit Port 4 select MSR.
E09H	3593	MSR_R0_PMON_IPERFO_P5	Package	Uncore R-box 0 perfmon IPERFO unit Port 5 select MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E0AH	3594	MSR_R0_PMON_IPERF0_P6	Package	Uncore R-box 0 perfmon IPERF0 unit Port 6 select MSR.
E0BH	3595	MSR_R0_PMON_IPERF0_P7	Package	Uncore R-box 0 perfmon IPERF0 unit Port 7 select MSR.
E0CH	3596	MSR_R0_PMON_QLX_P0	Package	Uncore R-box 0 perfmon QLX unit Port 0 select MSR.
E0DH	3597	MSR_R0_PMON_QLX_P1	Package	Uncore R-box 0 perfmon QLX unit Port 1 select MSR.
E0EH	3598	MSR_R0_PMON_QLX_P2	Package	Uncore R-box 0 perfmon QLX unit Port 2 select MSR.
E0FH	3599	MSR_R0_PMON_QLX_P3	Package	Uncore R-box 0 perfmon QLX unit Port 3 select MSR.
E10H	3600	MSR_R0_PMON_EVNT_SEL0	Package	Uncore R-box 0 perfmon event select MSR.
E11H	3601	MSR_R0_PMON_CTR0	Package	Uncore R-box 0 perfmon counter MSR.
E12H	3602	MSR_R0_PMON_EVNT_SEL1	Package	Uncore R-box 0 perfmon event select MSR.
E13H	3603	MSR_R0_PMON_CTR1	Package	Uncore R-box 0 perfmon counter MSR.
E14H	3604	MSR_R0_PMON_EVNT_SEL2	Package	Uncore R-box 0 perfmon event select MSR.
E15H	3605	MSR_R0_PMON_CTR2	Package	Uncore R-box 0 perfmon counter MSR.
E16H	3606	MSR_R0_PMON_EVNT_SEL3	Package	Uncore R-box 0 perfmon event select MSR.
E17H	3607	MSR_R0_PMON_CTR3	Package	Uncore R-box 0 perfmon counter MSR.
E18H	3608	MSR_R0_PMON_EVNT_SEL4	Package	Uncore R-box 0 perfmon event select MSR.
E19H	3609	MSR_R0_PMON_CTR4	Package	Uncore R-box 0 perfmon counter MSR.
E1AH	3610	MSR_R0_PMON_EVNT_SEL5	Package	Uncore R-box 0 perfmon event select MSR.
E1BH	3611	MSR_R0_PMON_CTR5	Package	Uncore R-box 0 perfmon counter MSR.
E1CH	3612	MSR_R0_PMON_EVNT_SEL6	Package	Uncore R-box 0 perfmon event select MSR.
E1DH	3613	MSR_R0_PMON_CTR6	Package	Uncore R-box 0 perfmon counter MSR.
E1EH	3614	MSR_R0_PMON_EVNT_SEL7	Package	Uncore R-box 0 perfmon event select MSR.
E1FH	3615	MSR_R0_PMON_CTR7	Package	Uncore R-box 0 perfmon counter MSR.
E20H	3616	MSR_R1_PMON_BOX_CTRL	Package	Uncore R-box 1 perfmon local box control MSR.
E21H	3617	MSR_R1_PMON_BOX_STATUS	Package	Uncore R-box 1 perfmon local box status MSR.
E22H	3618	MSR_R1_PMON_BOX_OVF_CTRL	Package	Uncore R-box 1 perfmon local box overflow control MSR.
E24H	3620	MSR_R1_PMON_IPERF1_P8	Package	Uncore R-box 1 perfmon IPERF1 unit Port 8 select MSR.



Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E25H	3621	MSR_R1_PMON_IPERF1_P9	Package	Uncore R-box 1 perfmon IPERF1 unit Port 9 select MSR.
E26H	3622	MSR_R1_PMON_IPERF1_P10	Package	Uncore R-box 1 perfmon IPERF1 unit Port 10 select MSR.
E27H	3623	MSR_R1_PMON_IPERF1_P11	Package	Uncore R-box 1 perfmon IPERF1 unit Port 11 select MSR.
E28H	3624	MSR_R1_PMON_IPERF1_P12	Package	Uncore R-box 1 perfmon IPERF1 unit Port 12 select MSR.
E29H	3625	MSR_R1_PMON_IPERF1_P13	Package	Uncore R-box 1 perfmon IPERF1 unit Port 13 select MSR.
E2AH	3626	MSR_R1_PMON_IPERF1_P14	Package	Uncore R-box 1 perfmon IPERF1 unit Port 14 select MSR.
E2BH	3627	MSR_R1_PMON_IPERF1_P15	Package	Uncore R-box 1 perfmon IPERF1 unit Port 15 select MSR.
E2CH	3628	MSR_R1_PMON_QLX_P4	Package	Uncore R-box 1 perfmon QLX unit Port 4 select MSR.
E2DH	3629	MSR_R1_PMON_QLX_P5	Package	Uncore R-box 1 perfmon QLX unit Port 5 select MSR.
E2EH	3630	MSR_R1_PMON_QLX_P6	Package	Uncore R-box 1 perfmon QLX unit Port 6 select MSR.
E2FH	3631	MSR_R1_PMON_QLX_P7	Package	Uncore R-box 1 perfmon QLX unit Port 7 select MSR.
E30H	3632	MSR_R1_PMON_EVNT_SEL8	Package	Uncore R-box 1 perfmon event select MSR.
E31H	3633	MSR_R1_PMON_CTR8	Package	Uncore R-box 1 perfmon counter MSR.
E32H	3634	MSR_R1_PMON_EVNT_SEL9	Package	Uncore R-box 1 perfmon event select MSR.
E33H	3635	MSR_R1_PMON_CTR9	Package	Uncore R-box 1 perfmon counter MSR.
E34H	3636	MSR_R1_PMON_EVNT_SEL10	Package	Uncore R-box 1 perfmon event select MSR.
E35H	3637	MSR_R1_PMON_CTR10	Package	Uncore R-box 1 perfmon counter MSR.
E36H	3638	MSR_R1_PMON_EVNT_SEL11	Package	Uncore R-box 1 perfmon event select MSR.
E37H	3639	MSR_R1_PMON_CTR11	Package	Uncore R-box 1 perfmon counter MSR.
E38H	3640	MSR_R1_PMON_EVNT_SEL12	Package	Uncore R-box 1 perfmon event select MSR.
E39H	3641	MSR_R1_PMON_CTR12	Package	Uncore R-box 1 perfmon counter MSR.
E3AH	3642	MSR_R1_PMON_EVNT_SEL13	Package	Uncore R-box 1 perfmon event select MSR.
E3BH	3643	MSR_R1_PMON_CTR13	Package	Uncore R-box 1 perfmon counter MSR.
E3CH	3644	MSR_R1_PMON_EVNT_SEL14	Package	Uncore R-box 1 perfmon event select MSR.
E3DH	3645	MSR_R1_PMON_CTR14	Package	Uncore R-box 1 perfmon counter MSR.

**Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E3EH	3646	MSR_R1_PMON_EVT_SEL15	Package	Uncore R-box 1 perfmon event select MSR.
E3FH	3647	MSR_R1_PMON_CTR15	Package	Uncore R-box 1 perfmon counter MSR.
E45H	3653	MSR_B0_PMON_MATCH	Package	Uncore B-box 0 perfmon local box match MSR.
E46H	3654	MSR_B0_PMON_MASK	Package	Uncore B-box 0 perfmon local box mask MSR.
E49H	3657	MSR_S0_PMON_MATCH	Package	Uncore S-box 0 perfmon local box match MSR.
E4AH	3658	MSR_S0_PMON_MASK	Package	Uncore S-box 0 perfmon local box mask MSR.
E4DH	3661	MSR_B1_PMON_MATCH	Package	Uncore B-box 1 perfmon local box match MSR.
E4EH	3662	MSR_B1_PMON_MASK	Package	Uncore B-box 1 perfmon local box mask MSR.
E54H	3668	MSR_M0_PMON_MM_CONFIG	Package	Uncore M-box 0 perfmon local box address match/mask config MSR.
E55H	3669	MSR_M0_PMON_ADDR_MATCH	Package	Uncore M-box 0 perfmon local box address match MSR.
E56H	3670	MSR_M0_PMON_ADDR_MASK	Package	Uncore M-box 0 perfmon local box address mask MSR.
E59H	3673	MSR_S1_PMON_MATCH	Package	Uncore S-box 1 perfmon local box match MSR.
E5AH	3674	MSR_S1_PMON_MASK	Package	Uncore S-box 1 perfmon local box mask MSR.
E5CH	3676	MSR_M1_PMON_MM_CONFIG	Package	Uncore M-box 1 perfmon local box address match/mask config MSR.
E5DH	3677	MSR_M1_PMON_ADDR_MATCH	Package	Uncore M-box 1 perfmon local box address match MSR.
E5EH	3678	MSR_M1_PMON_ADDR_MASK	Package	Uncore M-box 1 perfmon local box address mask MSR.
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

## 2.9 MSRS IN THE INTEL® XEON® PROCESSOR 5600 SERIES (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor 5600 Series (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 2-15, Table 2-16, plus additional MSR listed in Table 2-18. These MSRs apply to Intel Core i7, i5 and i3 processor family with CPUID signature DisplayFamily\_DisplayModel of 06\_25H and 06\_2CH, see Table 2-1.

**Table 2-18. Additional MSRs Supported by Intel Processors  
(Based on Intel® Microarchitecture Code Name Westmere)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved
1A7H	423	MSR_OFFCORE_RSP_1	Thread	Offcore Response Event Select Register (R/W)
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		39:32	Package	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 core active.
		47:40	Package	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 core active.
		63:48		Reserved
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.

## 2.10 MSRS IN THE INTEL® XEON® PROCESSOR E7 FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor E7 Family (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 2-15 (except MSR address 1ADH), Table 2-16, plus additional MSR listed in Table 2-19. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2FH.

Table 2-19. Additional MSRs Supported by Intel® Xeon® Processor E7 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved
1A7H	423	MSR_OFFCORE_RSP_1	Thread	Offcore Response Event Select Register (R/W)
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Reserved Attempt to read/write will cause #UD.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.
F40H	3904	MSR_C8_PMON_BOX_CTRL	Package	Uncore C-box 8 perfmon local box control MSR.
F41H	3905	MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon local box status MSR.
F42H	3906	MSR_C8_PMON_BOX_OVF_CTRL	Package	Uncore C-box 8 perfmon local box overflow control MSR.
F50H	3920	MSR_C8_PMON_EVNT_SELO	Package	Uncore C-box 8 perfmon event select MSR.
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR.
F52H	3922	MSR_C8_PMON_EVNT_SEL1	Package	Uncore C-box 8 perfmon event select MSR.
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR.
F54H	3924	MSR_C8_PMON_EVNT_SEL2	Package	Uncore C-box 8 perfmon event select MSR.
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR.
F56H	3926	MSR_C8_PMON_EVNT_SEL3	Package	Uncore C-box 8 perfmon event select MSR.
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR.
F58H	3928	MSR_C8_PMON_EVNT_SEL4	Package	Uncore C-box 8 perfmon event select MSR.
F59H	3929	MSR_C8_PMON_CTR4	Package	Uncore C-box 8 perfmon counter MSR.
F5AH	3930	MSR_C8_PMON_EVNT_SEL5	Package	Uncore C-box 8 perfmon event select MSR.
F5BH	3931	MSR_C8_PMON_CTR5	Package	Uncore C-box 8 perfmon counter MSR.

**Table 2-19. Additional MSRs Supported by Intel® Xeon® Processor E7 Family (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
FC0H	4032	MSR_C9_PMON_BOX_CTRL	Package	Uncore C-box 9 perfmon local box control MSR.
FC1H	4033	MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon local box status MSR.
FC2H	4034	MSR_C9_PMON_BOX_OVF_CTRL	Package	Uncore C-box 9 perfmon local box overflow control MSR.
FD0H	4048	MSR_C9_PMON_EVNT_SELO	Package	Uncore C-box 9 perfmon event select MSR.
FD1H	4049	MSR_C9_PMON_CTRL0	Package	Uncore C-box 9 perfmon counter MSR.
FD2H	4050	MSR_C9_PMON_EVNT_SEL1	Package	Uncore C-box 9 perfmon event select MSR.
FD3H	4051	MSR_C9_PMON_CTRL1	Package	Uncore C-box 9 perfmon counter MSR.
FD4H	4052	MSR_C9_PMON_EVNT_SEL2	Package	Uncore C-box 9 perfmon event select MSR.
FD5H	4053	MSR_C9_PMON_CTRL2	Package	Uncore C-box 9 perfmon counter MSR.
FD6H	4054	MSR_C9_PMON_EVNT_SEL3	Package	Uncore C-box 9 perfmon event select MSR.
FD7H	4055	MSR_C9_PMON_CTRL3	Package	Uncore C-box 9 perfmon counter MSR.
FD8H	4056	MSR_C9_PMON_EVNT_SEL4	Package	Uncore C-box 9 perfmon event select MSR.
FD9H	4057	MSR_C9_PMON_CTRL4	Package	Uncore C-box 9 perfmon counter MSR.
FDAH	4058	MSR_C9_PMON_EVNT_SEL5	Package	Uncore C-box 9 perfmon event select MSR.
FDBH	4059	MSR_C9_PMON_CTRL5	Package	Uncore C-box 9 perfmon counter MSR.

## 2.11 MSRS IN INTEL® PROCESSOR FAMILY BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Table 2-20 lists model-specific registers (MSRs) that are common to Intel® processor family based on Intel micro-architecture code name Sandy Bridge. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2AH, 06\_2DH, see Table 2-1. Additional MSRs specific to 06\_2AH are listed in Table 2-21.

**Table 2-20. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.23, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.23, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, “Monitor/Mwait Address Range Determination” and Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "Time-Stamp Counter" and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	Platform ID (R) See Table 2-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location" and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	SMI Counter (R/O)
		31:0		SMI Count (R/O) Count SMIs.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Enable VMX Inside SMX Operation (R/WL)
		2		Enable VMX Outside SMX Operation (R/WL)
		14:8		SENTER Local Functions Enables (R/WL)
15		SENTER Global Functions Enable (R/WL)		
79H	121	IA32_BIOS_UPDT_TRIG	Core	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Thread	BIOS Update Signature ID (RO) See Table 2-2.
C1H	193	IA32_PMC0	Thread	Performance Counter Register See Table 2-2.
C2H	194	IA32_PMC1	Thread	Performance Counter Register See Table 2-2.
C3H	195	IA32_PMC2	Thread	Performance Counter Register See Table 2-2.
C4H	196	IA32_PMC3	Thread	Performance Counter Register See Table 2-2.
C5H	197	IA32_PMC4	Core	Performance Counter Register (if core not shared by threads)
C6H	198	IA32_PMC5	Core	Performance Counter Register (if core not shared by threads)
C7H	199	IA32_PMC6	Core	Performance Counter Register (if core not shared by threads)
C8H	200	IA32_PMC7	Core	Performance Counter Register (if core not shared by threads)

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.  The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (R/W0) When set, locks bits 15:0 of this register until next reset.
		24:16		Reserved
		25		C3 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		C1 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		Enable C3 Undemotion (R/W) When set, enables undemotion from demoted C3.
		28		Enable C1 Undemotion (R/W) When set, enables undemotion from demoted C1.
		63:29		Reserved
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	Power Management IO Redirection in C-state (R/W) See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		C-State Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - C6 is the max C-State to include. 010b - C7 is the max C-State to include.
		63:19		Reserved
E7H	231	IA32_MPERF	Thread	Maximum Performance Frequency Clock Count (RW) See Table 2-2.



**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E8H	232	IA32_APERF	Thread	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
63:3		Reserved		
186H	390	IA32_PERFVTSELO	Thread	See Table 2-2.
187H	391	IA32_PERFVTSEL1	Thread	See Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
188H	392	IA32_PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Thread	See Table 2-2.
18AH	394	IA32_PERFEVTSEL4	Core	See Table 2-2. If CPUID.0AH:EAX[15:8] = 8.
18BH	395	IA32_PERFEVTSEL5	Core	See Table 2-2. If CPUID.0AH:EAX[15:8] = 8.
18CH	396	IA32_PERFEVTSEL6	Core	See Table 2-2. If CPUID.0AH:EAX[15:8] = 8.
18DH	397	IA32_PERFEVTSEL7	Core	See Table 2-2. If CPUID.0AH:EAX[15:8] = 8.
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
		15:0		Current Performance State Value
		63:16		Reserved
198H	408	MSR_PERF_STATUS	Package	Performance Status
		47:32		Core Voltage (R/O) P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2 <sup>13</sup> ).
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		3:0		On demand Clock Modulation Duty Cycle (R/W) In 6.25% increment.
		4		On demand Clock Modulation Enable (R/W)
		63:5		Reserved
19BH	411	IA32_THERM_INTERRUPT	Core	Thermal Interrupt Control (R/W) See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
		0		Thermal Status (RO) See Table 2-2.
		1		Thermal Status Log (R/WCO) See Table 2-2.
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		6		Thermal Threshold #1 Status (RO) See Table 2-2.
		7		Thermal Threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
		11		Power Limitation Log (R/WCO) See Table 2-2.
		15:12		Reserved
		22:16		Digital Readout (RO) See Table 2-2.
		26:23		Reserved
		30:27		Resolution in Degrees Celsius (RO) See Table 2-2.
		31		Reading Valid (RO) See Table 2-2.
		63:32		Reserved
1A0H	416	IA32_MISC_ENABLE		Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		0	Thread	Fast-Strings Enable See Table 2-2
		6:1		Reserved
		7	Thread	Performance Monitoring Available (R) See Table 2-2.
		10:8		Reserved
		11	Thread	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12	Thread	Processor Event Based Sampling Unavailable (RO) See Table 2-2.
		15:13		Reserved
		16	Package	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		18	Thread	ENABLE MONITOR FSM (R/W) See Table 2-2.
		21:19		Reserved
		22	Thread	Limit CPUID Maxval (R/W) See Table 2-2.
		23	Thread	xTPR Message Disable (R/W) See Table 2-2.
		33:24		Reserved
		34	Thread	XD Bit Disable (R/W) See Table 2-2.
		37:35		Reserved
		38	Package	Turbo Mode Disable (R/W) When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.
		63:39		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Unique	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R) The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.
		63:24		Reserved
1A4H	420	MSR_MISC_FEATURE_CONTROL		Miscellaneous Feature Control (R/W)
		0	Core	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	L2 Adjacent Cache Line Prefetcher Disable (R/W) If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2	Core	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	DCU IP Prefetcher Disable (R/W) If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved
1A6H	422	MSR_OFFCORE_RSP_0	Thread	Offcore Response Event Select Register (R/W)
1A7H	422	MSR_OFFCORE_RSP_1	Thread	Offcore Response Event Select Register (R/W)
1AAH	426	MSR_MISC_PWR_MGMT		Miscellaneous Power Management Control Various model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
1BOH	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	Last Branch Record Filtering Select Register (R/W) See Section 17.9.2, "Filtering of Last Branch Records."
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
63:9		Reserved		
1C9H	457	MSR_LASTBRANCH_TOS	Thread	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	Debug Control (R/W) See Table 2-2.
		0		LBR: Last Branch Record
		1		BTF
		5:2		Reserved
		6		TR: Branch Trace

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7		BTS: Log Branch Trace Message to BTS buffer
		8		BTINT
		9		BTS_OFF_OS
		10		BTS_OFF_USER
		11		FREEZE_LBR_ON_PMI
		12		FREEZE_PERFMON_ON_PMI
		13		ENABLE_UNCORE_PMI
		14		FREEZE_WHILE_SMM
		63:15		Reserved
1DDH	477	MSR_LER_FROM_LIP	Thread	Last Exception Record From Linear IP (R/W) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	Last Exception Record To Linear IP (R/W) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	See <a href="http://biosbits.org">http://biosbits.org</a> .
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Thread	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	Always 0 (CMCI not supported).
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	Default Memory Types (R/W) See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format See Table 2-2.
		6		PEBS Record Format.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7		PEBSSaveArchRegs See Table 2-2.
		11:8		PEBS_REC_FORMAT See Table 2-2.
		12		SMM_FREEZE See Table 2-2.
		63:13		Reserved
38DH	909	IA32_FIXED_CTR_CTRL	Thread	Fixed-Function-Counter Control Register (R/W) See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS		See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	Ovf_PMC0
		1	Thread	Ovf_PMC1
		2	Thread	Ovf_PMC2
		3	Thread	Ovf_PMC3
		4	Core	Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)
		5	Core	Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)
		6	Core	Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)
		7	Core	Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved
		32	Thread	Ovf_FixedCtr0
		33	Thread	Ovf_FixedCtr1
		34	Thread	Ovf_FixedCtr2
		60:35		Reserved
		61	Thread	Ovf_Uncore
		62	Thread	Ovf_BufDSSAVE
63	Thread	CondChgd		
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	Set 1 to enable PMC0 to count.
		1	Thread	Set 1 to enable PMC1 to count.
		2	Thread	Set 1 to enable PMC2 to count.
		3	Thread	Set 1 to enable PMC3 to count.
		4	Core	Set 1 to enable PMC4 to count (if CPUID.0AH:EAX[15:8] > 4).
		5	Core	Set 1 to enable PMC5 to count (if CPUID.0AH:EAX[15:8] > 5).



**Table 2-20. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		6	Core	Set 1 to enable PMC6 to count (if CPUID.0AH:EAX[15:8] > 6).
		7	Core	Set 1 to enable PMC7 to count (if CPUID.0AH:EAX[15:8] > 7).
		31:8		Reserved
		32	Thread	Set 1 to enable FixedCtr0 to count.
		33	Thread	Set 1 to enable FixedCtr1 to count.
		34	Thread	Set 1 to enable FixedCtr2 to count.
		63:35		Reserved
390H	912	IA32_PERF_GLOBAL_OVF_CTRL		See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	Set 1 to clear Ovf_PMC0.
		1	Thread	Set 1 to clear Ovf_PMC1.
		2	Thread	Set 1 to clear Ovf_PMC2.
		3	Thread	Set 1 to clear Ovf_PMC3.
		4	Core	Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4).
		5	Core	Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5).
		6	Core	Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6).
		7	Core	Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7).
		31:8		Reserved
		32	Thread	Set 1 to clear Ovf_FixedCtr0.
		33	Thread	Set 1 to clear Ovf_FixedCtr1.
		34	Thread	Set 1 to clear Ovf_FixedCtr2.
		60:35		Reserved
		3F1H	1009	MSR_PEBS_ENABLE
0				Enable PEBS on IA32_PMC0. (R/W)
1				Enable PEBS on IA32_PMC1. (R/W)
2				Enable PEBS on IA32_PMC2. (R/W)
3				Enable PEBS on IA32_PMC3. (R/W)
31:4				Reserved
32				Enable Load Latency on IA32_PMC0. (R/W)
33				Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		35		Enable Load Latency on IA32_PMC3. (R/W)
		62:36		Reserved
		63		Enable Precise Store (R/W)
3F6H	1014	MSR_PEBS_LD_LAT	Thread	See Section 18.3.1.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3FEH	1022	MSR_CORE_C7_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C7 Residency Counter (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.
400H	1024	IA32_MCO_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
402H	1026	IA32_MCO_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
403H	1027	IA32_MCO_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
		0		PCU Hardware Error (R/W) When set, enables signaling of PCU hardware detected errors.
		1		PCU Controller Error (R/W) When set, enables signaling of PCU controller detected errors.
		2		PCU Firmware Error (R/W) When set, enables signaling of PCU firmware detected errors.
		63:2		Reserved
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
480H	1152	IA32_VMX_BASIC	Thread	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLs	Thread	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLs	Thread	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLs	Thread	Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLs	Thread	Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Thread	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	Thread	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Thread	Capability Reporting Register of EPT and VPID (R/O) See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLSS	Thread	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLSS	Thread	Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLSS	Thread	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLSS	Thread	Capability Reporting Register of VM-Entry Flex Controls (R/O) See Table 2-2
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
4C3H	1219	IA32_A_PMC2	Thread	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Thread	See Table 2-2.
4C5H	1221	IA32_A_PMC4	Core	See Table 2-2.
4C6H	1222	IA32_A_PMC5	Core	See Table 2-2.
4C7H	1223	IA32_A_PMC6	Core	See Table 2-2.
4C8H	1224	IA32_A_PMC7	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers used in RAPL Interfaces (R/O) See Section 14.10.1, "RAPL Interfaces."
60AH	1546	MSR_PKGC3_IRTL	Package	Package C3 Interrupt Response Limit (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C3 state.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved
60BH	1547	MSR_PKGC6_IRTL	Package	Package C6 Interrupt Response Limit (R/W) This MSR defines the budget allocated for the package to exit from a C6 to a C0 state, where an interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:16		Reserved
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C2 Residency Counter (R/O) Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	PKG RAPL Power Limit Control (R/W) See Section 14.10.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameters (R/W) See Section 14.10.3, "Package RAPL Domain."
638H	1592	MSR_PPO_POWER_LIMIT	Package	PPO RAPL Power Limit Control (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	Last Branch Record 0 From IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.9.1 and record format in Section 17.4.8.1.</li> </ul>
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Thread	Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Thread	Last Branch Record 8 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Thread	Last Branch Record 9 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.

**Table 2-20. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Thread	Last Branch Record 10 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	Thread	Last Branch Record 11 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	Thread	Last Branch Record 12 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	Thread	Last Branch Record 13 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	Thread	Last Branch Record 14 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	Thread	Last Branch Record 15 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Thread	Last Branch Record 0 To IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	Thread	Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	Thread	Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	Thread	Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	Thread	Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	Thread	Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	Thread	Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	Thread	Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	Thread	Last Branch Record 8 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	Thread	Last Branch Record 9 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	Thread	Last Branch Record 10 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	Thread	Last Branch Record 11 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.



**Table 2-20. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	Thread	Last Branch Record 12 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	Thread	Last Branch Record 13 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	Thread	Last Branch Record 14 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	Thread	Last Branch Record 15 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table 2-2.
802H-83FH	2050-2111	X2APIC MSRs	Thread	See Table 2-2.
C000_0080H		IA32_EFER	Thread	Extended Feature Enables See Table 2-2.
C000_0081H		IA32_STAR	Thread	System Call Target Address (R/W) See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	IA-32e Mode System Call Target Address (R/W) See Table 2-2.
C000_0084H		IA32_FMASK	Thread	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	Map of BASE Address of FS (R/W) See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	Swap Target of BASE Address of GS (R/W) See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	AUXILIARY TSC Signature (R/W) See Table 2-2 and Section 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

### 2.11.1 MSRs In 2nd Generation Intel® Core™ Processor Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-21 and Table 2-22 list model-specific registers (MSRs) that are specific to the 2nd generation Intel® Core™ processor family (based on Intel microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2AH; see Table 2-1.

**Table 2-21. MSRs Supported by 2nd Generation Intel® Core™ Processors  
(Intel® microarchitecture code name Sandy Bridge)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		63:32		Reserved
60CH	1548	MSR_PKGC7_IRTL	Package	Package C7 Interrupt Response Limit (R/W) This MSR defines the budget allocated for the package to exit from a C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved

**Table 2-21. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
63AH	1594	MSR_PP0_POLICY	Package	PP0 Balance Policy (R/W) See Section 14.10.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	PP1 RAPL Power Limit Control (R/W) See Section 14.10.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	PP1 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	PP1 Balance Policy (R/W) See Section 14.10.4, "PP0/PP1 RAPL Domains."

See Table 2-20, Table 2-21, and Table 2-22 for MSR definitions applicable to processors with CPUID signature 06\_2AH.

Table 2-22 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06\_2AH.

**Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU Global Control
		0		Slice 0 select.
		1		Slice 1 select.
		2		Slice 2 select.
		3		Slice 3 select.
		4		Slice 4 select.
		18:5		Reserved
		29		Enable all uncore counters.
		30		Enable wake on PMI.
		31		Enable Freezing counter when overflow.
	63:32	Reserved		
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU Main Status
		0		Fixed counter overflowed.
		1		An ARB counter overflowed.
		2		Reserved
		3		A CBox counter overflowed (on any slice).
	63:4	Reserved		
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore Fixed Counter Control (R/W)
		19:0		Reserved
		20		Enable overflow propagation.

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		21		Reserved
		22		Enable counting.
		63:23		Reserved
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore Fixed Counter
		47:0		Current count.
		63:48		Reserved
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box Configuration Information (R/O)
		3:0		Report the number of C-Box units with performance counters, including processor cores and processor graphics.
		63:4		Reserved
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb Unit, Performance Counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb Unit, Performance Counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb Unit, Counter 0 Event Select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, Counter 1 Event Select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSELO	Package	Uncore C-Box 0, Counter 0 Event Select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, Counter 1 Event Select MSR
702H	1794	MSR_UNC_CBO_0_PERFEVTSEL2	Package	Uncore C-Box 0, Counter 2 Event Select MSR
703H	1795	MSR_UNC_CBO_0_PERFEVTSEL3	Package	Uncore C-Box 0, Counter 3 Event Select MSR
705H	1797	MSR_UNC_CBO_0_UNIT_STATUS	Package	Uncore C-Box 0, Unit Status for Counter 0-3
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, Performance Counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, Performance Counter 1
708H	1800	MSR_UNC_CBO_0_PERFCTR2	Package	Uncore C-Box 0, Performance Counter 2
709H	1801	MSR_UNC_CBO_0_PERFCTR3	Package	Uncore C-Box 0, Performance Counter 3
710H	1808	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, Counter 0 Event Select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, Counter 1 Event Select MSR
712H	1810	MSR_UNC_CBO_1_PERFEVTSEL2	Package	Uncore C-Box 1, Counter 2 Event Select MSR
713H	1811	MSR_UNC_CBO_1_PERFEVTSEL3	Package	Uncore C-Box 1, Counter 3 Event Select MSR
715H	1813	MSR_UNC_CBO_1_UNIT_STATUS	Package	Uncore C-Box 1, Unit Status for Counter 0-3
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, Performance Counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, Performance Counter 1
718H	1816	MSR_UNC_CBO_1_PERFCTR2	Package	Uncore C-Box 1, Performance Counter 2
719H	1817	MSR_UNC_CBO_1_PERFCTR3	Package	Uncore C-Box 1, Performance Counter 3
720H	1824	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, Counter 0 Event Select MSR
721H	1825	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, Counter 1 Event Select MSR
722H	1826	MSR_UNC_CBO_2_PERFEVTSEL2	Package	Uncore C-Box 2, Counter 2 Event Select MSR

**Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
723H	1827	MSR_UNC_CBO_2_PERFEVTSEL3	Package	Uncore C-Box 2, Counter 3 Event Select MSR
725H	1829	MSR_UNC_CBO_2_UNIT_STATUS	Package	Uncore C-Box 2, Unit Status for Counter 0-3
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, Performance Counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, Performance Counter 1
728H	1832	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, Performance Counter 2
729H	1833	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, Performance Counter 3
730H	1840	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, Counter 0 Event Select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, Counter 1 Event Select MSR
732H	1842	MSR_UNC_CBO_3_PERFEVTSEL2	Package	Uncore C-Box 3, Counter 2 Event Select MSR
733H	1843	MSR_UNC_CBO_3_PERFEVTSEL3	Package	Uncore C-Box 3, counter 3 Event Select MSR
735H	1845	MSR_UNC_CBO_3_UNIT_STATUS	Package	Uncore C-Box 3, Unit Status for Counter 0-3
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, Performance Counter 0
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, Performance Counter 1
738H	1848	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, Performance Counter 2
739H	1849	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, Performance Counter 3
740H	1856	MSR_UNC_CBO_4_PERFEVTSELO	Package	Uncore C-Box 4, Counter 0 Event Select MSR
741H	1857	MSR_UNC_CBO_4_PERFEVTSEL1	Package	Uncore C-Box 4, Counter 1 Event Select MSR
742H	1858	MSR_UNC_CBO_4_PERFEVTSEL2	Package	Uncore C-Box 4, Counter 2 Event Select MSR
743H	1859	MSR_UNC_CBO_4_PERFEVTSEL3	Package	Uncore C-Box 4, Counter 3 Event Select MSR
745H	1861	MSR_UNC_CBO_4_UNIT_STATUS	Package	Uncore C-Box 4, Unit status for Counter 0-3
746H	1862	MSR_UNC_CBO_4_PERFCTR0	Package	Uncore C-Box 4, Performance Counter 0
747H	1863	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, Performance Counter 1
748H	1864	MSR_UNC_CBO_4_PERFCTR2	Package	Uncore C-Box 4, Performance Counter 2
749H	1865	MSR_UNC_CBO_4_PERFCTR3	Package	Uncore C-Box 4, Performance Counter 3

### 2.11.2 MSRs In Intel® Xeon® Processor E5 Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-23 lists additional model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 Family (based on Intel® microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2DH, and also supports MSRs listed in Table 2-20 and Table 2-24.

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
17FH	383	MSR_ERROR_CONTROL	Package	MC Bank Error Configuration (R/W)
		0		Reserved

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family  
(based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		MemError Log Enable (R/W) When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 cores active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 cores active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 cores active.
		39:32	Package	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 cores active.
		47:40	Package	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 cores active.
		55:48	Package	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 cores active.
		63:56	Package	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 cores active.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family  
(based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
39CH	924	MSR_PEBS_NUM_ALT	Package	ENABLE_PEBS_NUM_ALT (RW)
		0		ENABLE_PEBS_NUM_ALT (RW) Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a> .
		63:1		Reserved, must be zero.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs" and Chapter 16.

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family  
(based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."



**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
613H	1555	MSR_PKG_PERF_STATUS	Package	Package RAPL Perf Status (R/O)
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."

See Table 2-20, Table 2-23, and Table 2-24 for MSR definitions applicable to processors with CPUID signature 06\_2DH.

### 2.11.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 Family

Intel Xeon Processor E5 family is based on the Sandy Bridge microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2DH

**Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C08H	3080	MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK Fixed Counter Control
C09H	3081	MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK Fixed Counter
C10H	3088	MSR_U_PMON_EVNTSELO	Package	Uncore U-box Perfmon Event Select for U-box Counter 0
C11H	3089	MSR_U_PMON_EVNTSEL1	Package	Uncore U-box Perfmon Event Select for U-box Counter 1
C16H	3094	MSR_U_PMON_CTR0	Package	Uncore U-box Perfmon Counter 0
C17H	3095	MSR_U_PMON_CTR1	Package	Uncore U-box Perfmon Counter 1
C24H	3108	MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU Perfmon for PCU-box-wide Control
C30H	3120	MSR_PCU_PMON_EVNTSELO	Package	Uncore PCU Perfmon Event Select for PCU Counter 0
C31H	3121	MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU Perfmon Event Select for PCU Counter 1
C32H	3122	MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU Perfmon Event Select for PCU Counter 2

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C33H	3123	MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU Perfmon Event Select for PCU Counter 3
C34H	3124	MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU Perfmon box-wide Filter
C36H	3126	MSR_PCU_PMON_CTR0	Package	Uncore PCU Perfmon Counter 0
C37H	3127	MSR_PCU_PMON_CTR1	Package	Uncore PCU Perfmon Counter 1
C38H	3128	MSR_PCU_PMON_CTR2	Package	Uncore PCU Perfmon Counter 2
C39H	3129	MSR_PCU_PMON_CTR3	Package	Uncore PCU Perfmon Counter 3
D04H	3332	MSR_CO_PMON_BOX_CTL	Package	Uncore C-box 0 Perfmon Local Box Wide Control
D10H	3344	MSR_CO_PMON_EVNTSEL0	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 0
D11H	3345	MSR_CO_PMON_EVNTSEL1	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 1
D12H	3346	MSR_CO_PMON_EVNTSEL2	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 2
D13H	3347	MSR_CO_PMON_EVNTSEL3	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 3
D14H	3348	MSR_CO_PMON_BOX_FILTER	Package	Uncore C-box 0 Perfmon Box Wide Filter
D16H	3350	MSR_CO_PMON_CTR0	Package	Uncore C-box 0 Perfmon Counter 0
D17H	3351	MSR_CO_PMON_CTR1	Package	Uncore C-box 0 Perfmon Counter 1
D18H	3352	MSR_CO_PMON_CTR2	Package	Uncore C-box 0 Perfmon Counter 2
D19H	3353	MSR_CO_PMON_CTR3	Package	Uncore C-box 0 Perfmon Counter 3
D24H	3364	MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 Perfmon Local Box Wide Control
D30H	3376	MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 0
D31H	3377	MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 1
D32H	3378	MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 2
D33H	3379	MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 3
D34H	3380	MSR_C1_PMON_BOX_FILTER	Package	Uncore C-box 1 Perfmon Box Wide Filter
D36H	3382	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 Perfmon Counter 0
D37H	3383	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 Perfmon Counter 1
D38H	3384	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 Perfmon Counter 2
D39H	3385	MSR_C1_PMON_CTR3	Package	Uncore C-box 1 Perfmon Counter 3
D44H	3396	MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 Perfmon Local Box Wide Control
D50H	3408	MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 0
D51H	3409	MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 1

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
D52H	3410	MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 2
D53H	3411	MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 3
D54H	3412	MSR_C2_PMON_BOX_FILTER	Package	Uncore C-box 2 Perfmon Box Wide Filter
D56H	3414	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 Perfmon Counter 0
D57H	3415	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 Perfmon Counter 1
D58H	3416	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 Perfmon Counter 2
D59H	3417	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 Perfmon Counter 3
D64H	3428	MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 Perfmon Local Box Wide Control
D70H	3440	MSR_C3_PMON_EVNTSELO	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 0
D71H	3441	MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 1
D72H	3442	MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 2
D73H	3443	MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 3
D74H	3444	MSR_C3_PMON_BOX_FILTER	Package	Uncore C-box 3 Perfmon Box Wide Filter
D76H	3446	MSR_C3_PMON_CTR0	Package	Uncore C-box 3 Perfmon Counter 0
D77H	3447	MSR_C3_PMON_CTR1	Package	Uncore C-box 3 Perfmon Counter 1
D78H	3448	MSR_C3_PMON_CTR2	Package	Uncore C-box 3 Perfmon Counter 2
D79H	3449	MSR_C3_PMON_CTR3	Package	Uncore C-box 3 Perfmon Counter 3
D84H	3460	MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 Perfmon Local Box Wide Control
D90H	3472	MSR_C4_PMON_EVNTSELO	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 0
D91H	3473	MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 1
D92H	3474	MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 2
D93H	3475	MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 3
D94H	3476	MSR_C4_PMON_BOX_FILTER	Package	Uncore C-box 4 Perfmon Box Wide Filter
D96H	3478	MSR_C4_PMON_CTR0	Package	Uncore C-box 4 Perfmon Counter 0
D97H	3479	MSR_C4_PMON_CTR1	Package	Uncore C-box 4 Perfmon Counter 1
D98H	3480	MSR_C4_PMON_CTR2	Package	Uncore C-box 4 Perfmon Counter 2
D99H	3481	MSR_C4_PMON_CTR3	Package	Uncore C-box 4 Perfmon Counter 3
DA4H	3492	MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 Perfmon Local Box Wide Control
DB0H	3504	MSR_C5_PMON_EVNTSELO	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 0

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DB1H	3505	MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 1
DB2H	3506	MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 2
DB3H	3507	MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 3
DB4H	3508	MSR_C5_PMON_BOX_FILTER	Package	Uncore C-box 5 Perfmon Box Wide Filter
DB6H	3510	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 Perfmon Counter 0
DB7H	3511	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 Perfmon Counter 1
DB8H	3512	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 Perfmon Counter 2
DB9H	3513	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 Perfmon Counter 3
DC4H	3524	MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 Perfmon Local Box Wide Control
DD0H	3536	MSR_C6_PMON_EVNTSELO	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 0
DD1H	3537	MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 1
DD2H	3538	MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 2
DD3H	3539	MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 3
DD4H	3540	MSR_C6_PMON_BOX_FILTER	Package	Uncore C-box 6 Perfmon Box Wide Filter
DD6H	3542	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 Perfmon Counter 0
DD7H	3543	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 Perfmon Counter 1
DD8H	3544	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 Perfmon Counter 2
DD9H	3545	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 Perfmon Counter 3
DE4H	3556	MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 Perfmon Local Box Wide Control
DF0H	3568	MSR_C7_PMON_EVNTSELO	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 0
DF1H	3569	MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 1
DF2H	3570	MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 2
DF3H	3571	MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 3
DF4H	3572	MSR_C7_PMON_BOX_FILTER	Package	Uncore C-box 7 Perfmon Box Wide Filter
DF6H	3574	MSR_C7_PMON_CTR0	Package	Uncore C-box 7 Perfmon Counter 0
DF7H	3575	MSR_C7_PMON_CTR1	Package	Uncore C-box 7 Perfmon Counter 1
DF8H	3576	MSR_C7_PMON_CTR2	Package	Uncore C-box 7 Perfmon Counter 2
DF9H	3577	MSR_C7_PMON_CTR3	Package	Uncore C-box 7 Perfmon Counter 3

## 2.12 MSRS IN THE 3RD GENERATION INTEL® CORE™ PROCESSOR FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME IVY BRIDGE)

The 3rd generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v2 product family (based on Intel microarchitecture code name Ivy Bridge) support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, and Table 2-25. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3AH.

**Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors  
(based on Intel® microarchitecture code name Ivy Bridge)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates that TDP Limit for Turbo mode is not programmable.
		31:30		Reserved
		32	Package	Low Power Mode Support (LPM) (R/O) When set to 1, indicates that LPM is supported. When set to 0, indicates LPM is not supported.
		34:33	Package	Number of ConfigTDP Levels (R/O) 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 03: Reserved
		39:35		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		55:48	Package	Minimum Operating Ratio (R/O) Contains the minimum supported operating ratio in units of 100 MHz.
63:56		Reserved		

**Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors  
(based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.
		24:16		Reserved
		25		C3 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		C1 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		Enable C3 Undemotion (R/W) When set, enables undemotion from demoted C3.
28		Enable C1 Undemotion (R/W) When set, enables undemotion from demoted C1.		

**Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors  
(based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:29		Reserved
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	Base TDP Ratio (R/O)
		7:0		Config_TDP_Base Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1 Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1 Max Power setting allowed for ConfigTDP Level 1.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL1 MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2 Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2 Max Power setting allowed for ConfigTDP Level 2.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL2 MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	ConfigTDP Control (R/W)

**Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors  
(based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1:0		TDP_LEVEL (RW/L) System BIOS can program this field.
		30:2		Reserved.
		31		Config_TDP_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	ConfigTDP Control (R/W)
		7:0		MAX_NON_TURBO_RATIO (RW/L) System BIOS can program this field.
		30:8		Reserved
		31		TURBO_ACTIVATION_RATIO_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved

See Table 2-20, Table 2-21 and Table 2-22 for other MSR definitions applicable to processors with CPUID signature 06\_3AH.

### 2.12.1 MSRs In Intel® Xeon® Processor E5 v2 Product Family (Based on Ivy Bridge-E Microarchitecture)

Table 2-26 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3EH, see Table 2-1. These processors supports the MSR interfaces listed in Table 2-20, and Table 2-26.

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)



**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		<p>LockOut (R/WO)</p> <p>If 0, indicates that further writes to MSR_PPIN_CTL is allowed.</p> <p>If 1, indicates that further writes to MSR_PPIN_CTL is disallowed. Writing 1 to this bit is only permitted if the Enable_PPIN bit is clear.</p> <p>The Privileged System Software Inventory Agent should read MSR_PPIN_CTL[bit 1] to determine if MSR_PPIN is accessible.</p> <p>The Privileged System Software Inventory Agent is not expected to write to this MSR.</p>
		1		<p>Enable_PPIN (R/W)</p> <p>If 1, indicates that MSR_PPIN is accessible using RDMSR.</p> <p>If 0, indicates that MSR_PPIN is inaccessible using RDMSR. Any attempt to read MSR_PPIN will cause #GP.</p>
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		<p>Protected Processor Inventory Number (R/O)</p> <p>A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'.</p>
CEH	206	MSR_PLATFORM_INFO	Package	<p>Platform Information</p> <p>Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</p>
		7:0		Reserved
		15:8	Package	<p>Maximum Non-Turbo Ratio (R/O)</p> <p>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</p>
		22:16		Reserved
		23	Package	<p>PPIN_CAP (R/O)</p> <p>When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for a privileged system inventory agent to read PPIN from MSR_PPIN.</p> <p>When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.</p>
		27:24		Reserved

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	Programmable TJ OFFSET (R/O) When set to 1, indicates that MSR_TEMPERATURE_TARGET,[27:24] is valid and writable to specify a temperature offset.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.
		63:16		Reserved
179H	377	IA32_MCG_CAP	Thread	Global Machine Check Capability (R/O)
		7:0		Count
		8		MCG_CTL_P
		9		MCG_EXT_P
		10		MCP_CMCI_P
		11		MCG_TES_P
		15:12		Reserved
		23:16		MCG_EXT_CNT
		24		MCG_SER_P
		25		Reserved
		26		MCG_ELOG_P
		63:27		Reserved
17FH	383	MSR_ERROR_CONTROL	Package	MC Bank Error Configuration (R/W)
		0		Reserved
		1		MemError Log Enable (R/W) When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (RO) The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.
		27:24		TCC Activation Offset (R/W) Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only if MSR_PLATFORM_INFO.[30] is set.
		63:28		Reserved
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 9C Maximum turbo ratio limit of 9 core active.

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		15:8	Package	Maximum Ratio Limit for 10C Maximum turbo ratio limit of 10 core active.
		23:16	Package	Maximum Ratio Limit for 11C Maximum turbo ratio limit of 11 core active.
		31:24	Package	Maximum Ratio Limit for 12C Maximum turbo ratio limit of 12 core active.
		63:32		Reserved
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
296H	662	IA32_MC22_CTL2	Package	See Table 2-2.
297H	663	IA32_MC23_CTL2	Package	See Table 2-2.
298H	664	IA32_MC24_CTL2	Package	See Table 2-2.
299H	665	IA32_MC25_CTL2	Package	See Table 2-2.
29AH	666	IA32_MC26_CTL2	Package	See Table 2-2.
29BH	667	IA32_MC27_CTL2	Package	See Table 2-2.
29CH	668	IA32_MC28_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MC5 reports MC errors from the Intel QPI module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." Bank MC11 reports MC errors from a specific channel of the integrated memory controller.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
458H	1112	IA32_MC22_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
459H	1113	IA32_MC22_STATUS	Package	
45AH	1114	IA32_MC22_ADDR	Package	
45BH	1115	IA32_MC22_MISC	Package	
45CH	1116	IA32_MC23_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
45DH	1117	IA32_MC23_STATUS	Package	
45EH	1118	IA32_MC23_ADDR	Package	
45FH	1119	IA32_MC23_MISC	Package	
460H	1120	IA32_MC24_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
461H	1121	IA32_MC24_STATUS	Package	
462H	1122	IA32_MC24_ADDR	Package	
463H	1123	IA32_MC24_MISC	Package	
464H	1124	IA32_MC25_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC25 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
465H	1125	IA32_MC25_STATUS	Package	
466H	1126	IA32_MC25_ADDR	Package	
467H	1127	IA32_MC2MISC	Package	
468H	1128	IA32_MC26_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC26 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
469H	1129	IA32_MC26_STATUS	Package	
46AH	1130	IA32_MC26_ADDR	Package	
46BH	1131	IA32_MC26_MISC	Package	
46CH	1132	IA32_MC27_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC27 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
46DH	1133	IA32_MC27_STATUS	Package	
46EH	1134	IA32_MC27_ADDR	Package	
46FH	1135	IA32_MC27_MISC	Package	
470H	1136	IA32_MC28_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
471H	1137	IA32_MC28_STATUS	Package	
472H	1138	IA32_MC28_ADDR	Package	
473H	1139	IA32_MC28_MISC	Package	
613H	1555	MSR_PKG_PERF_STATUS	Package	Package RAPL Perf Status (R/O)
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."

**Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."

See Table 2-20, for other MSR definitions applicable to Intel Xeon processor E5 v2 with CPUID signature 06\_3EH.

### 2.12.2 Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family

Intel® Xeon® processor E7 v2 family (based on Ivy Bridge-E microarchitecture) with CPUID DisplayFamily\_DisplayModel signature 06\_3EH supports the MSR interfaces listed in Table 2-20, Table 2-26, and Table 2-27.

**Table 2-27. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily\_DisplayModel Signature 06\_3EH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Enable VMX Inside SMX Operation (R/WL)
		2		Enable VMX Outside SMX Operation (R/WL)
		14:8		SENTER Local Functions Enables (R/WL)
		15		SENTER Global Functions Enable (R/WL)
		63:16		Reserved
179H	377	IA32_MCG_CAP	Thread	Global Machine Check Capability (R/O)
		7:0		Count
		8		MCG_CTL_P
		9		MCG_EXT_P
		10		MCP_CMCI_P
		11		MCG_TES_P
		15:12		Reserved
		23:16		MCG_EXT_CNT
		24		MCG_SER_P
63:25		Reserved		
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status (R/WO)
		0		RIPV
		1		EIPV



**Table 2-27. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily\_DisplayModel Signature 06\_3EH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2		MCIP
		3		LMCE Signaled
		63:4		Reserved
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 9C Maximum turbo ratio limit of 9 core active.
		15:8	Package	Maximum Ratio Limit for 10C Maximum turbo ratio limit of 10core active.
		23:16	Package	Maximum Ratio Limit for 11C Maximum turbo ratio limit of 11 core active.
		31:24	Package	Maximum Ratio Limit for 12C Maximum turbo ratio limit of 12 core active.
		39:32	Package	Maximum Ratio Limit for 13C Maximum turbo ratio limit of 13 core active.
		47:40	Package	Maximum Ratio Limit for 14C Maximum turbo ratio limit of 14 core active.
		55:48	Package	Maximum Ratio Limit for 15C Maximum turbo ratio limit of 15 core active.
		62:56		Reserved
		63	Package	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
29DH	669	IA32_MC29_CTL2	Package	See Table 2-2.
29EH	670	IA32_MC30_CTL2	Package	See Table 2-2.
29FH	671	IA32_MC31_CTL2	Package	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0 (R/W)
		1		Enable PEBS on IA32_PMC1 (R/W)
		2		Enable PEBS on IA32_PMC2 (R/W)
		3		Enable PEBS on IA32_PMC3 (R/W)
		31:4		Reserved
		32		Enable Load Latency on IA32_PMC0 (R/W)
		33		Enable Load Latency on IA32_PMC1 (R/W)

**Table 2-27. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily\_DisplayModel Signature 06\_3EH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		34		Enable Load Latency on IA32_PMC2 (R/W)
		35		Enable Load Latency on IA32_PMC3 (R/W)
		63:36		Reserved
41BH	1051	IA32_MC6_MISC	Package	Misc MAC Information of Integrated I/O (R/O) See Section 15.3.2.4.
		5:0		Recoverable Address LSB
		8:6		Address Mode
		15:9		Reserved
		31:16		PCI Express Requestor ID
		39:32		PCI Express Segment Number
		63:32		Reserved
474H	1140	IA32_MC29_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
475H	1141	IA32_MC29_STATUS	Package	
476H	1142	IA32_MC29_ADDR	Package	
477H	1143	IA32_MC29_MISC	Package	
478H	1144	IA32_MC30_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
479H	1145	IA32_MC30_STATUS	Package	
47AH	1146	IA32_MC30_ADDR	Package	
47BH	1147	IA32_MC30_MISC	Package	
47CH	1148	IA32_MC31_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
47DH	1149	IA32_MC31_STATUS	Package	
47EH	1150	IA32_MC31_ADDR	Package	
47FH	1147	IA32_MC31_MISC	Package	
See Table 2-20, Table 2-26 for other MSR definitions applicable to Intel Xeon processor E7 v2 with CPUID signature 06_3AH.				

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

### 2.12.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24 and Table 2-28. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v2 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3EH.

Table 2-28. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C00H	3072	MSR_PMON_GLOBAL_CTL	Package	Uncore Perfmon Per-Socket Global Control
C01H	3073	MSR_PMON_GLOBAL_STATUS	Package	Uncore Perfmon Per-Socket Global Status
C06H	3078	MSR_PMON_GLOBAL_CONFIG	Package	Uncore Perfmon Per-Socket Global Configuration
C15H	3093	MSR_U_PMON_BOX_STATUS	Package	Uncore U-box Perfmon U-Box Wide Status
C35H	3125	MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU Perfmon Box Wide Status
D1AH	3354	MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-Box 0 Perfmon Box Wide Filter1
D3AH	3386	MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-Box 1 Perfmon Box Wide Filter1
D5AH	3418	MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-Box 2 Perfmon Box Wide Filter1
D7AH	3450	MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-Box 3 Perfmon Box Wide Filter1
D9AH	3482	MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-Box 4 Perfmon Box Wide Filter1
DBAH	3514	MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-Box 5 Perfmon Box Wide Filter1
DDAH	3546	MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-Box 6 Perfmon Box Wide Filter1
DFAH	3578	MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-Box 7 Perfmon Box Wide Filter1
E04H	3588	MSR_C8_PMON_BOX_CTL	Package	Uncore C-Box 8 Perfmon Local Box Wide Control
E10H	3600	MSR_C8_PMON_EVNTSEL0	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 0
E11H	3601	MSR_C8_PMON_EVNTSEL1	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 1
E12H	3602	MSR_C8_PMON_EVNTSEL2	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 2
E13H	3603	MSR_C8_PMON_EVNTSEL3	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 3
E14H	3604	MSR_C8_PMON_BOX_FILTER	Package	Uncore C-Box 8 Perfmon Box Wide Filter
E16H	3606	MSR_C8_PMON_CTR0	Package	Uncore C-Box 8 Perfmon Counter 0
E17H	3607	MSR_C8_PMON_CTR1	Package	Uncore C-Box 8 Perfmon Counter 1
E18H	3608	MSR_C8_PMON_CTR2	Package	Uncore C-Box 8 Perfmon Counter 2
E19H	3609	MSR_C8_PMON_CTR3	Package	Uncore C-Box 8 Perfmon Counter 3
E1AH	3610	MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-Box 8 Perfmon Box Wide Filter1
E24H	3620	MSR_C9_PMON_BOX_CTL	Package	Uncore C-Box 9 Perfmon Local Box Wide Control
E30H	3632	MSR_C9_PMON_EVNTSEL0	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 0
E31H	3633	MSR_C9_PMON_EVNTSEL1	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 1
E32H	3634	MSR_C9_PMON_EVNTSEL2	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 2
E33H	3635	MSR_C9_PMON_EVNTSEL3	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 3
E34H	3636	MSR_C9_PMON_BOX_FILTER	Package	Uncore C-Box 9 Perfmon Box Wide Filter
E36H	3638	MSR_C9_PMON_CTR0	Package	Uncore C-Box 9 Perfmon Counter 0

Table 2-28. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E37H	3639	MSR_C9_PMON_CTR1	Package	Uncore C-Box 9 Perfmon Counter 1
E38H	3640	MSR_C9_PMON_CTR2	Package	Uncore C-Box 9 Perfmon Counter 2
E39H	3641	MSR_C9_PMON_CTR3	Package	Uncore C-Box 9 Perfmon Counter 3
E3AH	3642	MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-Box 9 Perfmon Box Wide Filter1
E44H	3652	MSR_C10_PMON_BOX_CTL	Package	Uncore C-Box 10 Perfmon Local Box Wide Control
E50H	3664	MSR_C10_PMON_EVNTSELO	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 0
E51H	3665	MSR_C10_PMON_EVNTSEL1	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 1
E52H	3666	MSR_C10_PMON_EVNTSEL2	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 2
E53H	3667	MSR_C10_PMON_EVNTSEL3	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 3
E54H	3668	MSR_C10_PMON_BOX_FILTER	Package	Uncore C-Box 10 Perfmon Box Wide Filter
E56H	3670	MSR_C10_PMON_CTR0	Package	Uncore C-Box 10 Perfmon Counter 0
E57H	3671	MSR_C10_PMON_CTR1	Package	Uncore C-Box 10 Perfmon Counter 1
E58H	3672	MSR_C10_PMON_CTR2	Package	Uncore C-Box 10 Perfmon Counter 2
E59H	3673	MSR_C10_PMON_CTR3	Package	Uncore C-Box 10 Perfmon Counter 3
E5AH	3674	MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-Box 10 Perfmon Box Wide Filter1
E64H	3684	MSR_C11_PMON_BOX_CTL	Package	Uncore C-Box 11 Perfmon Local Box Wide Control
E70H	3696	MSR_C11_PMON_EVNTSELO	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 0
E71H	3697	MSR_C11_PMON_EVNTSEL1	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 1
E72H	3698	MSR_C11_PMON_EVNTSEL2	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 2
E73H	3699	MSR_C11_PMON_EVNTSEL3	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 3
E74H	3700	MSR_C11_PMON_BOX_FILTER	Package	Uncore C-Box 11 Perfmon Box Wide Filter
E76H	3702	MSR_C11_PMON_CTR0	Package	Uncore C-Box 11 Perfmon Counter 0
E77H	3703	MSR_C11_PMON_CTR1	Package	Uncore C-Box 11 Perfmon Counter 1
E78H	3704	MSR_C11_PMON_CTR2	Package	Uncore C-Box 11 Perfmon Counter 2
E79H	3705	MSR_C11_PMON_CTR3	Package	Uncore C-Box 11 Perfmon Counter 3
E7AH	3706	MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-Box 11 Perfmon Box Wide Filter1
E84H	3716	MSR_C12_PMON_BOX_CTL	Package	Uncore C-Box 12 Perfmon Local Box Wide Control
E90H	3728	MSR_C12_PMON_EVNTSELO	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 0
E91H	3729	MSR_C12_PMON_EVNTSEL1	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 1

Table 2-28. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E92H	3730	MSR_C12_PMON_EVNTSEL2	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 2
E93H	3731	MSR_C12_PMON_EVNTSEL3	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 3
E94H	3732	MSR_C12_PMON_BOX_FILTER	Package	Uncore C-Box 12 Perfmon Box Wide Filter
E96H	3734	MSR_C12_PMON_CTR0	Package	Uncore C-Box 12 Perfmon Counter 0
E97H	3735	MSR_C12_PMON_CTR1	Package	Uncore C-Box 12 Perfmon Counter 1
E98H	3736	MSR_C12_PMON_CTR2	Package	Uncore C-Box 12 Perfmon Counter 2
E99H	3737	MSR_C12_PMON_CTR3	Package	Uncore C-Box 12 Perfmon Counter 3
E9AH	3738	MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-Box 12 Perfmon Box Wide Filter1
EA4H	3748	MSR_C13_PMON_BOX_CTL	Package	Uncore C-Box 13 Perfmon Local Box Wide Control
EBOH	3760	MSR_C13_PMON_EVNTSELO	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 0
EB1H	3761	MSR_C13_PMON_EVNTSEL1	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 1
EB2H	3762	MSR_C13_PMON_EVNTSEL2	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 2
EB3H	3763	MSR_C13_PMON_EVNTSEL3	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3
EB4H	3764	MSR_C13_PMON_BOX_FILTER	Package	Uncore C-Box 13 Perfmon Box Wide Filter
EB6H	3766	MSR_C13_PMON_CTR0	Package	Uncore C-Box 13 Perfmon Counter 0
EB7H	3767	MSR_C13_PMON_CTR1	Package	Uncore C-Box 13 Perfmon Counter 1
EB8H	3768	MSR_C13_PMON_CTR2	Package	Uncore C-Box 13 Perfmon Counter 2
EB9H	3769	MSR_C13_PMON_CTR3	Package	Uncore C-Box 13 Perfmon Counter 3
EBAH	3770	MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-Box 13 Perfmon Box Wide Filter1
EC4H	3780	MSR_C14_PMON_BOX_CTL	Package	Uncore C-Box 14 Perfmon Local Box Wide Control
EDO H	3792	MSR_C14_PMON_EVNTSELO	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 0
ED1H	3793	MSR_C14_PMON_EVNTSEL1	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 1
ED2H	3794	MSR_C14_PMON_EVNTSEL2	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 2
ED3H	3795	MSR_C14_PMON_EVNTSEL3	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 3
ED4H	3796	MSR_C14_PMON_BOX_FILTER	Package	Uncore C-Box 14 Perfmon Box Wide Filter
ED6H	3798	MSR_C14_PMON_CTR0	Package	Uncore C-Box 14 Perfmon Counter 0
ED7H	3799	MSR_C14_PMON_CTR1	Package	Uncore C-Box 14 Perfmon Counter 1
ED8H	3800	MSR_C14_PMON_CTR2	Package	Uncore C-Box 14 Perfmon Counter 2
ED9H	3801	MSR_C14_PMON_CTR3	Package	Uncore C-Box 14 Perfmon Counter 3
EDA H	3802	MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-Box 14 Perfmon Box Wide Filter1

## 2.13 MSRS IN THE 4TH GENERATION INTEL® CORE™ PROCESSORS (BASED ON HASWELL MICROARCHITECTURE)

The 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v3 product family (based on Haswell microarchitecture), with CPUID DisplayFamily\_DisplayModel signature 06\_3CH/06\_45H/06\_46H, support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, and Table 2-29. For an MSR listed in Table 2-20 that also appears in Table 2-29, Table 2-29 supersedes Table 2-20.

The MSRs listed in Table 2-29 also apply to processors based on Haswell-E microarchitecture (see Section 2.14).

**Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3BH	59	IA32_TSC_ADJUST	Thread	Per-Logical-Processor TSC ADJUST (R/W) See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved
		32	Package	Low Power Mode Support (LPM) (R/O) When set to 1, indicates that LPM is supported. When set to 0, indicates LPM is not supported.
		34:33	Package	Number of ConfigTDP Levels (R/O) 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 03: Reserved
		39:35		Reserved
	47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	

**Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		55:48	Package	Minimum Operating Ratio (R/O) Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved
186H	390	IA32_PERFEVTSELO	Thread	Performance Event Select for Counter 0 (R/W) Supports all fields described in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.
187H	391	IA32_PERFEVTSEL1	Thread	Performance Event Select for Counter 1 (R/W) Supports all fields described in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.
188H	392	IA32_PERFEVTSEL2	Thread	Performance Event Select for Counter 2 (R/W) Supports all fields described in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.
		33		IN_TXCP: See Section 18.3.6.5.1. When IN_TXCP=1 & IN_TX=1 and in sampling, a spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large "sample-after" value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.
189H	393	IA32_PERFEVTSEL3	Thread	Performance Event Select for Counter 3 (R/W) Supports all fields described in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.
1C8H	456	MSR_LBR_SELECT	Thread	Last Branch Record Filtering Select Register (R/W)
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL

**Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
		9		EN_CALL_STACK
		63:9		Reserved
1D9H	473	IA32_DEBUGCTL	Thread	Debug Control (R/W) See Table 2-2.
		0		LBR: Last Branch Record
		1		BTF
		5:2		Reserved
		6		TR: Branch Trace
		7		BTS: Log Branch Trace Message to BTS Buffer
		8		BTINT
		9		BTS_OFF_OS
		10		BTS_OFF_USER
		11		FREEZE_LBR_ON_PMI
		12		FREEZE_PERFMON_ON_PMI
		13		ENABLE_UNCORE_PMI
		14		FREEZE_WHILE_SMM
		15		RTM_DEBUG
		63:15		Reserved
491H	1169	IA32_VMX_VMFUNC	Thread	Capability Reporting Register of VM-Function Controls (R/O) See Table 2-2.
60BH	1548	MSR_PKGIC_IRTL1	Package	Package C6/C7 Interrupt Response Limit 1 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to a C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.



**Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved
60CH	1548	MSR_PKGC_IRTL2	Package	Package C6/C7 Interrupt Response Limit 2 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to a C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		Interrupt response time limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.
		14:13		Reserved
		15		Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved
613H	1555	MSR_PKG_PERF_STATUS	Package	PKG Perf Status (R/O) See Section 14.10.3, "Package RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	Base TDP Ratio (R/O)
		7:0		Config_TDP_Base Base TDP level ratio to be used for this specific processor (in units of 100 MHz).

**Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:8		Reserved
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 Ratio and Power Level (R/O)
		14:0		PKG_TDP_LVL1 Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1 Max Power setting allowed for ConfigTDP Level 1.
		62:47		PKG_MIN_PWR_LVL1 MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 Ratio and Power Level (R/O)
		14:0		PKG_TDP_LVL2 Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2 Max Power setting allowed for ConfigTDP Level 2.
		62:47		PKG_MIN_PWR_LVL2 MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	ConfigTDP Control (R/W)
		1:0		TDP_LEVEL (RW/L) System BIOS can program this field.
		30:2		Reserved
		31		Config_TDP_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	ConfigTDP Control (R/W)

**Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0		MAX_NON_TURBO_RATIO (R/W/L) System BIOS can program this field.
		30:8		Reserved
		31		TURBO_ACTIVATION_RATIO_Lock (R/W/L) When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved
C80H	3200	IA32_DEBUG_INTERFACE	Package	Silicon Debug Feature Control (R/W) See Table 2-2.

### 2.13.1 MSRs in 4th Generation Intel® Core™ Processor Family (based on Haswell Microarchitecture)

Table 2-30 lists model-specific registers (MSRs) that are specific to 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200 v3 product family (based on Haswell microarchitecture). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3CH/06\_45H/06\_46H, see Table 2-1.

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		3:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s Package C states C7 are not available to processors with a signature of 06_3CH.
		9:4		Reserved
		10		I/O MWAIT Redirection Enable (R/W)

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:11		Reserved
		15		CFG Lock (R/WO)
		24:16		Reserved
		25		C3 State Auto Demotion Enable (R/W)
		26		C1 State Auto Demotion Enable (R/W)
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		63:29		Reserved
17DH	381	MSR_SMM_MCA_CAP	THREAD	Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		Reserved
		58		SMM_Code_Access_Chk (SMM-RO) If set to 1, indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		63:32		Reserved
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU Global Control
		0		Core 0 select.
		1		Core 1 select.
		2		Core 2 select.
		3		Core 3 select.
		18:4		Reserved

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		29		Enable all uncore counters.
		30		Enable wake on PMI.
		31		Enable Freezing counter when overflow.
		63:32		Reserved
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU Main Status
		0		Fixed counter overflowed.
		1		An ARB counter overflowed.
		2		Reserved
		3		A CBox counter overflowed (on any slice).
		63:4		Reserved
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore Fixed Counter Control (R/W)
		19:0		Reserved
		20		Enable overflow propagation.
		21		Reserved
		22		Enable counting.
		63:23		Reserved
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore Fixed Counter
		47:0		Current count.
		63:48		Reserved
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box Configuration Information (R/O)
		3:0		Encoded number of C-Box, derive value by "-1".
		63:4		Reserved
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb Unit, Performance Counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb Unit, Performance Counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb Unit, Counter 0 Event Select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb Unit, Counter 1 Event Select MSR
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU Global Control
		0		Core 0 select.
		1		Core 1 select.
		2		Core 2 select.
		3		Core 3 select.
		18:4		Reserved
		29		Enable all uncore counters.
		30		Enable wake on PMI.
		31		Enable Freezing counter when overflow.
63:32		Reserved		

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore Fixed Counter
		47:0		Current count.
		63:48		Reserved
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb Unit, Counter 1 Event Select MSR
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	Enhanced SMM Feature Control (SMM-RW) Reports SMM capability Enhancement. Accessible only while in SMM.
		0		Lock (SMM-RWO) When set to '1' locks this register from further changes.
		1		Reserved
		2		SMM_Code_Chk_En (SMM-RW) This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	SMM Delayed (SMM-RO) Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.
		N-1:0		LOG_PROC_STATE (SMM-RO) Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	SMM Blocked (SMM-RO) Reports the blocked state of all logical processors in the package. Available only while in SMM.

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		N-1:0		LOG_PROC_STATE (SMM-RO) Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers Used in RAPL Interfaces (R/O)
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
		15:13	Package	Reserved
		19:16	Package	Time Units See Section 14.10.1, "RAPL Interfaces."
		63:20		Reserved
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	PP1 RAPL Power Limit Control (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	PP1 Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	PP1 Balance Policy (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)
		0		PROCHOT Status (RO) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (RO) When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		4		Graphics Driver Status (RO) When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		Autonomous Utilization-Based Frequency Control Status (RO) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		VR Therm Alert Status (RO) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved
		8		Electrical Design Point Status (RO) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).
		9		Core Power Limiting Status (RO) When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		Package-Level Power Limiting PL1 Status (RO) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		Package-Level PL2 Power Limiting Status (RO) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		12		Max Turbo Limit Status (RO) When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		Turbo Transition Attenuation Status (RO) When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.



**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17		<p>Thermal Log</p> <p>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		19:18		Reserved
		20		<p>Graphics Driver Log</p> <p>When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		21		<p>Autonomous Utilization-Based Frequency Control Log</p> <p>When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		22		<p>VR Therm Alert Log</p> <p>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		23		Reserved
		24		<p>Electrical Design Point Log</p> <p>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		25		<p>Core Power Limiting Log</p> <p>When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		26		<p>Package-Level PL1 Power Limiting Log</p> <p>When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		27		Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in the Processor Graphics (R/W) (Frequency refers to processor graphics frequency.)
		0		PROCHOT Status (R0) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved
		4		Graphics Driver Status (R0) When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		8		Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).
		9		Graphics Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved
		20		Graphics Driver Log When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		Autonomous Utilization-Based Frequency Control Log When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		23		Reserved
		24		Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in the Ring Interconnect (R/W) (Frequency refers to ring interconnect in the uncore.)
		0		PROCHOT Status (R0) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.
		5:2		Reserved
		6		VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved
		8		Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).
		9		Reserved
		10		Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		Graphics Driver Log When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		21		Autonomous Utilization-Based Frequency Control Log When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved
		24		Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		29		Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved
700H	1792	MSR_UNC_CBO_0_PERFEVTSEL0	Package	Uncore C-Box 0, Counter 0 Event Select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, Counter 1 Event Select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTRO	Package	Uncore C-Box 0, Performance Counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, Performance Counter 1
710H	1808	MSR_UNC_CBO_1_PERFEVTSEL0	Package	Uncore C-Box 1, Counter 0 Event Select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, Counter 1 Event Select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTRO	Package	Uncore C-Box 1, Performance Counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, Performance Counter 1
720H	1824	MSR_UNC_CBO_2_PERFEVTSEL0	Package	Uncore C-Box 2, Counter 0 Event Select MSR
721H	1824	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, Counter 1 Event Select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTRO	Package	Uncore C-Box 2, Performance Counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, Performance Counter 1
730H	1840	MSR_UNC_CBO_3_PERFEVTSEL0	Package	Uncore C-Box 3, Counter 0 Event Select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, Counter 1 Event Select MSR
736H	1846	MSR_UNC_CBO_3_PERFCTRO	Package	Uncore C-Box 3, Performance Counter 0
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, Performance Counter 1
See Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29 for other MSR definitions applicable to processors with CPUID signatures 063CH, 06_46H.				

### 2.13.2 Additional Residency MSRs Supported in 4th Generation Intel® Core™ Processors

The 4th generation Intel® Core™ processor family (based on Haswell microarchitecture) with CPUID DisplayFamily\_DisplayModel signature 06\_45H supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-30, and Table 2-31.

**Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily\_DisplayModel Signature 06\_45H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		3:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.  The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		I/O MWAIT Redirection Enable (R/W)
		14:11		Reserved
		15		CFG Lock (R/WO)
		24:16		Reserved
		25		C3 State Auto Demotion Enable (R/W)
		26		C1 State Auto Demotion Enable (R/W)
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		63:29		Reserved
630H	1584	MSR_PKG_C8_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C8 Residency Counter (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.
		63:60		Reserved



**Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily\_DisplayModel Signature 06\_45H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
631H	1585	MSR_PKG_C9_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C9 Residency Counter (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.
		63:60		Reserved
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C10 Residency Counter (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.
		63:60		Reserved

See Table 2-20, Table 2-21, Table 2-22, Table 2-29, Table 2-30 for other MSR definitions applicable to processors with CPUID signature 06\_45H.

## 2.14 MSRS IN INTEL® XEON® PROCESSOR E5 V3 AND E7 V3 PRODUCT FAMILY

Intel® Xeon® processor E5 v3 family and Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID DisplayFamily\_DisplayModel = 06\_3F). These processors supports the MSR interfaces listed in Table 2-20, Table 2-29, and Table 2-32.

**Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
35H	53	MSR_CORE_THREAD_COUNT	Package	Configured State of Enabled Processor Core Count and Logical Processor Count (RO) <ul style="list-style-type: none"> <li>After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package.</li> <li>Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.</li> </ul>
		15:0		THREAD_COUNT (RO) The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		31:16		Core_COUNT (RO) The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		63:32		Reserved
53H	83	MSR_THREAD_ID_INFO	Thread	A Hardware Assigned ID for the Logical Processor (RO)
		7:0		Logical_Processor_ID (RO) An implementation-specific numerical value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.
		63:8		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W)
		14:11		Reserved
		15		CFG Lock (R/W0)
		24:16		Reserved
		25		C3 State Auto Demotion Enable (R/W)
		26		C1 State Auto Demotion Enable (R/W)
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		29		Package C State Demotion Enable (R/W)
		30		Package C State UnDemotion Enable (R/W)
63:31		Reserved		

**Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
179H	377	IA32_MCG_CAP	Thread	Global Machine Check Capability (R/O)
		7:0		Count
		8		MCG_CTL_P
		9		MCG_EXT_P
		10		MCP_CMCI_P
		11		MCG_TES_P
		15:12		Reserved
		23:16		MCG_EXT_CNT
		24		MCG_SER_P
		25		MCG_EM_P
		26		MCG_ELOG_P
	63:27		Reserved	
17DH	381	MSR_SMM_MCA_CAP	Thread	Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		Reserved
		58		SMM_Code_Access_Chk (SMM-RO) If set to 1, indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
17FH	383	MSR_ERROR_CONTROL	Package	MC Bank Error Configuration (R/W)
		0		Reserved
		1		MemError Log Enable (R/W) When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		39:32	Package	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 core active.
		47:40	Package	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 core active.
		55:48	Package	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 core active.
		63:56	Package	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 core active.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 9C Maximum turbo ratio limit of 9 core active.
		15:8	Package	Maximum Ratio Limit for 10C Maximum turbo ratio limit of 10 core active.
		23:16	Package	Maximum Ratio Limit for 11C Maximum turbo ratio limit of 11 core active.
		31:24	Package	Maximum Ratio Limit for 12C Maximum turbo ratio limit of 12 core active.
		39:32	Package	Maximum Ratio Limit for 13C Maximum turbo ratio limit of 13 core active.
		47:40	Package	Maximum Ratio Limit for 14C Maximum turbo ratio limit of 14 core active.
		55:48	Package	Maximum Ratio Limit for 15C Maximum turbo ratio limit of 15 core active.
		63:56	Package	Maximum Ratio Limit for 16C Maximum turbo ratio limit of 16 core active.
1AFH	431	MSR_TURBO_RATIO_LIMIT2	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 17C Maximum turbo ratio limit of 17 core active.
		15:8	Package	Maximum Ratio Limit for 18C Maximum turbo ratio limit of 18 core active.
		62:16	Package	Reserved

**Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63	Package	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC errors from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC20 reports MC errors from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	

**Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers Used in RAPL Interfaces (R/O)
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{ESU}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
		15:13	Package	Reserved
		19:16	Package	Time Units See Section 14.10.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) Energy Consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
61EH	1566	MSR_PCIE_PLL_RATIO	Package	Configuration of PCIe PLL Relative to BCLK(R/W)
		1:0	Package	PCIe Ratio (R/W) 00b: Use 5:5 mapping for 100MHz operation (default). 01b: Use 5:4 mapping for 125MHz operation. 10b: Use 5:3 mapping for 166MHz operation. 11b: Use 5:2 mapping for 250MHz operation.
		2	Package	LPLL Select (R/W) If 1, use configured setting of PCIe Ratio.
		3	Package	LONG RESET (R/W) If 1, wait an additional time-out before re-locking Gen2/Gen3 PLLs.
		63:4		Reserved

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	Reserved (R/O) Reads return 0.
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)
		0		PROCHOT Status (R0) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.
		2		Power Budget Management Status (R0) When set, frequency is reduced below the operating system request due to PBM limit
		3		Platform Configuration Services Status (R0) When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved
		5		Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved



**Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		8		Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).
		9		Reserved
		10		Multi-Core Turbo Status (R0) When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits.
		12:11		Reserved
		13		Core Frequency P1 Status (R0) When set, frequency is reduced below max non-turbo P1.
		14		Core Max N-Core Turbo Frequency Limiting Status (R0) When set, frequency is reduced below max n-core turbo frequency.
		15		Core Frequency Limiting Status (R0) When set, frequency is reduced below the operating system request.
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		Power Budget Management Log When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		Platform Configuration Services Log When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		21		Autonomous Utilization-Based Frequency Control Log When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved
		24		Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved
		26		Multi-Core Turbo Log When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved
		29		Core Frequency P1 Log When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		Core Max N-Core Turbo Frequency Limiting Log When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		Core Frequency Limiting Log When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved
C8DH	3213	IA32_QM_EVTSEL	THREAD	Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.

**Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0		EventID (Rw) Event encoding: 0x0: No monitoring. 0x1: L3 occupancy monitoring. All other encoding reserved.
		31:8		Reserved
		41:32		RMID (Rw)
		63:42		Reserved
C8EH	3214	IA32_QM_CTR	THREAD	Monitoring Counter Register (R/O) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.
		61:0		Resource Monitored Data
		62		Unavailable: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.
		63		Error: If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.
C8FH	3215	IA32_PQR_ASSOC	THREAD	Resource Association Register (R/W)
		9:0		RMID
		63: 10		Reserved

See Table 2-20, Table 2-29 for other MSR definitions applicable to processors with CPUID signature 06\_3FH.

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

**2.14.1 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family**

Intel Xeon Processor E5 v3 and E7 v3 family are based on the Haswell-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-33. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v3 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3FH.

**Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
700H	1792	MSR_PMON_GLOBAL_CTL	Package	Uncore Perfmon Per-Socket Global Control
701H	1793	MSR_PMON_GLOBAL_STATUS	Package	Uncore Perfmon Per-Socket Global Status
702H	1794	MSR_PMON_GLOBAL_CONFIG	Package	Uncore Perfmon Per-Socket Global Configuration
703H	1795	MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-Box UCLK Fixed Counter Control
704H	1796	MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-Box UCLK Fixed Counter
705H	1797	MSR_U_PMON_EVNTSELO	Package	Uncore U-Box Perfmon Event Select for U-Box Counter 0

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
706H	1798	MSR_U_PMON_EVNTSEL1	Package	Uncore U-Box Perfmon Event Select for U-Box Counter 1
708H	1800	MSR_U_PMON_BOX_STATUS	Package	Uncore U-Box Perfmon U-Box Wide Status
709H	1801	MSR_U_PMON_CTR0	Package	Uncore U-Box Perfmon Counter 0
70AH	1802	MSR_U_PMON_CTR1	Package	Uncore U-Box Perfmon Counter 1
710H	1808	MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU Perfmon for PCU-Box-Wide Control
711H	1809	MSR_PCU_PMON_EVNTSELO	Package	Uncore PCU Perfmon Event Select for PCU Counter 0
712H	1810	MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU Perfmon Event Select for PCU Counter 1
713H	1811	MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU Perfmon Event Select for PCU Counter 2
714H	1812	MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU Perfmon Event Select for PCU Counter 3
715H	1813	MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU Perfmon Box-Wide Filter
716H	1814	MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU Perfmon Box Wide Status
717H	1815	MSR_PCU_PMON_CTR0	Package	Uncore PCU Perfmon Counter 0
718H	1816	MSR_PCU_PMON_CTR1	Package	Uncore PCU Perfmon Counter 1
719H	1817	MSR_PCU_PMON_CTR2	Package	Uncore PCU Perfmon Counter 2
71AH	1818	MSR_PCU_PMON_CTR3	Package	Uncore PCU Perfmon Counter 3
720H	1824	MSR_S0_PMON_BOX_CTL	Package	Uncore SBo 0 Perfmon for SBo 0 Box-Wide Control
721H	1825	MSR_S0_PMON_EVNTSELO	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 0
722H	1826	MSR_S0_PMON_EVNTSEL1	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 1
723H	1827	MSR_S0_PMON_EVNTSEL2	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 2
724H	1828	MSR_S0_PMON_EVNTSEL3	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 3
725H	1829	MSR_S0_PMON_BOX_FILTER	Package	Uncore SBo 0 Perfmon Box-Wide Filter
726H	1830	MSR_S0_PMON_CTR0	Package	Uncore SBo 0 Perfmon Counter 0
727H	1831	MSR_S0_PMON_CTR1	Package	Uncore SBo 0 Perfmon Counter 1
728H	1832	MSR_S0_PMON_CTR2	Package	Uncore SBo 0 Perfmon Counter 2
729H	1833	MSR_S0_PMON_CTR3	Package	Uncore SBo 0 Perfmon Counter 3
72AH	1834	MSR_S1_PMON_BOX_CTL	Package	Uncore SBo 1 Perfmon for SBo 1 Box-Wide Control
72BH	1835	MSR_S1_PMON_EVNTSELO	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 0
72CH	1836	MSR_S1_PMON_EVNTSEL1	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 1
72DH	1837	MSR_S1_PMON_EVNTSEL2	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 2
72EH	1838	MSR_S1_PMON_EVNTSEL3	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 3
72FH	1839	MSR_S1_PMON_BOX_FILTER	Package	Uncore SBo 1 Perfmon Box-Wide Filter

**Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
730H	1840	MSR_S1_PMON_CTR0	Package	Uncore SBo 1 Perfmon Counter 0
731H	1841	MSR_S1_PMON_CTR1	Package	Uncore SBo 1 Perfmon Counter 1
732H	1842	MSR_S1_PMON_CTR2	Package	Uncore SBo 1 Perfmon Counter 2
733H	1843	MSR_S1_PMON_CTR3	Package	Uncore SBo 1 Perfmon Counter 3
734H	1844	MSR_S2_PMON_BOX_CTL	Package	Uncore SBo 2 Perfmon for SBo 2 Box-Wide Control
735H	1845	MSR_S2_PMON_EVNTSELO	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 0
736H	1846	MSR_S2_PMON_EVNTSEL1	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 1
737H	1847	MSR_S2_PMON_EVNTSEL2	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 2
738H	1848	MSR_S2_PMON_EVNTSEL3	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 3
739H	1849	MSR_S2_PMON_BOX_FILTER	Package	Uncore SBo 2 Perfmon Box-Wide Filter
73AH	1850	MSR_S2_PMON_CTR0	Package	Uncore SBo 2 Perfmon Counter 0
73BH	1851	MSR_S2_PMON_CTR1	Package	Uncore SBo 2 Perfmon Counter 1
73CH	1852	MSR_S2_PMON_CTR2	Package	Uncore SBo 2 Perfmon Counter 2
73DH	1853	MSR_S2_PMON_CTR3	Package	Uncore SBo 2 Perfmon Counter 3
73EH	1854	MSR_S3_PMON_BOX_CTL	Package	Uncore SBo 3 Perfmon for SBo 3 Box-Wide Control
73FH	1855	MSR_S3_PMON_EVNTSELO	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 0
740H	1856	MSR_S3_PMON_EVNTSEL1	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 1
741H	1857	MSR_S3_PMON_EVNTSEL2	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 2
742H	1858	MSR_S3_PMON_EVNTSEL3	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 3
743H	1859	MSR_S3_PMON_BOX_FILTER	Package	Uncore SBo 3 Perfmon Box-Wide Filter
744H	1860	MSR_S3_PMON_CTR0	Package	Uncore SBo 3 Perfmon Counter 0
745H	1861	MSR_S3_PMON_CTR1	Package	Uncore SBo 3 Perfmon Counter 1
746H	1862	MSR_S3_PMON_CTR2	Package	Uncore SBo 3 Perfmon Counter 2
747H	1863	MSR_S3_PMON_CTR3	Package	Uncore SBo 3 Perfmon Counter 3
E00H	3584	MSR_CO_PMON_BOX_CTL	Package	Uncore C-Box 0 Perfmon for Box-Wide Control
E01H	3585	MSR_CO_PMON_EVNTSELO	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 0
E02H	3586	MSR_CO_PMON_EVNTSEL1	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 1
E03H	3587	MSR_CO_PMON_EVNTSEL2	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 2
E04H	3588	MSR_CO_PMON_EVNTSEL3	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E05H	3589	MSR_CO_PMON_BOX_FILTER0	Package	Uncore C-Box 0 Perfmon Box Wide Filter 0
E06H	3590	MSR_CO_PMON_BOX_FILTER1	Package	Uncore C-Box 0 Perfmon Box Wide Filter 1
E07H	3591	MSR_CO_PMON_BOX_STATUS	Package	Uncore C-Box 0 Perfmon Box Wide Status
E08H	3592	MSR_CO_PMON_CTR0	Package	Uncore C-Box 0 Perfmon Counter 0
E09H	3593	MSR_CO_PMON_CTR1	Package	Uncore C-Box 0 Perfmon Counter 1
E0AH	3594	MSR_CO_PMON_CTR2	Package	Uncore C-Box 0 Perfmon Counter 2
E0BH	3595	MSR_CO_PMON_CTR3	Package	Uncore C-Box 0 Perfmon Counter 3
E10H	3600	MSR_C1_PMON_BOX_CTL	Package	Uncore C-Box 1 Perfmon for Box-Wide Control
E11H	3601	MSR_C1_PMON_EVNTSELO	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 0
E12H	3602	MSR_C1_PMON_EVNTSEL1	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 1
E13H	3603	MSR_C1_PMON_EVNTSEL2	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 2
E14H	3604	MSR_C1_PMON_EVNTSEL3	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 3
E15H	3605	MSR_C1_PMON_BOX_FILTER0	Package	Uncore C-Box 1 Perfmon Box Wide Filter 0
E16H	3606	MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-Box 1 Perfmon Box Wide Filter1
E17H	3607	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-Box 1 Perfmon Box Wide Status
E18H	3608	MSR_C1_PMON_CTR0	Package	Uncore C-Box 1 Perfmon Counter 0
E19H	3609	MSR_C1_PMON_CTR1	Package	Uncore C-Box 1 Perfmon Counter 1
E1AH	3610	MSR_C1_PMON_CTR2	Package	Uncore C-Box 1 Perfmon Counter 2
E1BH	3611	MSR_C1_PMON_CTR3	Package	Uncore C-Box 1 Perfmon Counter 3
E20H	3616	MSR_C2_PMON_BOX_CTL	Package	Uncore C-Box 2 Perfmon for Box-Wide Control
E21H	3617	MSR_C2_PMON_EVNTSELO	Package	Uncore C-Box 2 Perfmon Event Select for C-Box 2 Counter 0
E22H	3618	MSR_C2_PMON_EVNTSEL1	Package	Uncore C-Box 2 Perfmon Event Select for C-Box 2 Counter 1
E23H	3619	MSR_C2_PMON_EVNTSEL2	Package	Uncore C-Box 2 Perfmon Event Select for C-Box 2 Counter 2
E24H	3620	MSR_C2_PMON_EVNTSEL3	Package	Uncore C-Box 2 Perfmon Event select for C-Box 2 Counter 3
E25H	3621	MSR_C2_PMON_BOX_FILTER0	Package	Uncore C-Box 2 Perfmon Box Wide Filter 0
E26H	3622	MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-Box 2 Perfmon Box Wide Filter1
E27H	3623	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-Box 2 Perfmon Box Wide Status
E28H	3624	MSR_C2_PMON_CTR0	Package	Uncore C-Box 2 Perfmon Counter 0
E29H	3625	MSR_C2_PMON_CTR1	Package	Uncore C-Box 2 Perfmon Counter 1
E2AH	3626	MSR_C2_PMON_CTR2	Package	Uncore C-Box 2 Perfmon Counter 2
E2BH	3627	MSR_C2_PMON_CTR3	Package	Uncore C-Box 2 Perfmon Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E30H	3632	MSR_C3_PMON_BOX_CTL	Package	Uncore C-Box 3 Perfmon for Box-Wide Control
E31H	3633	MSR_C3_PMON_EVNTSELO	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 0
E32H	3634	MSR_C3_PMON_EVNTSEL1	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 1
E33H	3635	MSR_C3_PMON_EVNTSEL2	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 2
E34H	3636	MSR_C3_PMON_EVNTSEL3	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 3
E35H	3637	MSR_C3_PMON_BOX_FILTER0	Package	Uncore C-Box 3 Perfmon Box Wide Filter 0
E36H	3638	MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-Box 3 Perfmon Box Wide Filter1
E37H	3639	MSR_C3_PMON_BOX_STATUS	Package	Uncore C-Box 3 Perfmon Box Wide Status
E38H	3640	MSR_C3_PMON_CTRL0	Package	Uncore C-Box 3 Perfmon Counter 0
E39H	3641	MSR_C3_PMON_CTRL1	Package	Uncore C-Box 3 Perfmon Counter 1
E3AH	3642	MSR_C3_PMON_CTRL2	Package	Uncore C-Box 3 Perfmon Counter 2
E3BH	3643	MSR_C3_PMON_CTRL3	Package	Uncore C-Box 3 Perfmon Counter 3
E40H	3648	MSR_C4_PMON_BOX_CTL	Package	Uncore C-Box 4 Perfmon for Box-Wide Control
E41H	3649	MSR_C4_PMON_EVNTSELO	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 0
E42H	3650	MSR_C4_PMON_EVNTSEL1	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 1
E43H	3651	MSR_C4_PMON_EVNTSEL2	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 2
E44H	3652	MSR_C4_PMON_EVNTSEL3	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 3
E45H	3653	MSR_C4_PMON_BOX_FILTER0	Package	Uncore C-Box 4 Perfmon Box Wide Filter 0
E46H	3654	MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-Box 4 Perfmon Box Wide Filter1
E47H	3655	MSR_C4_PMON_BOX_STATUS	Package	Uncore C-Box 4 Perfmon Box Wide Status
E48H	3656	MSR_C4_PMON_CTRL0	Package	Uncore C-Box 4 Perfmon Counter 0
E49H	3657	MSR_C4_PMON_CTRL1	Package	Uncore C-Box 4 Perfmon Counter 1
E4AH	3658	MSR_C4_PMON_CTRL2	Package	Uncore C-Box 4 Perfmon Counter 2
E4BH	3659	MSR_C4_PMON_CTRL3	Package	Uncore C-Box 4 Perfmon Counter 3
E50H	3664	MSR_C5_PMON_BOX_CTL	Package	Uncore C-Box 5 Perfmon for Box-Wide Control
E51H	3665	MSR_C5_PMON_EVNTSELO	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 0
E52H	3666	MSR_C5_PMON_EVNTSEL1	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 1
E53H	3667	MSR_C5_PMON_EVNTSEL2	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 2
E54H	3668	MSR_C5_PMON_EVNTSEL3	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E55H	3669	MSR_C5_PMON_BOX_FILTER0	Package	Uncore C-Box 5 Perfmon Box Wide Filter 0
E56H	3670	MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-Box 5 Perfmon Box Wide Filter 1
E57H	3671	MSR_C5_PMON_BOX_STATUS	Package	Uncore C-Box 5 Perfmon Box Wide Status
E58H	3672	MSR_C5_PMON_CTR0	Package	Uncore C-Box 5 Perfmon Counter 0
E59H	3673	MSR_C5_PMON_CTR1	Package	Uncore C-Box 5 Perfmon Counter 1
E5AH	3674	MSR_C5_PMON_CTR2	Package	Uncore C-Box 5 Perfmon Counter 2
E5BH	3675	MSR_C5_PMON_CTR3	Package	Uncore C-Box 5 Perfmon Counter 3
E60H	3680	MSR_C6_PMON_BOX_CTL	Package	Uncore C-Box 6 Perfmon for Box-Wide Control
E61H	3681	MSR_C6_PMON_EVNTSELO	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 0
E62H	3682	MSR_C6_PMON_EVNTSEL1	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 1
E63H	3683	MSR_C6_PMON_EVNTSEL2	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 2
E64H	3684	MSR_C6_PMON_EVNTSEL3	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 3
E65H	3685	MSR_C6_PMON_BOX_FILTER0	Package	Uncore C-Box 6 Perfmon Box Wide Filter 0
E66H	3686	MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-Box 6 Perfmon Box Wide Filter 1
E67H	3687	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-Box 6 Perfmon Box Wide Status
E68H	3688	MSR_C6_PMON_CTR0	Package	Uncore C-Box 6 Perfmon Counter 0
E69H	3689	MSR_C6_PMON_CTR1	Package	Uncore C-Box 6 Perfmon Counter 1
E6AH	3690	MSR_C6_PMON_CTR2	Package	Uncore C-Box 6 Perfmon Counter 2
E6BH	3691	MSR_C6_PMON_CTR3	Package	Uncore C-Box 6 Perfmon Counter 3
E70H	3696	MSR_C7_PMON_BOX_CTL	Package	Uncore C-Box 7 Perfmon for Box-Wide Control
E71H	3697	MSR_C7_PMON_EVNTSELO	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 0
E72H	3698	MSR_C7_PMON_EVNTSEL1	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 1
E73H	3699	MSR_C7_PMON_EVNTSEL2	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 2
E74H	3700	MSR_C7_PMON_EVNTSEL3	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 3
E75H	3701	MSR_C7_PMON_BOX_FILTER0	Package	Uncore C-Box 7 Perfmon Box Wide Filter 0
E76H	3702	MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-Box 7 Perfmon Box Wide Filter 1
E77H	3703	MSR_C7_PMON_BOX_STATUS	Package	Uncore C-Box 7 Perfmon Box Wide Status
E78H	3704	MSR_C7_PMON_CTR0	Package	Uncore C-Box 7 Perfmon Counter 0
E79H	3705	MSR_C7_PMON_CTR1	Package	Uncore C-Box 7 Perfmon Counter 1
E7AH	3706	MSR_C7_PMON_CTR2	Package	Uncore C-Box 7 Perfmon Counter 2
E7BH	3707	MSR_C7_PMON_CTR3	Package	Uncore C-Box 7 Perfmon Counter 3



**Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E80H	3712	MSR_C8_PMON_BOX_CTL	Package	Uncore C-Box 8 Perfmon Local Box Wide Control
E81H	3713	MSR_C8_PMON_EVNTSELO	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 0
E82H	3714	MSR_C8_PMON_EVNTSEL1	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 1
E83H	3715	MSR_C8_PMON_EVNTSEL2	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 2
E84H	3716	MSR_C8_PMON_EVNTSEL3	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 3
E85H	3717	MSR_C8_PMON_BOX_FILTER0	Package	Uncore C-Box 8 Perfmon Box Wide Filter 0
E86H	3718	MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-Box 8 Perfmon Box Wide Filter 1
E87H	3719	MSR_C8_PMON_BOX_STATUS	Package	Uncore C-Box 8 Perfmon Box Wide Status
E88H	3720	MSR_C8_PMON_CTRL0	Package	Uncore C-Box 8 Perfmon Counter 0
E89H	3721	MSR_C8_PMON_CTRL1	Package	Uncore C-Box 8 Perfmon Counter 1
E8AH	3722	MSR_C8_PMON_CTRL2	Package	Uncore C-Box 8 Perfmon Counter 2
E8BH	3723	MSR_C8_PMON_CTRL3	Package	Uncore C-Box 8 Perfmon Counter 3
E90H	3728	MSR_C9_PMON_BOX_CTL	Package	Uncore C-Box 9 Perfmon Local Box Wide Control
E91H	3729	MSR_C9_PMON_EVNTSELO	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 0
E92H	3730	MSR_C9_PMON_EVNTSEL1	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 1
E93H	3731	MSR_C9_PMON_EVNTSEL2	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 2
E94H	3732	MSR_C9_PMON_EVNTSEL3	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 3
E95H	3733	MSR_C9_PMON_BOX_FILTER0	Package	Uncore C-Box 9 Perfmon Box Wide Filter 0
E96H	3734	MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-Box 9 Perfmon Box Wide Filter 1
E97H	3735	MSR_C9_PMON_BOX_STATUS	Package	Uncore C-Box 9 Perfmon Box Wide Status
E98H	3736	MSR_C9_PMON_CTRL0	Package	Uncore C-Box 9 Perfmon Counter 0
E99H	3737	MSR_C9_PMON_CTRL1	Package	Uncore C-Box 9 Perfmon Counter 1
E9AH	3738	MSR_C9_PMON_CTRL2	Package	Uncore C-Box 9 Perfmon Counter 2
E9BH	3739	MSR_C9_PMON_CTRL3	Package	Uncore C-Box 9 Perfmon Counter 3
EA0H	3744	MSR_C10_PMON_BOX_CTL	Package	Uncore C-Box 10 Perfmon Local Box Wide Control
EA1H	3745	MSR_C10_PMON_EVNTSELO	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 0
EA2H	3746	MSR_C10_PMON_EVNTSEL1	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 1
EA3H	3747	MSR_C10_PMON_EVNTSEL2	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 2
EA4H	3748	MSR_C10_PMON_EVNTSEL3	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
EA5H	3749	MSR_C10_PMON_BOX_FILTER0	Package	Uncore C-Box 10 Perfmon Box Wide Filter 0
EA6H	3750	MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-Box 10 Perfmon Box Wide Filter 1
EA7H	3751	MSR_C10_PMON_BOX_STATUS	Package	Uncore C-Box 10 Perfmon Box Wide Status
EA8H	3752	MSR_C10_PMON_CTR0	Package	Uncore C-Box 10 Perfmon Counter 0
EA9H	3753	MSR_C10_PMON_CTR1	Package	Uncore C-Box 10 Perfmon Counter 1
EA AH	3754	MSR_C10_PMON_CTR2	Package	Uncore C-Box 10 Perfmon Counter 2
EABH	3755	MSR_C10_PMON_CTR3	Package	Uncore C-Box 10 Perfmon Counter 3
EBOH	3760	MSR_C11_PMON_BOX_CTL	Package	Uncore C-Box 11 Perfmon Local Box Wide Control
EB1H	3761	MSR_C11_PMON_EVNTSELO	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 0
EB2H	3762	MSR_C11_PMON_EVNTSEL1	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 1
EB3H	3763	MSR_C11_PMON_EVNTSEL2	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 2
EB4H	3764	MSR_C11_PMON_EVNTSEL3	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 3
EB5H	3765	MSR_C11_PMON_BOX_FILTER0	Package	Uncore C-Box 11 Perfmon Box Wide Filter 0
EB6H	3766	MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-Box 11 Perfmon Box Wide Filter 1
EB7H	3767	MSR_C11_PMON_BOX_STATUS	Package	Uncore C-Box 11 Perfmon Box Wide Status
EB8H	3768	MSR_C11_PMON_CTR0	Package	Uncore C-Box 11 Perfmon Counter 0
EB9H	3769	MSR_C11_PMON_CTR1	Package	Uncore C-Box 11 Perfmon Counter 1
EBAH	3770	MSR_C11_PMON_CTR2	Package	Uncore C-Box 11 Perfmon Counter 2
EBBH	3771	MSR_C11_PMON_CTR3	Package	Uncore C-Box 11 Perfmon Counter 3
EC0H	3776	MSR_C12_PMON_BOX_CTL	Package	Uncore C-Box 12 Perfmon Local Box Wide Control
EC1H	3777	MSR_C12_PMON_EVNTSELO	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 0
EC2H	3778	MSR_C12_PMON_EVNTSEL1	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 1
EC3H	3779	MSR_C12_PMON_EVNTSEL2	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 2
EC4H	3780	MSR_C12_PMON_EVNTSEL3	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 3
EC5H	3781	MSR_C12_PMON_BOX_FILTER0	Package	Uncore C-Box 12 Perfmon Box Wide Filter 0
EC6H	3782	MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-Box 12 Perfmon Box Wide Filter 1
EC7H	3783	MSR_C12_PMON_BOX_STATUS	Package	Uncore C-Box 12 Perfmon Box Wide Status
EC8H	3784	MSR_C12_PMON_CTR0	Package	Uncore C-Box 12 Perfmon Counter 0
EC9H	3785	MSR_C12_PMON_CTR1	Package	Uncore C-Box 12 Perfmon Counter 1
ECAH	3786	MSR_C12_PMON_CTR2	Package	Uncore C-Box 12 Perfmon Counter 2
ECBH	3787	MSR_C12_PMON_CTR3	Package	Uncore C-Box 12 Perfmon Counter 3

**Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
ED0H	3792	MSR_C13_PMON_BOX_CTL	Package	Uncore C-Box 13 Perfmon local box wide control.
ED1H	3793	MSR_C13_PMON_EVNTSELO	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 0
ED2H	3794	MSR_C13_PMON_EVNTSEL1	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 1
ED3H	3795	MSR_C13_PMON_EVNTSEL2	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 2
ED4H	3796	MSR_C13_PMON_EVNTSEL3	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3
ED5H	3797	MSR_C13_PMON_BOX_FILTER0	Package	Uncore C-Box 13 Perfmon Box Wide Filter 0
ED6H	3798	MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-Box 13 Perfmon Box Wide Filter 1
ED7H	3799	MSR_C13_PMON_BOX_STATUS	Package	Uncore C-Box 13 Perfmon Box Wide Status
ED8H	3800	MSR_C13_PMON_CTR0	Package	Uncore C-Box 13 Perfmon Counter 0
ED9H	3801	MSR_C13_PMON_CTR1	Package	Uncore C-Box 13 Perfmon Counter 1
EDAH	3802	MSR_C13_PMON_CTR2	Package	Uncore C-Box 13 Perfmon Counter 2
EDBH	3803	MSR_C13_PMON_CTR3	Package	Uncore C-Box 13 Perfmon Counter 3
EE0H	3808	MSR_C14_PMON_BOX_CTL	Package	Uncore C-Box 14 Perfmon Local Box Wide Control
EE1H	3809	MSR_C14_PMON_EVNTSELO	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 0
EE2H	3810	MSR_C14_PMON_EVNTSEL1	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 1
EE3H	3811	MSR_C14_PMON_EVNTSEL2	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 2
EE4H	3812	MSR_C14_PMON_EVNTSEL3	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 3
EE5H	3813	MSR_C14_PMON_BOX_FILTER	Package	Uncore C-Box 14 Perfmon Box Wide Filter 0
EE6H	3814	MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-Box 14 Perfmon Box Wide Filter 1
EE7H	3815	MSR_C14_PMON_BOX_STATUS	Package	Uncore C-Box 14 Perfmon Box Wide Status
EE8H	3816	MSR_C14_PMON_CTR0	Package	Uncore C-Box 14 Perfmon Counter 0
EE9H	3817	MSR_C14_PMON_CTR1	Package	Uncore C-Box 14 Perfmon Counter 1
EEAH	3818	MSR_C14_PMON_CTR2	Package	Uncore C-Box 14 Perfmon Counter 2
EEBH	3819	MSR_C14_PMON_CTR3	Package	Uncore C-Box 14 Perfmon Counter 3
EF0H	3824	MSR_C15_PMON_BOX_CTL	Package	Uncore C-Box 15 Perfmon Local Box Wide Control
EF1H	3825	MSR_C15_PMON_EVNTSELO	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 0
EF2H	3826	MSR_C15_PMON_EVNTSEL1	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 1
EF3H	3827	MSR_C15_PMON_EVNTSEL2	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 2
EF4H	3828	MSR_C15_PMON_EVNTSEL3	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
EF5H	3829	MSR_C15_PMON_BOX_FILTER0	Package	Uncore C-Box 15 Perfmon Box Wide Filter 0
EF6H	3830	MSR_C15_PMON_BOX_FILTER1	Package	Uncore C-Box 15 Perfmon Box Wide Filter 1
EF7H	3831	MSR_C15_PMON_BOX_STATUS	Package	Uncore C-Box 15 Perfmon Box Wide Status
EF8H	3832	MSR_C15_PMON_CTR0	Package	Uncore C-Box 15 Perfmon Counter 0
EF9H	3833	MSR_C15_PMON_CTR1	Package	Uncore C-Box 15 Perfmon Counter 1
EFAH	3834	MSR_C15_PMON_CTR2	Package	Uncore C-Box 15 Perfmon Counter 2
EFBH	3835	MSR_C15_PMON_CTR3	Package	Uncore C-Box 15 Perfmon Counter 3
FO0H	3840	MSR_C16_PMON_BOX_CTL	Package	Uncore C-Box 16 Perfmon for Box-Wide Control
F01H	3841	MSR_C16_PMON_EVNTSELO	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 0
F02H	3842	MSR_C16_PMON_EVNTSEL1	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 1
F03H	3843	MSR_C16_PMON_EVNTSEL2	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 2
F04H	3844	MSR_C16_PMON_EVNTSEL3	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 3
F05H	3845	MSR_C16_PMON_BOX_FILTER0	Package	Uncore C-Box 16 Perfmon Box Wide Filter 0
F06H	3846	MSR_C16_PMON_BOX_FILTER1	Package	Uncore C-Box 16 Perfmon Box Wide Filter 1
F07H	3847	MSR_C16_PMON_BOX_STATUS	Package	Uncore C-Box 16 Perfmon Box Wide Status
F08H	3848	MSR_C16_PMON_CTR0	Package	Uncore C-Box 16 Perfmon Counter 0
F09H	3849	MSR_C16_PMON_CTR1	Package	Uncore C-Box 16 Perfmon Counter 1
F0AH	3850	MSR_C16_PMON_CTR2	Package	Uncore C-Box 16 Perfmon Counter 2
F0BH	3851	MSR_C16_PMON_CTR3	Package	Uncore C-Box 16 Perfmon Counter 3
F10H	3856	MSR_C17_PMON_BOX_CTL	Package	Uncore C-Box 17 Perfmon for Box-Wide Control
F11H	3857	MSR_C17_PMON_EVNTSELO	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 0
F12H	3858	MSR_C17_PMON_EVNTSEL1	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 1
F13H	3859	MSR_C17_PMON_EVNTSEL2	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 2
F14H	3860	MSR_C17_PMON_EVNTSEL3	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 3
F15H	3861	MSR_C17_PMON_BOX_FILTER0	Package	Uncore C-Box 17 Perfmon Box Wide Filter 0
F16H	3862	MSR_C17_PMON_BOX_FILTER1	Package	Uncore C-Box 17 Perfmon Box Wide Filter1
F17H	3863	MSR_C17_PMON_BOX_STATUS	Package	Uncore C-Box 17 Perfmon Box Wide Status
F18H	3864	MSR_C17_PMON_CTR0	Package	Uncore C-Box 17 Perfmon Counter 0
F19H	3865	MSR_C17_PMON_CTR1	Package	Uncore C-Box 17 Perfmon Counter 1
F1AH	3866	MSR_C17_PMON_CTR2	Package	Uncore C-Box 17 Perfmon Counter 2
F1BH	3867	MSR_C17_PMON_CTR3	Package	Uncore C-Box 17 Perfmon Counter 3

## 2.15 MSRS IN INTEL® CORE™ M PROCESSORS AND 5TH GENERATION INTEL CORE PROCESSORS

The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors, and Intel® Xeon® Processor E3-1200 v4 family are based on the Broadwell microarchitecture. The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors have CPUID DisplayFamily\_DisplayModel signature 06\_3DH. Intel® Xeon® Processor E3-1200 v4 family and the 5th generation Intel® Core™ Processors have CPUID DisplayFamily\_DisplayModel signature 06\_47H. Processors with signatures 06\_3DH and 06\_47H support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, Table 2-34, and Table 2-35. For an MSR listed in Table 2-35 that also appears in the model-specific tables of prior generations, Table 2-35 supersedes prior generation tables.

Table 2-34 lists MSRs that are common to processors based on the Broadwell microarchitectures (including CPUID signatures 06\_3DH, 06\_47H, 06\_4FH, and 06\_56H).

**Table 2-34. Additional MSRs Common to Processors Based the Broadwell Microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2. See Section 18.6.2.2, “Global Counter Control Facilities.”
		0		Ovf_PMC0
		1		Ovf_PMC1
		2		Ovf_PMC2
		3		Ovf_PMC3
		31:4		Reserved
		32		Ovf_FixedCtr0
		33		Ovf_FixedCtr1
		34		Ovf_FixedCtr2
		54:35		Reserved
		55		Trace_ToPA_PMI See Section 31.2.6.2, “Table of Physical Addresses (ToPA).”
		60:56		Reserved
		61		Ovf_Uncore
		62		Ovf_BufDSSAVE
63		CondChgd		
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, “Global Counter Control Facilities.”
		0		Set 1 to clear Ovf_PMC0
		1		Set 1 to clear Ovf_PMC1
		2		Set 1 to clear Ovf_PMC2
		3		Set 1 to clear Ovf_PMC3
		31:4		Reserved
		32		Set 1 to clear Ovf_FixedCtr0
		33		Set 1 to clear Ovf_FixedCtr1

Table 2-34. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		34		Set 1 to clear Ovf_FixedCtr2
		54:35		Reserved.
		55		Set 1 to clear Trace_ToPA_PMI. See Section 31.2.6.2, "Table of Physical Addresses (ToPA)."
		60:56		Reserved
		61		Set 1 to clear Ovf_Uncore
		62		Set 1 to clear Ovf_BufDSSAVE
		63		Set 1 to clear CondChgd
560H	1376	IA32_RTIT_OUTPUT_BASE	THREAD	Trace Output Base Register (R/W)
		6:0		Reserved
		MAXPHYADDR <sup>1</sup> -1:7		Base physical address.
		63:MAXPHYADDR		Reserved
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	THREAD	Trace Output Mask Pointers Register (R/W)
		6:0		Reserved
		31:7		MaskOrTableOffset
		63:32		Output Offset.
570H	1392	IA32_RTIT_CTL	Thread	Trace Control Register (R/W)
		0		TraceEn
		1		Reserved, must be zero.
		2		OS
		3		User
		6:4		Reserved, must be zero.
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		Reserved, must be zero.
		10		TSCEn
		11		DisRETC
		12		Reserved, must be zero.
		13		Reserved; writing 0 will #GP if also setting TraceEn.
		63:14		Reserved, must be zero.
571H	1393	IA32_RTIT_STATUS	Thread	Tracing Status Register (R/W)
		0		Reserved, writes ignored.
		1		ContexEn, writes ignored.
		2		TriggerEn, writes ignored.
		3		Reserved

**Table 2-34. Additional MSRs Common to Processors Based the Broadwell Microarchitectures**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		4		Error (R/W)
		5		Stopped
		63:6		Reserved, must be zero.
572H	1394	IA32_RTIT_CR3_MATCH	THREAD	Trace Filter CR3 Match Register (R/W)
		4:0		Reserved
		63:5		CR3[63:5] value to match.
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.

**NOTES:**

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 2-35 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

**Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .

Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		I/O MWAIT Redirection Enable (R/W)
		14:11		Reserved
		15		CFG Lock (R/W0)
		24:16		Reserved
		25		C3 State Auto Demotion Enable (R/W)
		26		C1 State Auto Demotion Enable (R/W)
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		29		Enable Package C-State Auto-Demotion (R/W)
		30		Enable Package C-State Undemotion (R/W)
		63:31		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.



**Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		39:32	Package	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5core active.
		47:40	Package	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6core active.
		63:48		Reserved
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."

See Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, Table 2-34 for other MSR definitions applicable to processors with CPUID signature 06\_3DH.

## 2.16 MSRS IN INTEL® XEON® PROCESSORS E5 V4 FAMILY

The MSRs listed in Table 2-36 are available and common to Intel® Xeon® Processor D product Family (CPUID DisplayFamily\_DisplayModel = 06\_56H) and to Intel Xeon processors E5 v4, E7 v4 families (CPUID DisplayFamily\_DisplayModel = 06\_4FH). They are based on the Broadwell microarchitecture.

See Section 2.16.1 for lists of tables of MSRs that are supported by Intel® Xeon® Processor D Family.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/W/O) See Table 2-26.
		1		Enable_PPIN (R/W) See Table 2-26.
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		Protected Processor Inventory Number (R/O) See Table 2-26.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) See Table 2-26.
		22:16		Reserved.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		23	Package	PPIN_CAP (R/O) See Table 2-26.
		27:24		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) See Table 2-26.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) See Table 2-26.
		30	Package	Programmable TJ OFFSET (R/O) See Table 2-26.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) See Table 2-26.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W)
		14:11		Reserved
		15		CFG Lock (R/WO)
		16		Automatic C-State Conversion Enable (R/W) If 1, the processor will convert HALT or MWAIT(C1) to MWAIT(C6).
		24:17		Reserved
		25		C3 State Auto Demotion Enable (R/W)

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		26		C1 State Auto Demotion Enable (R/W)
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		29		Package C State Demotion Enable (R/W)
		30		Package C State UnDemotion Enable (R/W)
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	Global Machine Check Capability (R/O)
		7:0		Count
		8		MCG_CTL_P
		9		MCG_EXT_P
		10		MCP_CMCI_P
		11		MCG_TES_P
		15:12		Reserved
		23:16		MCG_EXT_CNT
		24		MCG_SER_P
		25		MCG_EM_P
		26		MCG_ELOG_P
		63:27		Reserved
17DH	381	MSR_SMM_MCA_CAP	Thread	Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		Reserved
		58		SMM_Code_Access_Chk (SMM-RO) If set to 1, indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
		0		Thermal Status (RO) See Table 2-2.
		1		Thermal Status Log (R/WCO) See Table 2-2.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.
		6		Thermal Threshold #1 Status (RO) See Table 2-2.
		7		Thermal Threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
		11		Power Limitation Log (R/WCO) See Table 2-2.
		12		Current Limit Status (RO) See Table 2-2.
		13		Current Limit Log (R/WCO) See Table 2-2.
		14		Cross Domain Limit Status (RO) See Table 2-2.
		15		Cross Domain Limit Log (R/WCO) See Table 2-2.
		22:16		Digital Readout (RO) See Table 2-2.
		26:23		Reserved
		30:27		Resolution in Degrees Celsius (RO) See Table 2-2.
31		Reading Valid (RO) See Table 2-2.		
63:32		Reserved		
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		23:16		Temperature Target (RO) See Table 2-26.
		27:24		TCC Activation Offset (R/W) See Table 2-26.
		63:28		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 1C
		15:8	Package	Maximum Ratio Limit for 2C
		23:16	Package	Maximum Ratio Limit for 3C
		31:24	Package	Maximum Ratio Limit for 4C
		39:32	Package	Maximum Ratio Limit for 5C
		47:40	Package	Maximum Ratio Limit for 6C
		55:48	Package	Maximum Ratio Limit for 7C
		63:56	Package	Maximum Ratio Limit for 8C
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		7:0	Package	Maximum Ratio Limit for 9C
		15:8	Package	Maximum Ratio Limit for 10C
		23:16	Package	Maximum Ratio Limit for 11C
		31:24	Package	Maximum Ratio Limit for 12C
		39:32	Package	Maximum Ratio Limit for 13C
		47:40	Package	Maximum Ratio Limit for 14C
		55:48	Package	Maximum Ratio Limit for 15C
		63:56	Package	Maximum Ratio Limit for 16C
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers Used in RAPL Interfaces (R/O)
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
		15:13	Package	Reserved

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		19:16	Package	Time Units See Section 14.10.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) Energy consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	Reserved (R/O) Reads return 0.
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)
		0		PROCHOT Status (R/O) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (R/O) When set, frequency is reduced below the operating system request due to a thermal event.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2		Power Budget Management Status (R0) When set, frequency is reduced below the operating system request due to PBM limit.
		3		Platform Configuration Services Status (R0) When set, frequency is reduced below the operating system request due to PCS limit.
		4		Reserved
		5		Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved
		8		Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).
		9		Reserved
		10		Multi-Core Turbo Status (R0) When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits.
		12:11		Reserved
		13		Core Frequency P1 Status (R0) When set, frequency is reduced below max non-turbo P1.
		14		Core Max N-Core Turbo Frequency Limiting Status (R0) When set, frequency is reduced below max n-core turbo frequency.
		15		Core Frequency Limiting Status (R0) When set, frequency is reduced below the operating system request.
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		Power Budget Management Log When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		Platform Configuration Services Log When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved
		21		Autonomous Utilization-Based Frequency Control Log When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved
		24		Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved
		26		Multi-Core Turbo Log When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved



**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		29		Core Frequency P1 Log When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		Core Max N-Core Turbo Frequency Limiting Log When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		Core Frequency Limiting Log When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, "Enabling HWP".
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities".
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, "Managing HWP".
		7:0		Minimum Performance (R/W)
		15:8		Maximum Performance (R/W)
		23:16		Desired Performance (R/W)
		63:24		Reserved
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, "HWP Feedback".
		1:0		Reserved
		2		Excursion to Minimum (RO)
		63:3		Reserved
C8DH	3213	IA32_QM_EVTSEL	THREAD	Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.
		7:0		EventID (RW) Event encoding: 0x00: No monitoring. 0x01: L3 occupancy monitoring. 0x02: Total memory bandwidth monitoring. 0x03: Local memory bandwidth monitoring. All other encoding reserved.
		31:8		Reserved

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		41:32		RMID (Rw)
		63:42		Reserved
C8FH	3215	IA32_PQR_ASSOC	THREAD	Resource Association Register (R/W)
		9:0		RMID
		31:10		Reserved
		51:32		COS (R/W)
		63: 52		Reserved
C90H	3216	IA32_L3_QOS_MASK_0	Package	L3 Class Of Service Mask - COS 0 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0.
		0:19		CBM: Bit vector of available L3 ways for COS 0 enforcement.
		63:20		Reserved
C91H	3217	IA32_L3_QOS_MASK_1	Package	L3 Class Of Service Mask - COS 1 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1.
		0:19		CBM: Bit vector of available L3 ways for COS 1 enforcement.
		63:20		Reserved
C92H	3218	IA32_L3_QOS_MASK_2	Package	L3 Class Of Service Mask - COS 2 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2.
		0:19		CBM: Bit vector of available L3 ways for COS 2 enforcement.
		63:20		Reserved
C93H	3219	IA32_L3_QOS_MASK_3	Package	L3 Class Of Service Mask - COS 3 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3.
		0:19		CBM: Bit vector of available L3 ways for COS 3 enforcement.
		63:20		Reserved
C94H	3220	IA32_L3_QOS_MASK_4	Package	L3 Class Of Service Mask - COS 4 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4.
		0:19		CBM: Bit vector of available L3 ways for COS 4 enforcement.
		63:20		Reserved
C95H	3221	IA32_L3_QOS_MASK_5	Package	L3 Class Of Service Mask - COS 5 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5.
		0:19		CBM: Bit vector of available L3 ways for COS 5 enforcement.
		63:20		Reserved
C96H	3222	IA32_L3_QOS_MASK_6	Package	L3 Class Of Service Mask - COS 6 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0:19		CBM: Bit vector of available L3 ways for COS 6 enforcement.
		63:20		Reserved
C97H	3223	IA32_L3_QOS_MASK_7	Package	L3 Class Of Service Mask - COS 7 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=7.
		0:19		CBM: Bit vector of available L3 ways for COS 7 enforcement.
		63:20		Reserved
C98H	3224	IA32_L3_QOS_MASK_8	Package	L3 Class Of Service Mask - COS 8 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=8.
		0:19		CBM: Bit vector of available L3 ways for COS 8 enforcement.
		63:20		Reserved
C99H	3225	IA32_L3_QOS_MASK_9	Package	L3 Class Of Service Mask - COS 9 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=9.
		0:19		CBM: Bit vector of available L3 ways for COS 9 enforcement.
		63:20		Reserved
C9AH	3226	IA32_L3_QOS_MASK_10	Package	L3 Class Of Service Mask - COS 10 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=10.
		0:19		CBM: Bit vector of available L3 ways for COS 10 enforcement.
		63:20		Reserved
C9BH	3227	IA32_L3_QOS_MASK_11	Package	L3 Class Of Service Mask - COS 11 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=11.
		0:19		CBM: Bit vector of available L3 ways for COS 11 enforcement.
		63:20		Reserved
C9CH	3228	IA32_L3_QOS_MASK_12	Package	L3 Class Of Service Mask - COS 12 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=12.
		0:19		CBM: Bit vector of available L3 ways for COS 12 enforcement.
		63:20		Reserved
C9DH	3229	IA32_L3_QOS_MASK_13	Package	L3 Class Of Service Mask - COS 13 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=13.
		0:19		CBM: Bit vector of available L3 ways for COS 13 enforcement.
		63:20		Reserved
C9EH	3230	IA32_L3_QOS_MASK_14	Package	L3 Class Of Service Mask - COS 14 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=14.

**Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0:19		CBM: Bit vector of available L3 ways for COS 14 enforcement.
		63:20		Reserved
C9FH	3231	IA32_L3_QOS_MASK_15	Package	L3 Class Of Service Mask - COS 15 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15.
		0:19		CBM: Bit vector of available L3 ways for COS 15 enforcement.
		63:20		Reserved

### 2.16.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 2-37 are available to Intel® Xeon® Processor D Product Family (CPUID DisplayFamily\_DisplayModel = 06\_56H). The Intel® Xeon® processor D product family is based on the Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-20, Table 2-29, Table 2-34, Table 2-36, and Table 2-37.

**Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily\_DisplayModel 06\_56H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	Config Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		62:0	Package	Reserved
		63	Package	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.

**Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily\_DisplayModel 06\_56H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
See Table 2-20, Table 2-29, Table 2-34, and Table 2-36 for other MSR definitions applicable to processors with CPUID signature 06_56H.				

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

**2.16.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families**

The MSRs listed in Table 2-37 are available to Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID DisplayFamily\_DisplayModel = 06\_4FH). The Intel® Xeon® processor E5 v4 family is based on the Broadwell micro-

architecture and supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-34, Table 2-36, and Table 2-38.

**Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	Config Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0. RW if MSR_PLATFORM_INFO.[28] = 1.
		62:0	Package	Reserved
		63	Package	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC errors from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

**Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

**Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC20 reports MC errors from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
C81H	3201	IA32_L3_QOS_CFG	Package	Cache Allocation Technology Configuration (R/W)
		0		CAT Enable. Set 1 to enable Cache Allocation Technology.
		63:1		Reserved

See Table 2-20, Table 2-21, Table 2-29, and Table 2-30 for other MSR definitions applicable to processors with CPUID signature 06\_45H.

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.



## 2.17 MSRS IN THE 6TH GENERATION, 7TH GENERATION, 8TH GENERATION, 9TH GENERATION, 10TH GENERATION, AND 11TH GENERATION INTEL® CORE™ PROCESSORS, INTEL® XEON® PROCESSOR SCALABLE FAMILY, 2ND AND 3RD GENERATION INTEL® XEON® PROCESSOR SCALABLE FAMILY, 8TH GENERATION INTEL® CORE™ i3 PROCESSORS, AND INTEL® XEON® E PROCESSORS

6th generation Intel® Core™ processors are based on the Skylake microarchitecture and have CPUID DisplayFamily\_DisplayModel signatures of 06\_4EH and 06\_5EH.

The Intel® Xeon® Processor Scalable Family based on the Skylake microarchitecture, the 2nd generation Intel® Xeon® Processor Scalable Family based on the Cascade Lake product, and the 3rd generation Intel® Xeon® Processor Scalable Family based on the Cooper Lake product all have a CPUID DisplayFamily\_DisplayModel signature of 06\_55H.

7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture, 8th generation and 9th generation Intel® Core™ processors and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture; these processors have CPUID DisplayFamily\_DisplayModel signatures of 06\_8EH and 06\_9EH.

8th generation Intel® Core™ i3 processors are based on Cannon Lake microarchitecture and have a CPUID DisplayFamily\_DisplayModel signature of 06\_66H.

10th generation Intel® Core™ processors are based on Comet Lake microarchitecture (with CPUID DisplayFamily\_DisplayModel signatures of 06\_A5H, 06\_A6H) and Ice Lake microarchitecture (with CPUID DisplayFamily\_DisplayModel signatures of 06\_7DH and 06\_7EH).

11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture and have CPUID DisplayFamily\_DisplayModel signatures of 06\_8CH and 06\_8DH.

The 3rd generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture have CPUID DisplayFamily\_DisplayModel signatures of 06\_6AH and 06\_6CH.

These processors support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-25, Table 2-29, Table 2-35, Table 2-39, and Table 2-40. For an MSR listed in Table 2-39 that also appears in the model-specific tables of prior generations, Table 2-39 supersedes prior generation tables.

The notation of “Platform” in the Scope column (with respect to MSR\_PLATFORM\_ENERGY\_COUNTER and MSR\_PLATFORM\_POWER\_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor’s implementation.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64 Processor (R/W) See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	MTRR Capability (RO, Architectural) See Table 2-2
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
		0		Thermal Status (RO) See Table 2-2.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		Thermal Status Log (R/WCO) See Table 2-2.
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.
		6		Thermal threshold #1 Status (RO) See Table 2-2.
		7		Thermal threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
		11		Power Limitation Log (R/WCO) See Table 2-2.
		12		Current Limit Status (RO) See Table 2-2.
		13		Current Limit Log (R/WCO) See Table 2-2.
		14		Cross Domain Limit Status (RO) See Table 2-2.
		15		Cross Domain Limit Log (R/WCO) See Table 2-2.
		22:16		Digital Readout (RO) See Table 2-2.
		26:23		Reserved
		30:27		Resolution in Degrees Celsius (RO) See Table 2-2.
		31		Reading Valid (RO) See Table 2-2.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:32		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.
		15:8	Package	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.
		63:32		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Thread	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		Reserved
		1	Package	C1E Enable (R/W) When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		18:2		Reserved
		19		Disable Energy Efficiency Optimization (R/W) Setting this bit disables the P-States energy efficiency optimization. Default value is 0. Disable/enable the energy efficiency optimization in P-State legacy mode (when IA32_PM_ENABLE[HWP_ENABLE] = 0), has an effect only in the turbo range or into PERF_MIN_CTL value if it is not zero set. In HWP mode (IA32_PM_ENABLE[HWP_ENABLE] == 1), has an effect between the OS desired or OS maximize to the OS minimize performance setting.
20		Disable Race to Halt Optimization (R/W) Setting this bit disables the Race to Halt optimization and avoids this optimization limitation to execute below the most efficient frequency ratio. Default value is 0 for processors that support Race to Halt optimization.		

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:21		Reserved
300H	768	MSR_SGXOWNEREPOCH0	Package	Lower 64 Bit CR_SGXOWNEREPOCH (w) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.
301H	768	MSR_SGXOWNEREPOCH1	Package	Upper 64 Bit CR_SGXOWNEREPOCH (w) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.
38EH	910	IA32_PERF_GLOBAL_STATUS		See Table 2-2. See Section 18.2.4, “Architectural Performance Monitoring Version 4.”
		0	Thread	Ovf_PMC0
		1	Thread	Ovf_PMC1
		2	Thread	Ovf_PMC2
		3	Thread	Ovf_PMC3
		4	Thread	Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)
		5	Thread	Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)
		6	Thread	Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)
		7	Thread	Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved
		32	Thread	Ovf_FixedCtr0
		33	Thread	Ovf_FixedCtr1
		34	Thread	Ovf_FixedCtr2
		54:35		Reserved
		55	Thread	Trace_ToPA_PMI
		57:56		Reserved
		58	Thread	LBR_Frz
		59	Thread	CTR_Frz
		60	Thread	ASCI
		61	Thread	Ovf_Uncore
62	Thread	Ovf_BufDSSAVE		
63	Thread	CondChgd		

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
390H	912	IA32_PERF_GLOBAL_STATUS_RESET		See Table 2-2. See Section 18.2.4, “Architectural Performance Monitoring Version 4.”
		0	Thread	Set 1 to clear Ovf_PMC0.
		1	Thread	Set 1 to clear Ovf_PMC1.
		2	Thread	Set 1 to clear Ovf_PMC2.
		3	Thread	Set 1 to clear Ovf_PMC3.
		4	Thread	Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4).
		5	Thread	Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5).
		6	Thread	Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6).
		7	Thread	Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7).
		31:8		Reserved
		32	Thread	Set 1 to clear Ovf_FixedCtr0.
		33	Thread	Set 1 to clear Ovf_FixedCtr1.
		34	Thread	Set 1 to clear Ovf_FixedCtr2.
		54:35		Reserved
		55	Thread	Set 1 to clear Trace_ToPA_PMI.
		57:56		Reserved
		58	Thread	Set 1 to clear LBR_Frz.
		59	Thread	Set 1 to clear CTR_Frz.
		60	Thread	Set 1 to clear ASCI.
		61	Thread	Set 1 to clear Ovf_Uncore.
62	Thread	Set 1 to clear Ovf_BufDSSAVE.		
63	Thread	Set 1 to clear CondChgd.		
391H	913	IA32_PERF_GLOBAL_STATUS_SET		See Table 2-2. See Section 18.2.4, “Architectural Performance Monitoring Version 4.”
		0	Thread	Set 1 to cause Ovf_PMC0 = 1.
		1	Thread	Set 1 to cause Ovf_PMC1 = 1.
		2	Thread	Set 1 to cause Ovf_PMC2 = 1.
		3	Thread	Set 1 to cause Ovf_PMC3 = 1.
		4	Thread	Set 1 to cause Ovf_PMC4=1 (if CPUID.0AH:EAX[15:8] > 4).
		5	Thread	Set 1 to cause Ovf_PMC5=1 (if CPUID.0AH:EAX[15:8] > 5).
		6	Thread	Set 1 to cause Ovf_PMC6=1 (if CPUID.0AH:EAX[15:8] > 6).

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7	Thread	Set 1 to cause Ovf_PMC7=1 (if CPUID.0AH:EAX[15:8] > 7).
		31:8		Reserved
		32	Thread	Set 1 to cause Ovf_FixedCtr0 = 1.
		33	Thread	Set 1 to cause Ovf_FixedCtr1 = 1.
		34	Thread	Set 1 to cause Ovf_FixedCtr2 = 1.
		54:35		Reserved
		55	Thread	Set 1 to cause Trace_ToPA_PMI = 1.
		57:56		Reserved
		58	Thread	Set 1 to cause LBR_Frz = 1.
		59	Thread	Set 1 to cause CTR_Frz = 1.
		60	Thread	Set 1 to cause ASCI = 1.
		61	Thread	Set 1 to cause Ovf_Uncore.
		62	Thread	Set 1 to cause Ovf_BufDSSAVE.
		63		Reserved
392H	913	IA32_PERF_GLOBAL_INUSE	Thread	See Table 2-2.
3F7H	1015	MSR_PEBS_FRONTEND	Thread	FrontEnd Precise Event Condition Select (R/W)
		2:0		Event Code Select
		3		Reserved
		4		Event Code Select High
		7:5		Reserved
		19:8		IDQ_Bubble_Length Specifier
		22:20		IDQ_Bubble_Width Specifier
		63:23		Reserved
500H	1280	IA32_SGX_SVN_STATUS	Thread	Status and SVN Threshold of SGX Support for ACM (RO)
		0		Lock See Section 37.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1		Reserved
		23:16		SGX_SVN_SINIT See Section 37.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		63:24		Reserved
560H	1376	IA32_RTIT_OUTPUT_BASE	Thread	Trace Output Base Register (R/W) See Table 2-2.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Thread	Trace Output Mask Pointers Register (R/W) See Table 2-2.
570H	1392	IA32_RTIT_CTL	Thread	Trace Control Register (R/W)
		0		TraceEn
		1		CYCEn
		2		OS
		3		User
		6:4		Reserved, must be zero.
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		MTCEn
		10		TSCEn
		11		DisRETC
		12		Reserved, must be zero.
		13		BranchEn
		17:14		MTCFreq
		18		Reserved, must be zero.
		22:19		CYCThresh
		23		Reserved, must be zero.
		27:24		PSBFreq
		31:28		Reserved, must be zero.
		35:32		ADDR0_CFG
39:36		ADDR1_CFG		
63:40		Reserved, must be zero.		
571H	1393	IA32_RTIT_STATUS	Thread	Tracing Status Register (R/W)
		0		FilterEn, writes ignored.
		1		ContexEn, writes ignored.
		2		TriggerEn, writes ignored.
		3		Reserved
		4		Error (R/W)
		5		Stopped
		31:6		Reserved, must be zero.
		48:32		PacketByteCnt
		63:49		Reserved, must be zero.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
572H	1394	IA32_RTIT_CR3_MATCH	Thread	Trace Filter CR3 Match Register (R/W)
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Thread	Region 0 Start Address (R/W)
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Thread	Region 0 End Address (R/W)
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Thread	Region 1 Start Address (R/W)
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Thread	Region 1 End Address (R/W)
		63:0		See Table 2-2.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
64DH	1613	MSR_PLATFORM_ENERGY_COUNTER	Platform*	Platform Energy Counter (R/O) This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.
		31:0		Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.Energy_Status_Unit.
		63:32		Reserved
64EH	1614	MSR_PPERF	Thread	Productive Performance Count (R/O)
		63:0		Hardware's view of workload scalability. See Section 14.4.5.1.
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)
		0		PROCHOT Status (R/O) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (R/O) When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved



**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		4		Residency State Regulation Status (R0) When set, frequency is reduced below the operating system request due to residency state regulation limit.
		5		Running Average Thermal Limit Status (R0) When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).
		6		VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).
		7		VR Therm Design Current Status (R0) When set, frequency is reduced below the operating system request due to VR thermal design current limit.
		8		Other Status (R0) When set, frequency is reduced below the operating system request due to electrical or other constraints.
		9		Reserved
		10		Package/Platform-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.
		11		Package/Platform-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.
		12		Max Turbo Limit Status (R0) When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		Turbo Transition Attenuation Status (R0) When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		Residency State Regulation Log When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		Running Average Thermal Limit Log When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		VR Thermal Design Current Log When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		Other Log When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved
		26		Package/Platform-Level PL1 Power Limiting Log When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		27		Package/Platform-Level PL2 Power Limiting Log When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved
652H	1618	MSR_PKG_HDC_CONFIG	Package	HDC Configuration (R/W)
		2:0		PKG_Cx_Monitor Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY.
		63:3		Reserved
653H	1619	MSR_CORE_HDC_RESIDENCY	Core	Core HDC Idle Residency (R/O)
		63:0		Core_Cx_Duty_Cycle_Cnt
655H	1621	MSR_PKG_HDC_SHALLOW_RESIDENCY	Package	Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle (R/O)
		63:0		Pkg_C2_Duty_Cycle_Cnt
656H	1622	MSR_PKG_HDC_DEEP_RESIDENCY	Package	Package Cx HDC Idle Residency (R/O)
		63:0		Pkg_Cx_Duty_Cycle_Cnt
658H	1624	MSR_WEIGHTED_CORE_CO	Package	Core-count Weighted C0 Residency (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.
659H	1625	MSR_ANY_CORE_CO	Package	Any Core C0 Residency (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
65AH	1626	MSR_ANY_GFXE_CO	Package	Any Graphics Engine C0 Residency (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.
65BH	1627	MSR_CORE_GFXE_OVERLAP_CO	Package	Core and Graphics Engine Overlapped C0 Residency (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.
65CH	1628	MSR_PLATFORM_POWER_LIMIT	Platform*	Platform Power Limit Control (R/W-L) Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.
		14:0		Platform Power Limit #1 Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.
		15		Enable Platform Power Limit #1 When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #1 over the time window specified by Power Limit #1 Time Window.
		16		Platform Clamping Limitation #1 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #1 value. This bit is writeable only when CPUID (EAX=6):EAX[4] is set.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		23:17		Time Window for Platform Power Limit #1 Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation: Time Window = (float) ((1+(X/4))*(2^Y)), where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17] The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is 0DH, The unit is specified in MSR_RAPLPOWER_UNIT[Time Unit].
		31:24		Reserved
		46:32		Platform Power Limit #2 Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e., Platform Power Limit # 1).
		47		Enable Platform Power Limit #2 When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.
		48		Platform Clamping Limitation #2 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.
		62:49		Reserved
		63		Lock. Setting this bit will lock all other bits of this MSR until system RESET.
690H	1680	MSR_LASTBRANCH_16_FROM_IP	Thread	Last Branch Record 16 From IP (R/W) One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>Last Branch Record Stack TOS at 1C9H.</li> <li>Section 17.12.</li> </ul>
691H	1681	MSR_LASTBRANCH_17_FROM_IP	Thread	Last Branch Record 17 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_LASTBRANCH_18_FROM_IP	Thread	Last Branch Record 18 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
693H	1683	MSR_LASTBRANCH_19_FROM_IP	Thread	Last Branch Record 19 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_LASTBRANCH_20_FROM_IP	Thread	Last Branch Record 20 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_LASTBRANCH_21_FROM_IP	Thread	Last Branch Record 21 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_LASTBRANCH_22_FROM_IP	Thread	Last Branch Record 22 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_LASTBRANCH_23_FROM_IP	Thread	Last Branch Record 23 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_LASTBRANCH_24_FROM_IP	Thread	Last Branch Record 24 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_LASTBRANCH_25_FROM_IP	Thread	Last Branch Record 25 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_LASTBRANCH_26_FROM_IP	Thread	Last Branch Record 26 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_LASTBRANCH_27_FROM_IP	Thread	Last Branch Record 27 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_LASTBRANCH_28_FROM_IP	Thread	Last Branch Record 28 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_LASTBRANCH_29_FROM_IP	Thread	Last Branch Record 29 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_LASTBRANCH_30_FROM_IP	Thread	Last Branch Record 30 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_LASTBRANCH_31_FROM_IP	Thread	Last Branch Record 31 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.
6BOH	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in the Processor Graphics (R/W) (Frequency refers to processor graphics frequency.)
		0		PROCHOT Status (RO) When set, frequency is reduced due to assertion of external PROCHOT.
		1		Thermal Status (RO) When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		Running Average Thermal Limit Status (RO) When set, frequency is reduced due to running average thermal limit.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		6		VR Therm Alert Status (R0) When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		VR Thermal Design Current Status (R0) When set, frequency is reduced due to VR TDC limit.
		8		Other Status (R0) When set, frequency is reduced due to electrical or other constraints.
		9		Reserved
		10		Package/Platform-Level Power Limiting PL1 Status (R0) When set, frequency is reduced due to package/platform-level power limiting PL1.
		11		Package/Platform-Level PL2 Power Limiting Status (R0) When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.
		12		Inefficient Operation Status (R0) When set, processor graphics frequency is operating below target frequency.
		15:13		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		Running Average Thermal Limit Log When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		VR Thermal Design Current Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		Other Log When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved
		26		Package/Platform-Level PL1 Power Limiting Log When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		Package/Platform-Level PL2 Power Limiting Log When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		Inefficient Operation Log When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:29		Reserved
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in the Ring Interconnect (R/W) (Frequency refers to ring interconnect in the uncore.)
		0		PROCHOT Status (RO) When set, frequency is reduced due to assertion of external PROCHOT.



**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		Thermal Status (RO) When set, frequency is reduced due to a thermal event.
		4:2		Reserved
		5		Running Average Thermal Limit Status (RO) When set, frequency is reduced due to running average thermal limit.
		6		VR Therm Alert Status (RO) When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		VR Thermal Design Current Status (RO) When set, frequency is reduced due to VR TDC limit.
		8		Other Status (RO) When set, frequency is reduced due to electrical or other constraints.
		9		Reserved
		10		Package/Platform-Level Power Limiting PL1 Status (RO) When set, frequency is reduced due to package/Platform-level power limiting PL1.
		11		Package/Platform-Level PL2 Power Limiting Status (RO) When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.
		15:12		Reserved
		16		PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved
		21		Running Average Thermal Limit Log When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		22		VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		VR Thermal Design Current Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		Other Log When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved
		26		Package/Platform-Level PL1 Power Limiting Log When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		Package/Platform-Level PL2 Power Limiting Log When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:28		Reserved
6D0H	1744	MSR_LASTBRANCH_16_TO_IP	Thread	Last Branch Record 16 To IP (R/W) One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.12.</li> </ul>
6D1H	1745	MSR_LASTBRANCH_17_TO_IP	Thread	Last Branch Record 17 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_LASTBRANCH_18_TO_IP	Thread	Last Branch Record 18 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_LASTBRANCH_19_TO_IP	Thread	Last Branch Record 19To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
6D4H	1748	MSR_LASTBRANCH_20_TO_IP	Thread	Last Branch Record 20 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_LASTBRANCH_21_TO_IP	Thread	Last Branch Record 21 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_LASTBRANCH_22_TO_IP	Thread	Last Branch Record 22 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_LASTBRANCH_23_TO_IP	Thread	Last Branch Record 23 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_LASTBRANCH_24_TO_IP	Thread	Last Branch Record 24 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_LASTBRANCH_25_TO_IP	Thread	Last Branch Record 25 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_LASTBRANCH_26_TO_IP	Thread	Last Branch Record 26 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_LASTBRANCH_27_TO_IP	Thread	Last Branch Record 27 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_LASTBRANCH_28_TO_IP	Thread	Last Branch Record 28 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_LASTBRANCH_29_TO_IP	Thread	Last Branch Record 29 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_LASTBRANCH_30_TO_IP	Thread	Last Branch Record 30 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_LASTBRANCH_31_TO_IP	Thread	Last Branch Record 31 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, "Enabling HWP".
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities".
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Section 14.4.4, "Managing HWP".
773H	1907	IA32_HWP_INTERRUPT	Thread	See Section 14.4.6, "HWP Notifications".
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, "Managing HWP".
		7:0		Minimum Performance (R/W)
		15:8		Maximum Performance (R/W)
		23:16		Desired Performance (R/W)
		31:24		Energy/Performance Preference (R/W)
		41:32		Activity Window (R/W)
		42		Package Control (R/W)

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:43		Reserved
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, "HWP Feedback".
D90H	3472	IA32_BNDCFGS	Thread	See Table 2-2.
DA0H	3488	IA32_XSS	Thread	See Table 2-2.
DB0H	3504	IA32_PKG_HDC_CTL	Package	See Section 14.5.2, "Package level Enabling HDC".
DB1H	3505	IA32_PM_CTL1	Thread	See Section 14.5.3, "Logical-Processor Level HDC Control".
DB2H	3506	IA32_THREAD_STALL	Thread	See Section 14.5.4.1, "IA32_THREAD_STALL".
DC0H	3520	MSR_LBR_INFO_0	Thread	Last Branch Record 0 Additional Information (R/W) One of 32 triplet of last branch record registers on the last branch record stack. This part of the stack contains flag, TSX-related and elapsed cycle information. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.9.1, "LBR Stack."</li> </ul>
DC1H	3521	MSR_LBR_INFO_1	Thread	Last Branch Record 1 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC2H	3522	MSR_LBR_INFO_2	Thread	Last Branch Record 2 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC3H	3523	MSR_LBR_INFO_3	Thread	Last Branch Record 3 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC4H	3524	MSR_LBR_INFO_4	Thread	Last Branch Record 4 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC5H	3525	MSR_LBR_INFO_5	Thread	Last Branch Record 5 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC6H	3526	MSR_LBR_INFO_6	Thread	Last Branch Record 6 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC7H	3527	MSR_LBR_INFO_7	Thread	Last Branch Record 7 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC8H	3528	MSR_LBR_INFO_8	Thread	Last Branch Record 8 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DC9H	3529	MSR_LBR_INFO_9	Thread	Last Branch Record 9 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DCAH	3530	MSR_LBR_INFO_10	Thread	Last Branch Record 10 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DCBH	3531	MSR_LBR_INFO_11	Thread	Last Branch Record 11 Additional Information (R/W) See description of MSR_LBR_INFO_0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DCCH	3532	MSR_LBR_INFO_12	Thread	Last Branch Record 12 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DCDH	3533	MSR_LBR_INFO_13	Thread	Last Branch Record 13 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DCEH	3534	MSR_LBR_INFO_14	Thread	Last Branch Record 14 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DCFH	3535	MSR_LBR_INFO_15	Thread	Last Branch Record 15 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DDOH	3536	MSR_LBR_INFO_16	Thread	Last Branch Record 16 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD1H	3537	MSR_LBR_INFO_17	Thread	Last Branch Record 17 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD2H	3538	MSR_LBR_INFO_18	Thread	Last Branch Record 18 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD3H	3539	MSR_LBR_INFO_19	Thread	Last Branch Record 19 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD4H	3520	MSR_LBR_INFO_20	Thread	Last Branch Record 20 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD5H	3521	MSR_LBR_INFO_21	Thread	Last Branch Record 21 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD6H	3522	MSR_LBR_INFO_22	Thread	Last Branch Record 22 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD7H	3523	MSR_LBR_INFO_23	Thread	Last Branch Record 23 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD8H	3524	MSR_LBR_INFO_24	Thread	Last Branch Record 24 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DD9H	3525	MSR_LBR_INFO_25	Thread	Last Branch Record 25 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DDAH	3526	MSR_LBR_INFO_26	Thread	Last Branch Record 26 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DDBH	3527	MSR_LBR_INFO_27	Thread	Last Branch Record 27 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DDCH	3528	MSR_LBR_INFO_28	Thread	Last Branch Record 28 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DDDH	3529	MSR_LBR_INFO_29	Thread	Last Branch Record 29 Additional Information (R/W) See description of MSR_LBR_INFO_0.

**Table 2-39. Additional MSRs Supported by 6th Generation, 7th Generation, 8th Generation, 9th Generation, 10th Generation, and 11th Generation Intel® Core™ Processors, Intel® Xeon® Processor Scalable Family, 2nd and 3rd Generation Intel® Xeon® Processor Scalable Family, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DDEH	3530	MSR_LBR_INFO_30	Thread	Last Branch Record 30 Additional Information (R/W) See description of MSR_LBR_INFO_0.
DDFH	3531	MSR_LBR_INFO_31	Thread	Last Branch Record 31 Additional Information (R/W) See description of MSR_LBR_INFO_0.

Table 2-40 lists the MSRs of uncore PMU for Intel processors with CPUID DisplayFamily\_DisplayModel signatures of 06\_4EH, 06\_5EH, 06\_8EH, 06\_9EH, and 06\_66H.

**Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore Fixed Counter Control (R/W)
		19:0		Reserved
		20		Enable overflow propagation.
		21		Reserved
		22		Enable counting.
		63:23		Reserved
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore Fixed Counter
		43:0		Current count.
		63:44		Reserved
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box Configuration Information (R/O)
		3:0		Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics).
		63:4		Reserved
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb Unit, Performance Counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb Unit, Performance Counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb Unit, Counter 0 Event Select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb Unit, Counter 1 Event Select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSELO	Package	Uncore C-Box 0, Counter 0 Event Select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, Counter 1 Event Select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, Performance Counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, Performance Counter 1
710H	1808	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, Counter 0 Event Select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, Counter 1 Event Select MSR

**Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
716H	1814	MSR_UNC_CBO_1_PERFCTRO	Package	Uncore C-Box 1, Performance Counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, Performance Counter 1
720H	1824	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, Counter 0 Event Select MSR
721H	1825	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, Counter 1 Event Select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTRO	Package	Uncore C-Box 2, Performance Counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, Performance Counter 1
730H	1840	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, Counter 0 Event Select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, Counter 1 Event Select MSR
736H	1846	MSR_UNC_CBO_3_PERFCTRO	Package	Uncore C-Box 3, Performance Counter 0
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, Performance Counter 1
E01H	3585	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU Global Control
		0		Slice 0 select.
		1		Slice 1 select.
		2		Slice 2 select.
		3		Slice 3 select.
		4		Slice 4select.
		18:5		Reserved
		29		Enable all uncore counters.
		30		Enable wake on PMI.
		31		Enable Freezing counter when overflow.
		63:32		Reserved
E02H	3586	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU Main Status
		0		Fixed counter overflowed.
		1		An ARB counter overflowed.
		2		Reserved
		3		A CBox counter overflowed (on any slice).
		63:4		Reserved

## 2.17.1 MSRs Specific to 7th Generation and 8th Generation Intel® Core™ Processors based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Table 2-42 lists additional MSRs for 7th generation and 8th generation Intel Core processors with a CPUID DisplayFamily\_DisplayModel signatures of 06\_8EH and 06\_9EH. For an MSR listed in Table 2-42 that also appears in the model-specific tables of prior generations, Table 2-42 supersedes prior generation tables.

**Table 2-41. Additional MSRs Supported by 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
80H	128	MSR_TRACE_HUB_STH ACPIBAR_BASE	Package	NPK Address Used by AET Messages (R/W)
		0		Lock Bit If set, this MSR cannot be re-written anymore. Lock bit has to be set in order for the AET packets to be directed to NPK MMIO.
		17:1		Reserved
		63:18		ACPIBAR_BASE_ADDRESS AET target address in NPK MMIO space.
1F4H	500	MSR_PRMRR_PHYS_BASE	Core	Processor Reserved Memory Range Register - Physical Base Control Register (R/W)
		2:0		MemType PRMRR BASE MemType.
		11:3		Reserved
		45:12		Base PRMRR Base Address.
		63:46		Reserved
1F5H	501	MSR_PRMRR_PHYS_MASK	Core	Processor Reserved Memory Range Register - Physical Mask Control Register (R/W)
		9:0		Reserved
		10		Lock Lock bit for the PRMRR.
		11		VLD Enable bit for the PRMRR.
		45:12		Mask PRMRR MASK bits.
		63:46		Reserved
1FBH	507	MSR_PRMRR_VALID_CONFIG	Core	Valid PRMRR Configurations (R/W)
		0		1M supported MEE size.
		4:1		Reserved
		5		32M supported MEE size.
		6		64M supported MEE size.
		7		128M supported MEE size.
		31:8		Reserved



**Table 2-41. Additional MSRs Supported by 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
2F4H	756	MSR_UNCORE_PRMRR_PHYS_BASE	Package	(R/W) The PRMRR range is used to protect the processor reserved memory from unauthorized reads and writes. Any IO access to this range is aborted. This register controls the location of the PRMRR range by indicating its starting address. It functions in tandem with the PRMRR mask register.
		11:0		Reserved
		PAWIDTH-1:12		Range Base This field corresponds to bits PAWIDTH-1:12 of the base address memory range which is allocated to PRMRR memory.
		63:PAWIDTH		Reserved
2F5H	757	MSR_UNCORE_PRMRR_PHYS_MASK	Package	(R/W) This register controls the size of the PRMRR range by indicating which address bits must match the PRMRR base register value.
		9:0		Reserved
		10		Lock Setting this bit locks all writeable settings in this register, including itself.
		11		Range_En Indicates whether the PRMRR range is enabled and valid.
		38:12		Range_Mask This field indicates which address bits must match PRMRR base in order to qualify as an PRMRR access.
		63:39		Reserved
620H	1568	MSR_RING_RATIO_LIMIT	Package	Ring Ratio Limit (R/W) This register provides Min/Max Ratio Limits for the LLC and Ring.
		6:0		MAX_Ratio This field is used to limit the max ratio of the LLC/Ring.
		7		Reserved
		14:8		MIN_Ratio Writing to this field controls the minimum possible ratio of the LLC/Ring.
		63:15		Reserved

## 2.17.2 MSRs Specific to 8th Generation Intel® Core™ i3 Processors

Table 2-42 lists additional MSRs for 8th generation Intel Core i3 processors with a CPUID DisplayFamily\_DisplayModel signature of 06\_66H. For an MSR listed in Table 2-42 that also appears in the model-specific tables of prior generations, Table 2-42 supersedes prior generation tables.

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors  
Based on Cannon Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Enable VMX Inside SMX Operation (R/WL)
		2		Enable VMX Outside SMX Operation (R/WL)
		14:8		SENTER Local Functions Enables (R/WL)
		15		SENTER Global Functions Enable (R/WL)
		17		SGX Launch Control Enable (R/WL) This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Available only if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.
		18		SGX Global Functions Enable (R/WL)
		63:21		Reserved
350H	848	MSR_BR_DETECT_CTRL		Branch Monitoring Global Control (R/W)
		0		EnMonitoring Global enable for branch monitoring.
		1		EnExcept Enable branch monitoring event signaling on threshold trip. The branch monitoring event handler is signaled via the existing PMI signaling mechanism as programmed from the corresponding local APIC LVT entry.
		2		EnLBRFrz Enable LBR freeze on threshold trip. This will cause the LBR frozen bit 58 to be set in IA32_PERF_GLOBAL_STATUS when a triggering condition occurs and this bit is enabled.
		3		DisableInGuest When set to '1', branch monitoring, event triggering and LBR freeze actions are disabled when operating at VMX non-root operation.
		7:4		Reserved

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors  
Based on Cannon Lake Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17:8		WindowSize Window size defined by WindowCntSel. Values 0 - 1023 are supported. Once the Window counter reaches the WindowSize count both the Window Counter and all Branch Monitoring Counters are cleared.
		23:18		Reserved
		25:24		WindowCntSel Window event count select: '00 = Instructions retired. '01 = Branch instructions retired '10 = Return instructions retired. '11 = Indirect branch instructions retired.
		26		CntAndMode When set to '1', the overall branch monitoring event triggering condition is true only if all enabled counters' threshold conditions are true. When '0', the threshold tripping condition is true if any enabled counters' threshold is true.
		63:27		Reserved
351H	849	MSR_BR_DETECT_STATUS		Branch Monitoring Global Status (R/W)
		0		Branch Monitoring Event Signaled When set to '1', Branch Monitoring event signaling is blocked until this bit is cleared by software.
		1		LBRsValid This status bit is set to '1' if the LBR state is considered valid for sampling by branch monitoring software.
		7:2		Reserved
		8		CntrHit0 Branch monitoring counter #0 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.
		9		CntrHit1 Branch monitoring counter #1 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.
		15:10		Reserved Reserved for additional branch monitoring counters threshold hit status.

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors  
Based on Cannon Lake Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		25:16		CountWindow The current value of the window counter. The count value is frozen on a valid branch monitoring triggering condition. This is a 10-bit unsigned value.
		31:26		Reserved Reserved for future extension of CountWindow.
		39:32		Count0 The current value of counter 0 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit0 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).
		47:40		Count1 The current value of counter 1 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit1 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).
		63:48		Reserved
354H - 355H	852 - 853	MSR_BR_DETECT_COUNTER_CONFIG_i		Branch Monitoring Detect Counter Configuration (R/W)
		0		CntrEn Enable counter.
		7:1		CntrEvSel Event select (other values #GP) '0000000 = RETs. '0000001 = RET-CALL bias. '0000010 = RET mispredicts. '0000011 = Branch (all) mispredicts. '0000100 = Indirect branch mispredicts. '0000101 = Far branch instructions.

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors  
Based on Cannon Lake Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:8		CntrThreshold Threshold (an unsigned value of 0 to 127 supported). The value 0 of counter threshold will result in event signaled after every instruction. #GP if threshold is < 2.
		15		MispredEventCnt Mispredict events counting behavior: '0 = Mispredict events are counted in a window. '1 = Mispredict events are counted based on a consecutive occurrence. CntrThreshold is treated as # of consecutive mispredicts. This control bit only applies to events specified by CntrEvSel that involve a prediction (0000010, 0000011, 0000100). Setting this bit for other events is ignored.
		63:16		Reserved
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Package C3 Residency Counter (R/O)
		63:0		Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
620H	1568	MSR_RING_RATIO_LIMIT	Package	Ring Ratio Limit (R/W) This register provides Min/Max Ratio Limits for the LLC and Ring.
		6:0		MAX_Ratio This field is used to limit the max ratio of the LLC/Ring.
		7		Reserved
		14:8		MIN_Ratio Writing to this field controls the minimum possible ratio of the LLC/Ring.
		63:15		Reserved
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Core C1 Residency Counter (R/O)
		63:0		Value since last reset for the Core C1 residency. Counter rate is the Max Non-Turbo frequency (same as TSC). This counter counts in case both of the core's threads are in an idle state and at least one of the core's thread residency is in a C1 state or in one of its sub states. The counter is updated only after a core C state exit. Note: Always reads 0 if core C1 is unsupported. A value of zero indicates that this processor does not support core C1 or never entered core C1 level state.
662H	1634	MSR_CORE_C3_RESIDENCY	Core	Core C3 Residency Counter (R/O)
		63:0		Will always return 0.

Table 2-43 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06\_66H.

**Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore Fixed Counter Control (R/W)
		19:0		Reserved
		20		Enable overflow propagation.
		21		Reserved
		22		Enable counting.
		63:23		Reserved
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore Fixed Counter
		47:0		Current count.
		63:48		Reserved
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box Configuration Information (R/O)
		3:0		Report the number of C-Box units with performance counters, including processor cores and processor graphics.
		63:4		Reserved
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb Unit, Performance Counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb Unit, Performance Counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb Unit, Counter 0 Event Select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, Counter 1 Event Select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSELO	Package	Uncore C-Box 0, Counter 0 Event Select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, Counter 1 Event Select MSR
702H	1794	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, Performance Counter 0
703H	1795	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, Performance Counter 1
708H	1800	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, Counter 0 Event Select MSR
709H	1801	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, Counter 1 Event Select MSR
70AH	1802	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, Performance Counter 0
70BH	1803	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, Performance Counter 1
710H	1808	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, Counter 0 Event Select MSR
711H	1809	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, Counter 1 Event Select MSR
712H	1810	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, Performance Counter 0
713H	1811	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, Performance Counter 1
718H	1816	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, Counter 0 Event Select MSR
719H	1817	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, Counter 1 Event Select MSR
71AH	1818	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, Performance Counter 0
71BH	1819	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, Performance Counter 1

**Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
720H	1824	MSR_UNC_CBO_4_PERFEVTSELO	Package	Uncore C-Box 4, Counter 0 Event Select MSR
721H	1825	MSR_UNC_CBO_4_PERFEVTSEL1	Package	Uncore C-Box 4, Counter 1 Event Select MSR
722H	1826	MSR_UNC_CBO_4_PERFCTRO	Package	Uncore C-Box 4, Performance Counter 0
723H	1827	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, Performance Counter 1
728H	1832	MSR_UNC_CBO_5_PERFEVTSELO	Package	Uncore C-Box 5, Counter 0 Event Select MSR
729H	1833	MSR_UNC_CBO_5_PERFEVTSEL1	Package	Uncore C-Box 5, Counter 1 Event Select MSR
72AH	1834	MSR_UNC_CBO_5_PERFCTRO	Package	Uncore C-Box 5, Performance Counter 0
72BH	1835	MSR_UNC_CBO_5_PERFCTR1	Package	Uncore C-Box 5, Performance Counter 1
730H	1840	MSR_UNC_CBO_6_PERFEVTSELO	Package	Uncore C-Box 6, Counter 0 Event Select MSR
731H	1841	MSR_UNC_CBO_6_PERFEVTSEL1	Package	Uncore C-Box 6, Counter 1 Event Select MSR
732H	1842	MSR_UNC_CBO_6_PERFCTRO	Package	Uncore C-Box 6, Performance Counter 0
733H	1843	MSR_UNC_CBO_6_PERFCTR1	Package	Uncore C-Box 6, Performance Counter 1
738H	1848	MSR_UNC_CBO_7_PERFEVTSELO	Package	Uncore C-Box 7, Counter 0 Event Select MSR
739H	1849	MSR_UNC_CBO_7_PERFEVTSEL1	Package	Uncore C-Box 7, Counter 1 Event Select MSR
73AH	1850	MSR_UNC_CBO_7_PERFCTRO	Package	Uncore C-Box 7, Performance Counter 0
73BH	1851	MSR_UNC_CBO_7_PERFCTR1	Package	Uncore C-Box 7, Performance Counter 1
E01H	3585	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU Global Control
		0		Slice 0 select.
		1		Slice 1 select.
		2		Slice 2 select.
		3		Slice 3 select.
		4		Slice 4select.
		18:5		Reserved
		29		Enable all uncore counters.
		30		Enable wake on PMI.
		31		Enable Freezing counter when overflow.
63:32		Reserved		
E02H	3586	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU Main Status
		0		Fixed counter overflowed.
		1		An ARB counter overflowed.
		2		Reserved
		3		A CBox counter overflowed (on any slice).
		63:4		Reserved

## 2.17.3 MSRs Specific to 10th Generation Intel® Core™ Processors

Table 2-44 lists additional MSRs for 10th generation Intel Core processors with a CPUID DisplayFamily\_DisplayModel signature values of 06\_7DH and 06\_7EH. For an MSR listed in Table 2-44 that also appears in the model-specific tables of prior generations, Table 2-44 supersedes prior generation tables.

**Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
33H	51	MSR_TEST_CTRL	Core	Test Control Register
		28:0		Reserved.
		29		Enable #AC(0) exception for split locked accesses: Cause #AC(0) exception for split locked access at all CPL irrespective of CRO.AM or EFLAGS.AC. If bits 29 and 31 are both set, bit 29 takes precedence.
		30		Reserved.
		31		Reserved.
48H	72	IA32_SPEC_CTRL	Core	See Table 2-2.
49H	73	IA32_PREDICT_CMD	Thread	See Table 2-2.
8CH	140	IA32_SGXLEPUBKEYHASH0	Thread	See Table 2-2.
8DH	141	IA32_SGXLEPUBKEYHASH1	Thread	See Table 2-2.
8EH	142	IA32_SGXLEPUBKEYHASH2	Thread	See Table 2-2.
8FH	143	IA32_SGXLEPUBKEYHASH3	Thread	See Table 2-2.
A0H	160	MSR_BIOS_MCU_ERRORCODE	Package	BIOS MCU ERRORCODE (R/O) This MSR indicates if WRMSR 0x79 failed to configure PRM memory and gives a hint to debug BIOS.
		15:0	Package	Error Codes (R/O)
		30:16		Reserved.
		31	Thread	MCU Partial Success (R/O) When set to 1, WRMSR 0x79 skipped part of the functionality during BIOS.
A5H	165	MSR_FIT_BIOS_ERROR	Thread	FIT BIOS ERROR (R/W) Report error codes for debug in case the processor failed to parse the Firmware Table in BIOS. Can also be used to log BIOS information.
		7:0		Error Codes (R/W) Error codes for debug.
		15:8		Entry Type (R/W) Failed FIT entry type.
		16		FIT MCU Entry (R/W) FIT contains MCU entry.
		62:17		Reserved.
		63		LOCK (R/W) When set to 1, writes to this MSR will be skipped.



**Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
10BH	267	IA32_FLUSH_CMD	Thread	See Table 2-2.
151H	337	MSR_BIOS_DONE	Thread	BIOS Done (R/WO)
		0	Thread	BIOS Done Indication (R/WO) Set by BIOS when it finishes programming the processor and wants to lock the memory configuration from changes by software that is running on this thread. Writes to the bit will be ignored if EAX[0] is 0.
		1	Package	Package BIOS Done Indication (R/O) When set to 1, all threads in the package have bit 0 of this MSR set.
		31:2		Reserved.
1F1H	497	MSR_CRASHLOG_CONTROL	Thread	Write Data to a Crash Log Configuration
		0		CDDIS: CrashDump_Disable If set, indicates that Crash Dump is disabled.
		63:1		Reserved.
2A0H	672	MSR_PRMRR_BASE_0	Core	Processor Reserved Memory Range Register - Physical Base Control Register (R/W)
		2:0		MEMTYPE: PRMRR BASE Memory Type.
		3		CONFIGURED: PRMRR BASE Configured.
		11:4		Reserved.
		51:12		BASE: PRMRR Base Address.
		63:52		Reserved.
30CH	780	IA32_FIXED_CTR3	Thread	Fixed-Function Performance Counter Register 3 (R/W) Bit definitions are the same as found in IA32_FIXED_CTR0, offset 309H. See Table 2-2.
329H	809	MSR_PERF_METRICS	Thread	Performance Metrics (R/W) Reports metrics directly. Software can check (and/or expose to its guests) the availability of PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15).
		7:0		Retiring. Percent of utilized slots by uops that eventually retire (commit).
		15:8		Bad Speculation. Percent of wasted slots due to incorrect speculation, covering utilized by uops that do not retire, or recovery bubbles (unutilized slots).
		23:16		Frontend Bound. Percent of unutilized slots where front-end did not deliver a uop while back-end is ready.
		31:24		Backend Bound. Percent of unutilized slots where a uop was not delivered to back-end due to lack of back-end resources.

**Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:25		Reserved.
3F2H	1010	MSR_PEBS_DATA_CFG	Thread	PEBS Data Configuration (R/W) Provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.
		0		Memory Info. Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.
		1		GPRs. Setting this bit will capture the contents of the General Purpose registers in the PEBS record.
		2		XMMs. Setting this bit will capture the contents of the XMM registers in the PEBS record.
		3		LBRs. Setting this bit will capture LBR TO, FROM and INFO in the PEBS record.
		23:4		Reserved.
		31:24		LBR Entries. Set the field to the desired number of entries - 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, software can use LBR_entries that is multiple of 3.
541H	1345	MSR_CORE_UARCH_CTL	Core	Core Microarchitecture Control MSR (R/W)
		0		L1 Scrubbing Enable When set to 1, enable L1 scrubbing.
		31:1		Reserved.
657H	1623	MSR_FAST_UNCORE_MSRS_CTL	Thread	Fast WRMSR/RDMSR Control MSR (R/W)
		3:0		FAST_ACCESS_ENABLE: Bit 0: When set to '1', provides a hint for the hardware to enable fast access mode for the IA32_HWP_REQUEST MSR. This bit is sticky and is cleaned by the hardware only during reset time. This bit is valid only if FAST_UNCORE_MSRS_CAPABILITY[0] is set. Setting this bit will cause CPUID[6].EAX[18] to be set.
		31:4		Reserved.

**Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
65EH	1630	MSR_FAST_UNCORE_MSRS_STATUS	Thread	Indication of Uncore MSRs, Post Write Activates
		0		Indicates whether the CPU is still in the middle of writing IA32_HWP_REQUEST MSR, even after the WRMSR instruction has retired. A value of 1 indicates the last write of IA32_HWP_REQUEST is still ongoing. A value of 0 indicates the last write of IA32_HWP_REQUEST is visible outside the logical processor. Software can use the status of this bit to avoid overwriting IA32_HWP_REQUEST.
		31:1		Reserved.
65FH	1631	MSR_FAST_UNCORE_MSRS_CAPABILITY	Thread	Fast WRMSR/RDMSR Enumeration MSR (RO)
		3:0		MSRS_CAPABILITY: Bit 0: If set to '1', hardware supports the fast access mode for the IA32_HWP_REQUEST MSR.
		31:4		Reserved.
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Table 2-2.
775H	1909	IA32_PECI_HWP_REQUEST_INFO	Thread	See Table 2-2.
777H	1911	IA32_HWP_STATUS	Thread	See Table 2-2.

### 2.17.4 MSRs Specific to 11th Generation Intel® Core™ Processors based on Tiger Lake Microarchitecture

Table 2-45 lists additional MSRs for 11th generation Intel Core processors with CPUID DisplayFamily\_DisplayModel signatures of 06\_8CH and 06\_8DH. For an MSR listed in Table 2-45 that also appears in the model-specific tables of prior generations, Table 2-45 supersedes prior generation tables.

**Table 2-45. Additional MSRs Supported by 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
A0H	160	MSR_BIOS_MCU_ERRORCODE	Package	BIOS MCU ERRORCODE (R/O)
		15:0		Error Codes
		31:16		Reserved
A7H	167	MSR_BIOS_DEBUG	Thread	BIOS DEBUG (R/O) This MSR indicates if WRMSR 79H failed to configure PRM memory and gives a hint to debug BIOS.
		30:0		Reserved
		31		MCU Partial Success When set to 1, WRMSR 79H skipped part of the functionality during BIOS.

**Table 2-45. Additional MSRs Supported by 11th Generation Intel® Core™ Processors  
Based on Tiger Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:32		Reserved
CFH	207	IA32_CORE_CAPABILITIES	Package	IA32_CR_CORE_CAPABILITIES (R/O) This MSR provides an architectural enumeration function for model-specific behavior.
		0		STLB_QOS_SUPPORTED When set to 1, the STLB QoS feature is supported and the STLB QoS MSRs (1A8FH -1A97H) are accessible. When set to 0, access to these MSRs will #GP.
		1		Reserved
		2		FUSA_SUPPORTED
		3		RSM_IN_CPL0_ONLY When set to 1, the RSM instruction is only allowed in CPL0 (#GP triggered in any CPL != 0). When set to 0, then any CPL may execute the RSM instruction.
		4		Reserved
		5		SPLIT_LOCK_DISABLE_SUPPORTED When set to 1, the ability to set MEMORY_CONTROL (MSR 33H) bit 29 enables an #AC to be created when a split lock is detected.
		6		SNOOP_FILTER_QOS_SUPPORTED When set to 1, the Snoop Filter Qos Mask MSRs are supported. When set to 0, access to these MSRs will #GP.
		31:7		Reserved
492H	1170	IA32_VMX_PROCBASED_CTL3	Core	IA32_VMX_PROCBASED_CTL3 This MSR enumerates the allowed 1-settings of the third set of processor-based controls. Specifically, VM entry allows bit X of the tertiary processor-based VM-execution controls to be 1 if and only if bit X of the MSR is set to 1.  If bit X of the MSR is cleared to 0, VM entry fails if control X and the “activate tertiary controls” primary processor-based VM-execution control are both 1.
		0		LOADIWKEY This control determines whether executions of LOADIWKEY cause VM exits.
		63:1		Reserved
601H	1537	MSR_VR_CURRENT_CONFIG	Package	Power Limit 4 (PL4) Package-level maximum power limit (in Watts). It is a proactive, instantaneous limit.

**Table 2-45. Additional MSRs Supported by 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12:0		PL4 Value PL4 value in 0.125 A increments. This field is locked by VR_CURRENT_CONFIG[LOCK]. When the LOCK bit is set to 1b, this field becomes Read Only.
		30:13		Reserved
		31		Lock Indication (LOCK) This bit will lock the CURRENT_LIMIT settings in this register and will also lock this setting. This means that once set to 1b, the CURRENT_LIMIT setting and this bit become Read Only until the next Warm Reset.
		62:32		Not in use.
		63		Reserved
981H	2433	IA32_TME_CAPABILITY		See Table 2-2.
982H	2434	IA32_TME_ACTIVATE		See Table 2-2.
983H	2435	IA32_TME_EXCLUDE_MASK		See Table 2-2.
984H	2436	IA32_TME_EXCLUDE_BASE		See Table 2-2.
990H	2448	IA32_COPY_STATUS <sup>1</sup>	Thread	See Table 2-2.
991H	2449	IA32_IWKEYBACKUP_STATUS <sup>1</sup>	Platform	See Table 2-2.
C82H	3202	IA32_L2_QOS_CFG	Core	IA32_CR_L2_QOS_CFG This MSR provides software an enumeration of the parameters that L2 QoS (Intel RDT) support in any particular implementation.
		0		CDP_ENABLE When set to 1, it will enable the code and data prioritization for the L2 CAT/Intel RDT feature. When set to 0, code and data prioritization is disabled for L2 CAT/Intel RDT. See Chapter 17, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features” for further details on CDP.
		31:1		Reserved
D10H - D17H	3220 - 3351	IA32_L2_QOS_MASK_[0-7]	Package	IA32_CR_L2_QOS_MASK_[0-7] Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 17, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features”.

**Table 2-45. Additional MSRs Supported by 11th Generation Intel® Core™ Processors  
Based on Tiger Lake Microarchitecture**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		19:0		<p>WAYS_MASK</p> <p>Setting a 1 in this bit X allows threads with CLOS &lt;n&gt; (where N is [0-7]) to allocate to way X in the MLC. Ones are only allowed to be written to ways that physically exist in the MLC (CPUID.4.2:EBX[31:22] will indicate this).</p> <p>Writing a 1 to a value beyond the highest way or a non-contiguous set of 1s will cause a #GP on the WRMSR to this MSR.</p>
		31:20		Reserved
D91H	3473	IA32_COPY_LOCAL_TO_PLATFORM <sup>1</sup>	Thread	See Table 2-2.
D92H	3474	IA32_COPY_PLATFORM_TO_LOCAL <sup>1</sup>	Thread	See Table 2-2.

**NOTES:**

1. Further details on Key Locker and usage of this MSR can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

## 2.17.5 MSRs Specific to Intel® Xeon® Processor Scalable Family

Intel® Xeon® Processor Scalable Family (CPUID DisplayFamily\_DisplayModel = 06\_55H) support the MSRs listed in Table 2-46.

**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Enable VMX Inside SMX Operation (R/WL)
		2		Enable VMX Outside SMX Operation (R/WL)
		14:8		SENTER Local Functions Enables (R/WL)
		15		SENTER Global Functions Enable (R/WL)
		18		SGX Global Functions Enable (R/WL)
		20		LMCE_ENABLED (R/WL)
		63:21		Reserved
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) See Table 2-26.

**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		Enable_PPIN (R/W) See Table 2-26.
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		Protected Processor Inventory Number (R/O) See Table 2-26.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) See Table 2-26.
		22:16		Reserved.
		23	Package	PPIN_CAP (R/O) See Table 2-26.
		27:24		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) See Table 2-26.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) See Table 2-26.
		30	Package	Programmable TJ OFFSET (R/O) See Table 2-26.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) See Table 2-26.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .

Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2:0		Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		I/O MWait Redirection Enable (R/W)
		14:11		Reserved
		15		CFG Lock (R/W0)
		16		Automatic C-State Conversion Enable (R/W) If 1, the processor will convert HALT or MWait(C1) to MWait(C6).
		24:17		Reserved
		25		C3 State Auto Demotion Enable (R/W)
		26		C1 State Auto Demotion Enable (R/W)
		27		Enable C3 Undemotion (R/W)
		28		Enable C1 Undemotion (R/W)
		29		Package C State Demotion Enable (R/W)
		30		Package C State Undemotion Enable (R/W)
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	Global Machine Check Capability (R/O)
		7:0		Count
		8		MCG_CTL_P
		9		MCG_EXT_P
		10		MCP_CMCI_P
		11		MCG_TES_P
		15:12		Reserved
		23:16		MCG_EXT_CNT
		24		MCG_SER_P
		25		MCG_EM_P
		26		MCG_ELOG_P
		63:27		Reserved



**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
17DH	381	MSR_SMM_MCA_CAP	THREAD	Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		Reserved
		58		SMM_Code_Access_Chk (SMM-RO) If set to 1 indicates that the SMM code access restriction is supported and a host-space interface is available to SMM handler.
		59		Long_Flow_Indication (SMM-RO) If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface is available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
		0		Thermal Status (RO) See Table 2-2.
		1		Thermal Status Log (R/WCO) See Table 2-2.
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.
		6		Thermal Threshold #1 Status (RO) See Table 2-2.
		7		Thermal Threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
11		Power Limitation Log (R/WCO) See Table 2-2.		

Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12		Current Limit Status (RO) See Table 2-2.
		13		Current Limit Log (R/WCO) See Table 2-2.
		14		Cross Domain Limit Status (RO) See Table 2-2.
		15		Cross Domain Limit Log (R/WCO) See Table 2-2.
		22:16		Digital Readout (RO) See Table 2-2.
		26:23		Reserved
		30:27		Resolution in Degrees Celsius (RO) See Table 2-2.
		31		Reading Valid (RO) See Table 2-2.
		63:32		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (RO) See Table 2-26.
		27:24		TCC Activation Offset (R/W) See Table 2-26.
		63:28		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	This register defines the ratio limits. RATIO[0:7] must be populated in ascending order. RATIO[i+1] must be less than or equal to RATIO[i]. Entries with RATIO[i] will be ignored. If any of the rules above are broken, the configuration is silently rejected. If the programmed ratio is: <ul style="list-style-type: none"> <li>▪ Above the fused ratio for that core count, it will be clipped to the fuse limits (assuming !IOC).</li> <li>▪ Below the min supported ratio, it will be clipped.</li> </ul>
		7:0		RATIO_0 Defines ratio limits.
		15:8		RATIO_1 Defines ratio limits.
		23:16		RATIO_2 Defines ratio limits.
		31:24		RATIO_3 Defines ratio limits.

**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		39:32		RATIO_4 Defines ratio limits.
		47:40		RATIO_5 Defines ratio limits.
		55:48		RATIO_6 Defines ratio limits.
		63:56		RATIO_7 Defines ratio limits.
1AEH	430	MSR_TURBO_RATIO_LIMIT_CORES	Package	This register defines the active core ranges for each frequency point. NUMCORE[0:7] must be populated in ascending order. NUMCORE[i+1] must be greater than NUMCORE[i]. Entries with NUMCORE[i] == 0 will be ignored. The last valid entry must have NUMCORE >= the number of cores in the SKU. If any of the rules above are broken, the configuration is silently rejected.
		7:0		NUMCORE_0 Defines the active core ranges for each frequency point.
		15:8		NUMCORE_1 Defines the active core ranges for each frequency point.
		23:16		NUMCORE_2 Defines the active core ranges for each frequency point.
		31:24		NUMCORE_3 Defines the active core ranges for each frequency point.
		39:32		NUMCORE_4 Defines the active core ranges for each frequency point.
		47:40		NUMCORE_5 Defines the active core ranges for each frequency point.
		55:48		NUMCORE_6 Defines the active core ranges for each frequency point.
		63:56		NUMCORE_7 Defines the active core ranges for each frequency point.
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.

Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
284H	644	IA32_MC4_CTL2	Package	See Table 2-2.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
400H	1024	IA32_MCO_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MCO reports MC errors from the IFU module.
401H	1025	IA32_MCO_STATUS	Core	
402H	1026	IA32_MCO_ADDR	Core	
403H	1027	IA32_MCO_MISC	Core	
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MC1 reports MC errors from the DCU module.
405H	1029	IA32_MC1_STATUS	Core	
406H	1030	IA32_MC1_ADDR	Core	
407H	1031	IA32_MC1_MISC	Core	
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MC2 reports MC errors from the DTLB module.
409H	1033	IA32_MC2_STATUS	Core	
40AH	1034	IA32_MC2_ADDR	Core	
40BH	1035	IA32_MC2_MISC	Core	
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MC3 reports MC errors from the MLC module.
40DH	1037	IA32_MC3_STATUS	Core	
40EH	1038	IA32_MC3_ADDR	Core	
40FH	1039	IA32_MC3_MISC	Core	
410H	1040	IA32_MC4_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MC4 reports MC errors from the PCU module.
411H	1041	IA32_MC4_STATUS	Package	
412H	1042	IA32_MC4_ADDR	Package	
413H	1043	IA32_MC4_MISC	Package	

**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC errors from a link interconnect module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC errors from the M2M 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC8 reports MC errors from the M2M 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 - MC11 report MC errors from the CHA
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 - MC11 report MC errors from the CHA.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 - MC11 report MC errors from the CHA.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC12 report MC errors from each channel of a link interconnect module.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC errors from the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	

Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC errors from the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC errors from the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC errors from the integrated memory controllers
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC errors from the integrated memory controllers.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC errors from the integrated memory controllers.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC errors from a link interconnect module.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers Used in RAPL Interfaces (R/O)
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
		15:13	Package	Reserved
		19:16	Package	Time Units See Section 14.10.1, "RAPL Interfaces."
		63:20		Reserved

**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) Energy consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	Reserved (R/O) Reads return 0.
C8DH	3213	IA32_QM_EVTSEL	THREAD	Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.
		7:0		EventID (RW) Event encoding: 0x00: No monitoring. 0x01: L3 occupancy monitoring. 0x02: Total memory bandwidth monitoring. 0x03: Local memory bandwidth monitoring. All other encoding reserved.
		31:8		Reserved
		41:32		RMID (RW)
		63:42		Reserved
C8FH	3215	IA32_PQR_ASSOC	THREAD	Resource Association Register (R/W)
		9:0		RMID

Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		31:10		Reserved
		51:32		COS (R/W)
		63: 52		Reserved
C90H	3216	IA32_L3_QOS_MASK_0	Package	L3 Class Of Service Mask - COS 0 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0.
		0:19		CBM: Bit vector of available L3 ways for COS 0 enforcement.
		63:20		Reserved
C91H	3217	IA32_L3_QOS_MASK_1	Package	L3 Class Of Service Mask - COS 1 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1.
		0:19		CBM: Bit vector of available L3 ways for COS 1 enforcement.
		63:20		Reserved
C92H	3218	IA32_L3_QOS_MASK_2	Package	L3 Class Of Service Mask - COS 2 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2.
		0:19		CBM: Bit vector of available L3 ways for COS 2 enforcement.
		63:20		Reserved
C93H	3219	IA32_L3_QOS_MASK_3	Package	L3 Class Of Service Mask - COS 3 (R/W). If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3.
		0:19		CBM: Bit vector of available L3 ways for COS 3 enforcement.
		63:20		Reserved
C94H	3220	IA32_L3_QOS_MASK_4	Package	L3 Class Of Service Mask - COS 4 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4.
		0:19		CBM: Bit vector of available L3 ways for COS 4 enforcement.
		63:20		Reserved
C95H	3221	IA32_L3_QOS_MASK_5	Package	L3 Class Of Service Mask - COS 5 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5.
		0:19		CBM: Bit vector of available L3 ways for COS 5 enforcement.
		63:20		Reserved
C96H	3222	IA32_L3_QOS_MASK_6	Package	L3 Class Of Service Mask - COS 6 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6.
		0:19		CBM: Bit vector of available L3 ways for COS 6 enforcement.
		63:20		Reserved
C97H	3223	IA32_L3_QOS_MASK_7	Package	L3 Class Of Service Mask - COS 7 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7.



**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0:19		CBM: Bit vector of available L3 ways for COS 7 enforcement.
		63:20		Reserved
C98H	3224	IA32_L3_QOS_MASK_8	Package	L3 Class Of Service Mask - COS 8 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=8.
		0:19		CBM: Bit vector of available L3 ways for COS 8 enforcement.
		63:20		Reserved
C99H	3225	IA32_L3_QOS_MASK_9	Package	L3 Class Of Service Mask - COS 9 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=9.
		0:19		CBM: Bit vector of available L3 ways for COS 9 enforcement.
		63:20		Reserved
C9AH	3226	IA32_L3_QOS_MASK_10	Package	L3 Class Of Service Mask - COS 10 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=10.
		0:19		CBM: Bit vector of available L3 ways for COS 10 enforcement.
		63:20		Reserved
C9BH	3227	IA32_L3_QOS_MASK_11	Package	L3 Class Of Service Mask - COS 11 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=11.
		0:19		CBM: Bit vector of available L3 ways for COS 11 enforcement.
		63:20		Reserved
C9CH	3228	IA32_L3_QOS_MASK_12	Package	L3 Class Of Service Mask - COS 12 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=12.
		0:19		CBM: Bit vector of available L3 ways for COS 12 enforcement.
		63:20		Reserved
C9DH	3229	IA32_L3_QOS_MASK_13	Package	L3 Class Of Service Mask - COS 13 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=13.
		0:19		CBM: Bit vector of available L3 ways for COS 13 enforcement.
		63:20		Reserved
C9EH	3230	IA32_L3_QOS_MASK_14	Package	L3 Class Of Service Mask - COS 14 (R/W) If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] >=14.
		0:19		CBM: Bit vector of available L3 ways for COS 14 enforcement.

**Table 2-46. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:20		Reserved
C9FH	3231	IA32_L3_QOS_MASK_15	Package	L3 Class Of Service Mask - COS 15 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >= 15.
		0:19		CBM: Bit vector of available L3 ways for COS 15 enforcement.
		63:20		Reserved

## 2.17.6 MSRs Specific to 3rd Generation Intel® Xeon® Processor Scalable Family based on Ice Lake Microarchitecture

The 3rd generation Intel® Xeon® Processor Scalable Family based on Ice Lake microarchitecture (CPUID DisplayFamily\_DisplayModel signatures of 06\_6AH and 06\_6CH) support the MSRs listed in Table 2-47.

**Table 2-47. MSRs Supported by 3rd Generation Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel Signatures of 06\_6AH and 06\_6CH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
612H	1554	MSR_PACKAGE_ENERGY_TIME_STATUS	Package	Package energy consumed by the entire CPU (R/W)
		31:0		Total amount of energy consumed since last reset.
		63:32		Total time elapsed when the energy was last updated. This is a monotonic increment counter with auto wrap back to zero after overflow. Unit is 10ns.
618H	1560	MSR_DRAM_POWER_LIMIT	Package	Allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.
		14:0		DRAM_PP_PWR_LIM: Power Limit[0] for DDR domain. Units = Watts, Format = 11.3, Resolution = 0.125W, Range = 0-2047.875W.
		15		PWR_LIM_CTRL_EN: Power Limit[0] enable bit for DDR domain.
		16		Reserved
		23:17		CTRL_TIME_WIN: Power Limit[0] time window Y value, for DDR domain. Actual time_window for RAPL is: (1/1024 seconds) * (1+(x/4)) * (2^y)
		62:24		Reserved
		63		PP_PWR_LIM_LOCK: When set, this entire register becomes read-only. This bit will typically be set by BIOS during boot.

**Table 2-47. MSRs Supported by 3rd Generation Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel Signatures of 06\_6AH and 06\_6CH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM Power Parameters (R/W)
		14:0		Spec DRAM Power (DRAM_TDP): The Spec power allowed for DRAM. The TDP setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].
		15		Reserved
		30:16		Minimal DRAM Power (DRAM_MIN_PWR): The minimal power setting allowed for DRAM. Lower values will be clamped to this value. The minimum setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].
		31		Reserved
		46:32		Maximal Package Power (DRAM_MAX_PWR): The maximal power setting allowed for DRAM. Higher values will be clamped to this value. The maximum setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].
		47		Reserved
		54:48		Maximal Time Window (DRAM_MAX_WIN): The maximal time window allowed for the DRAM. Higher values will be clamped to this value. x = PKG_MAX_WIN[54:53] y = PKG_MAX_WIN[52:48] The timing interval window is Floating Point number given by 1.x *power(2,y). The unit of measurement is defined in MSR_DRAM_POWER_INFO_UNIT[TIME_UNIT].
62:55		Reserved		

**Table 2-47. MSRs Supported by 3rd Generation Intel® Xeon® Processor Scalable Family with DisplayFamily\_DisplayModel Signatures of 06\_6AH and 06\_6CH**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63		LOCK: Lock bit to lock the register.
981H	2433	IA32_TME_CAPABILITY		See Table 2-2.
982H	2434	IA32_TME_ACTIVATE		See Table 2-2.
983H	2435	IA32_TME_EXCLUDE_MASK		See Table 2-2.
984H	2436	IA32_TME_EXCLUDE_BASE		See Table 2-2.

## 2.18 MSRS IN INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES AND INTEL® XEON PHI™ PROCESSOR 7215/7285/7295 SERIES

Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with CPUID DisplayFamily\_DisplayModel signature 06\_57H, supports the MSR interfaces listed in Table 2-48. These processors are based on the Knights Landing microarchitecture. Intel® Xeon Phi™ processor 7215, 7285, 7295 series, with CPUID DisplayFamily\_DisplayModel signature 06\_85H, supports the MSR interfaces listed in Table 2-48 and Table 2-49. These processors are based on the Knights Mill microarchitecture. Some MSRs are shared between a pair of processor cores, the scope is marked as module.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.23, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination." See Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	Platform ID (R) See Table 2-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	SMI Counter (R/O)
		31:0		SMI Count (R/O)
		63:32		Reserved
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64Processor (R/W) See Table 2-2.
		0		Lock (R/WL)
		1		Reserved
		2		Enable VMX outside SMX operation (R/WL)

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3BH	59	IA32_TSC_ADJUST	THREAD	Per-Logical-Processor TSC ADJUST (R/W) See Table 2-2.
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/W/O) See Table 2-26.
		1		Enable_PPIN (R/W) See Table 2-26.
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		Protected Processor Inventory Number (R/O) A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'.
79H	121	IA32_BIOS_UPDT_TRIG	Core	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	THREAD	BIOS Update Signature ID (RO) See Table 2-2.
C1H	193	IA32_PMC0	THREAD	Performance Counter Register See Table 2-2.
C2H	194	IA32_PMC1	THREAD	Performance Counter Register See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		39:30		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Package	C-State Configuration Control (R/W)
		2:0		Package C-State Limit (R/W) Specifies the lowest C-state for the package. This feature does not limit the processor core C-state. The power-on default value from bit[2:0] of this register reports the deepest package C-state the processor is capable to support when manufactured. It is recommended that BIOS always read the power-on default value reported from this bit field to determine the supported deepest C-state on the processor and leave it as default without changing it. 000b - C0/C1 (No package C-state support) 001b - C2 010b - C6 (non retention)* 011b - C6 (Retention)* 100b - Reserved 101b - Reserved 110b - Reserved 111b - No package C-state limit. All C-States supported by the processor are available. Note: C6 retention mode provides more power saving than C6 non-retention mode. Limiting the package to C6 non retention mode does prevent the MSR_PKG_C6_RESIDENCY counter (MSR 3F9h) from being incremented.
		9:3		Reserved
		10		I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO registers at MSR_PMG_IO_CAPTURE_BASE[15:0] to MWAIT instructions.
		14:11		Reserved
		15		CFG Lock (RO) When set, locks bits [15:0] of this register for further writes until the next reset occurs.
		25		Reserved
		26		C1 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		27		Reserved
		28		C1 State Auto Undemotion Enable (R/W) When set, enables Undemotion from Demoted C1.
		29		PKG C-State Auto Demotion Enable (R/W) When set, enables Package C state demotion.
		63:30		Reserved
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Tile	Power Management IO Capture Base (R/W)
		15:0		LVL_2 Base Address (R/W) Microcode will compare IO-read zone to this base address to determine if an MWAIT(C2/3/4) needs to be issued instead of the IO-read. Should be programmed to the chipset Plevel_2 IO address.
		22:16		C-State Range (R/W) The IO-port block size in which IO-redirection will be executed (0-127). Should be programmed based on the number of LVLx registers existing in the chipset.
		63:23		Reserved
E7H	231	IA32_MPERF	Thread	Maximum Performance Frequency Clock Count (RW) See Table 2-2.
E8H	232	IA32_APERF	Thread	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	Memory Type Range Register (R) See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, the AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved
140H	320	MISC_FEATURE_ENABLES	Thread	MISC_FEATURE_ENABLES
		0		Reserved

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		1		User Mode MONITOR and MWAIT (R/W) If set to 1, the MONITOR and MWAIT instructions do not cause invalid-opcode exceptions when executed with CPL > 0 or in virtual-8086 mode. If MWAIT is executed when CPL > 0 or in virtual-8086 mode, and if EAX indicates a C-state other than C0 or C1, the instruction operates as if EAX indicated the C-state C1.
		63:2		Reserved
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	See Table 2-2.
17DH	381	MSR_SMM_MCA_CAP	Thread	Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.
		31:0		Bank Support (SMM-RO) One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA).
		55:32		Reserved
		56		Targeted SMI (SMM-RO) Set if targeted SMI is supported.
		57		SMM_CPU_SVRSTR (SMM-RO) Set if SMM SRAM save/restore feature is supported.
		58		SMM_CODE_ACCESS_CHK (SMM-RO) Set if SMM code access check feature is supported.
		59		Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
186H	390	IA32_PERFEVTSELO	Thread	Performance Monitoring Event Select Register (R/W) See Table 2-2.
		7:0		Event Select
		15:8		UMask
		16		USR
		17		OS
		18		Edge
		19		PC



**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		20		INT
		21		AnyThread
		22		EN
		23		INV
		31:24		CMASK
		63:32		Reserved
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	Clock Modulation (R/W) See Table 2-2.
19BH	411	IA32_THERM_INTERRUPT	Module	Thermal Interrupt Control (R/W) See Table 2-2.
19CH	412	IA32_THERM_STATUS	Module	Thermal Monitor Status (R/W) See Table 2-2.
		0		Thermal Status (RO)
		1		Thermal Status Log (R/WCO)
		2		PROTCHOT # or FORCEPR# Status (RO)
		3		PROTCHOT # or FORCEPR# Log (R/WCO)
		4		Critical Temperature Status (RO)
		5		Critical Temperature Status Log (R/WCO)
		6		Thermal Threshold #1 Status (RO)
		7		Thermal Threshold #1 Log (R/WCO)
		8		Thermal Threshold #2 Status (RO)
		9		Thermal Threshold #2 Log (R/WCO)
		10		Power Limitation Status (RO)
		11		Power Limitation Log (R/WCO)
		15:12		Reserved
		22:16		Digital Readout (RO)
		26:23		Reserved
		30:27		Resolution in Degrees Celsius (RO)
		31		Reading Valid (RO)
		63:32		Reserved
1A0H	416	IA32_MISC_ENABLE	Thread	Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		0		Fast-Strings Enable

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2:1		Reserved
		3		Automatic Thermal Control Circuit Enable (R/W)
		6:4		Reserved
		7		Performance Monitoring Available (R)
		10:8		Reserved
		11		Branch Trace Storage Unavailable (RO)
		12		Processor Event Based Sampling Unavailable (RO)
		15:13		Reserved
		16		Enhanced Intel SpeedStep Technology Enable (R/W)
		18		ENABLE MONITOR FSM (R/W)
		21:19		Reserved
		22		Limit CPUID Maxval (R/W)
		23		xTPR Message Disable (R/W)
		33:24		Reserved
		34		XD Bit Disable (R/W)
		37:35		Reserved
		38		Turbo Mode Disable (R/W)
		63:39		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R)
		29:24		Target Offset (R/W)
		63:30		Reserved
1A4H	420	MSR_MISC_FEATURE_CONTROL		Miscellaneous Feature Control (R/W)
		0	Core	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher.
		1	Core	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher.
		63:2		Reserved
1A6H	422	MSR_OFFCORE_RSP_0	Shared	Offcore Response Event Select Register (R/W)
1A7H	423	MSR_OFFCORE_RSP_1	Shared	Offcore Response Event Select Register (R/W)
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Maximum Ratio Limit of Turbo Mode for Groups of Cores (RW)
		0		Reserved
		7:1	Package	Maximum Number of Cores in Group 0 Number active processor cores which operates under the maximum ratio limit for group 0.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		15:8	Package	Maximum Ratio Limit for Group 0 Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.
		20:16	Package	Number of Incremental Cores Added to Group 1 Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = "group 0 Max ratio limit" - "group ratio delta for group 1".
		23:21	Package	Group Ratio Delta for Group 1 An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.
		28:24	Package	Number of Incremental Cores Added to Group 2 Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = "group 1 Max ratio limit" - "group ratio delta for group 2".
		31:29	Package	Group Ratio Delta for Group 2 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.
		36:32	Package	Number of Incremental Cores Added to Group 3 Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = "group 2 Max ratio limit" - "group ratio delta for group 3".
		39:37	Package	Group Ratio Delta for Group 3 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.
		44:40	Package	Number of Incremental Cores Added to Group 4 Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = "group 3 Max ratio limit" - "group ratio delta for group 4".
		47:45	Package	Group Ratio Delta for Group 4 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.
		52:48	Package	Number of Incremental Cores Added to Group 5 Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = "group 4 Max ratio limit" - "group ratio delta for group 5".
		55:53	Package	Group Ratio Delta for Group 5 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 4.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		60:56	Package	Number of Incremental Cores Added to Group 6 Group 6, which includes the specified number of additional cores plus all the cores in group 5, operates under the group 6 turbo max ratio limit = "group 5 Max ratio limit" - "group ratio delta for group 6".
		63:61	Package	Group Ratio Delta for Group 6 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 5.
1B0H	432	IA32_ENERGY_PERF_BIAS	Thread	See Table 2-2.
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	Last Branch Record Filtering Select Register (R/W) See Section 17.9.2, "Filtering of Last Branch Records."
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
		63:9		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Thread	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1D9H	473	IA32_DEBUGCTL	Thread	Debug Control (R/W)
		0		LBR Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.
		1		BTF Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.
		5:2		Reserved
		6		TR Setting this bit to 1 enables branch trace messages to be sent.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7		BTS Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.
		8		BTINT When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.
		9		BTS_OFF_OS When set, BTS or BTM is skipped if CPL = 0.
		10		BTS_OFF_USR When set, BTS or BTM is skipped if CPL > 0.
		11		FREEZE_LBRS_ON_PMI When set, the LBR stack is frozen on a PMI request.
		12		FREEZE_PERFMON_ON_PMI When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.
		13		Reserved
		14		FREEZE_WHILE_SMM When set, freezes perfmon and trace messages while in SMM.
		31:15		Reserved
1DDH	477	MSR_LER_FROM_LIP	Thread	Last Exception Record from Linear IP (R)
1DEH	478	MSR_LER_TO_LIP	Thread	Last Exception Record to Linear IP (R)
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Core	Default Memory Types (R/W) See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Package	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Thread	Fixed-Function-Counter Control Register (R/W) See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2.
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2.
3F1H	1009	MSR_PEBB_ENABLE	Thread	See Table 2-2.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		63:0		Package C3 Residency Counter (R/O)
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	
		63:0		Package C6 Residency Counter (R/O)
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:0		Package C7 Residency Counter (R/O)
3FCH	1020	MSR_MC0_RESIDENCY	Module	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		63:0		Module C0 Residency Counter (R/O)
3FDH	1021	MSR_MC6_RESIDENCY	Module	
		63:0		Module C6 Residency Counter (R/O)
3FFH	1023	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		63:0		CORE C6 Residency Counter (R/O)
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	DS Save Area (R/W) See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers Used in RAPL Interfaces (R/O)

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
		15:13	Package	Reserved
		19:16	Package	Time Units See Section 14.10.1, "RAPL Interfaces."
		63:20		Reserved
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
		63:0		Package C2 Residency Counter (R/O)
610H	1552	MSR_PKG_POWER_LIMIT	Package	PKG RAPL Power Limit Control (R/W) See Section 14.10.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	PKG Perf Status (R/O) See Section 14.10.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameters (R/W) See Section 14.10.3, "Package RAPL Domain."
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
638H	1592	MSR_PPO_POWER_LIMIT	Package	PPO RAPL Power Limit Control (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	Base TDP Ratio (R/O) See Table 2-25.



**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O) See Table 2-25.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O) See Table 2-25.
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	ConfigTDP Control (R/W) See Table 2-25.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	ConfigTDP Control (R/W) See Table 2-25.
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)
		0		PROCHOT Status (R0)
		1		Thermal Status (R0)
		5:2		Reserved
		6		VR Therm Alert Status (R0)
		7		Reserved
		8		Electrical Design Point Status (R0)
		63:9		Reserved
6E0H	1760	IA32_TSC_DEADLINE	Core	TSC Target of Local APIC's TSC Deadline Mode (R/W) See Table 2-2.
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID Register (R/O)
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version Register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority Register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority Register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI Register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination Register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector Register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service Register Bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service Register Bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service Register Bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service Register Bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service Register Bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service Register Bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service Register Bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service Register Bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMRO	Thread	x2APIC Trigger Mode Register Bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode Register Bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode Register Bits [95:64] (R/O)

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode Register Bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode Register Bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode Register Bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode Register Bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode Register Bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request Register Bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request Register Bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request Register Bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request Register Bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request Register Bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request Register Bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request Register Bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request Register Bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status Register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt Register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command Register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt Register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt Register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor Register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 Register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 Register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error Register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count Register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count Register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration Register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI Register (W/O)
C000_0080H		IA32_EFER	Thread	Extended Feature Enables See Table 2-2.
C000_0081H		IA32_STAR	Thread	System Call Target Address (R/W) See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	IA-32e Mode System Call Target Address (R/W) See Table 2-2.
C000_0084H		IA32_FMASK	Thread	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	Map of BASE Address of FS (R/W) See Table 2-2.

**Table 2-48. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signatures 06\_57H and 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C000_0101H		IA32_GS_BASE	Thread	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	Swap Target of BASE Address of GS (R/W) See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	AUXILIARY TSC Signature (R/W) See Table 2-2

Table 2-49 lists model-specific registers that are supported by Intel® Xeon Phi™ processor 7215, 7285, 7295 series based on the Knights Mill microarchitecture.

**Table 2-49. Additional MSRs Supported by Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with DisplayFamily\_DisplayModel Signature 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
9BH	155	IA32_SMM_MONITOR_CTL	Core	SMM Monitor Configuration (R/W) This MSR is readable only if VMX is enabled, and writeable only if VMX is enabled and in SMM mode, and is used to configure the VMX MSEG base address. See Table 2-2.
480H	1152	IA32_VMX_BASIC	Core	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2.
481H	1153	IA32_VMX_PINBASED_CTL	Core	Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2.
482H	1154	IA32_VMX_PROCBASED_CTL	Core	Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)
483H	1155	IA32_VMX_EXIT_CTL	Core	Capability Reporting Register of VM-exit Controls (R/O) See Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTL	Core	Capability Reporting Register of VM-entry Controls (R/O) See Table 2-2.
485H	1157	IA32_VMX_MISC	Core	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2.
486H	1158	IA32_VMX_CRO_FIXED0	Core	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Table 2-2.
487H	1159	IA32_VMX_CRO_FIXED1	Core	Capability Reporting Register of CRO Bits Fixed to 1 (R/O) See Table 2-2.

**Table 2-49. Additional MSRs Supported by Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with DisplayFamily\_DisplayModel Signature 06\_85H**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
488H	1160	IA32_VMX_CR4_FIXED0	Core	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2.
489H	1161	IA32_VMX_CR4_FIXED1	Core	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2.
48AH	1162	IA32_VMX_VMCS_ENUM	Core	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2.
48BH	1163	IA32_VMX_PROCBASED_CTL2	Core	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Table 2-2.
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	Capability Reporting Register of EPT and VPID (R/O) See Table 2-2.
48DH	1165	IA32_VMX_TRUE_PINBASED_CTL2	Core	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2.
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL2	Core	Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Table 2-2.
48FH	1167	IA32_VMX_TRUE_EXIT_CTL2	Core	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2.
490H	1168	IA32_VMX_TRUE_ENTRY_CTL2	Core	Capability Reporting Register of VM-Entry Flex Controls (R/O) See Table 2-2.
491H	1169	IA32_VMX_FMFUNC	Core	Capability Reporting Register of VM-Function Controls (R/O) See Table 2-2.

## 2.19 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-50 lists MSRs (architectural and model-specific) that are defined across processor generations based on Intel NetBurst microarchitecture. The processor can be identified by its CPUID signatures of DisplayFamily encoding of 0FH, see Table 2-1.

- MSRs with an “IA32\_” prefix are designated as “architectural.” This means that the functions of these MSRs and their addresses remain the same for succeeding families of IA-32 processors.
- MSRs with an “MSR\_” prefix are model specific with respect to address functionalities. The column “Model Availability” lists the model encoding value(s) within the Pentium 4 and Intel Xeon processor family at the specified register address. The model encoding value of a processor can be queried using CPUID. See “CPUID—CPU Identification” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors**

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
0H	0	IA32_P5_MC_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	0, 1, 2, 3, 4, 6	Shared	See Section 2.23, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_LINE_SIZE	3, 4, 6	Shared	See Section 8.10.5, "Monitor/Mwait Address Range Determination."
10H	16	IA32_TIME_STAMP_COUNTER	0, 1, 2, 3, 4, 6	Unique	Time Stamp Counter See Table 2-2.
					On earlier processors, only the lower 32 bits are writable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writable.
17H	23	IA32_PLATFORM_ID	0, 1, 2, 3, 4, 6	Shared	Platform ID (R) See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	0, 1, 2, 3, 4, 6	Unique	APIC Location and Status (R/W) See Table 2-2. See Section 10.4.4, "Local APIC Status and Location."
2AH	42	MSR_EBC_HARD_POWERON	0, 1, 2, 3, 4, 6	Shared	Processor Hard Power-On Configuration (R/W) Enables and disables processor features. (R) Indicates current processor configuration.
		0			Output Tri-state Enabled (R) Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		1			Execute BIST (R) Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		2			In Order Queue Depth (R) Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		3			MCERR# Observation Disabled (R) Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		4			BINIT# Observation Enabled (R) Indicates whether BINIT# observation is enabled (0) or disabled (1) as determined by the strapping of A10#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		6:5			APIC Cluster ID (R) Contains the logical APIC cluster ID value as set by the strapping of A12# and A11#. The logical cluster ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		7			Bus Park Disable (R) Indicates whether bus park is enabled (0) or disabled (1) as set by the strapping of A15#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		11:8			Reserved
		13:12			Agent ID (R) Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		63:14			Reserved
2BH	43	MSR_EBC_SOFT_POWERON	0, 1, 2, 3, 4, 6	Shared	Processor Soft Power-On Configuration (R/W) Enables and disables processor features.
		0			RCNT/SCNT On Request Encoding Enable (R/W) Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).
		1			Data Error Checking Disable (R/W) Set to disable system data bus parity checking; clear to enable parity checking.
		2			Response Error Checking Disable (R/W) Set to disable (default); clear to enable.
		3			Address/Request Error Checking Disable (R/W) Set to disable (default); clear to enable.

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
		4			Initiator MCERR# Disable (R/W) Set to disable MCERR# driving for initiator bus requests (default); clear to enable.
		5			Internal MCERR# Disable (R/W) Set to disable MCERR# driving for initiator internal errors (default); clear to enable.
		6			BINIT# Driver Disable (R/W) Set to disable BINIT# driver (default); clear to enable driver.
		63:7			Reserved
2CH	44	MSR_EBC_FREQUENCY_ID	2,3, 4, 6	Shared	Processor Frequency Configuration The bit field layout of this MSR varies according to the MODEL value in the CUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2. (R) The field Indicates the current processor frequency configuration.
		15:0			Reserved
		18:16			Scalable Bus Speed (R/W) Indicates the intended scalable bus speed: <u>Encoding Scalable Bus Speed</u> 000B 100 MHz (Model 2) 000B 266 MHz (Model 3 or 4) 001B 133 MHz 010B 200 MHz 011B 166 MHz 100B 333 MHz (Model 6)
					133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
					266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6. All other values are reserved.
		23:19			Reserved

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		31:24			Core Clock Frequency to System Bus Frequency Ratio (R) The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.
		63:25			Reserved
2CH	44	MSR_EBC_FREQUENCY_ID	0, 1	Shared	Processor Frequency Configuration (R) The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.
		20:0			Reserved
		23:21			Scalable Bus Speed (R/W) Indicates the intended scalable bus speed: <u>Encoding Scalable Bus Speed</u> 000B 100 MHz All others values reserved.
		63:24			Reserved
3AH	58	IA32_FEATURE_CONTROL	3, 4, 6	Unique	Control Features in IA-32 Processor (R/W) See Table 2-2. (If CPUID.01H:ECX.[bit 5])
79H	121	IA32_BIOS_UPDT_TRIG	0, 1, 2, 3, 4, 6	Shared	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	0, 1, 2, 3, 4, 6	Unique	BIOS Update Signature ID (R/W) See Table 2-2.
9BH	155	IA32_SMM_MONITOR_CTL	3, 4, 6	Unique	SMM Monitor Configuration (R/W) See Table 2-2.
FEH	254	IA32_MTRRCAP	0, 1, 2, 3, 4, 6	Unique	MTRR Information See Section 11.11.1, "MTRR Feature Identification."
174H	372	IA32_SYSENTER_CS	0, 1, 2, 3, 4, 6	Unique	CS Register Target for CPL 0 Code (R/W) See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
175H	373	IA32_SYSENTER_ESP	0, 1, 2, 3, 4, 6	Unique	Stack Pointer for CPL 0 Stack (R/W) See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."



**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
176H	374	IA32_SYSENTER_EIP	0, 1, 2, 3, 4, 6	Unique	CPL 0 Code Entry Point (R/W) See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
179H	377	IA32_MCG_CAP	0, 1, 2, 3, 4, 6	Unique	Machine Check Capabilities (R) See Table 2-2. See Section 15.3.1.1, "IA32_MCG_CAP MSR."
17AH	378	IA32_MCG_STATUS	0, 1, 2, 3, 4, 6	Unique	Machine Check Status (R) See Table 2-2. See Section 15.3.1.2, "IA32_MCG_STATUS MSR."
17BH	379	IA32_MCG_CTL			Machine Check Feature Enable (R/W) See Table 2-2. See Section 15.3.1.3, "IA32_MCG_CTL MSR."
180H	384	MSR_MCG_RAX	0, 1, 2, 3, 4, 6	Unique	Machine Check EAX/RAX Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
181H	385	MSR_MCG_RBX	0, 1, 2, 3, 4, 6	Unique	Machine Check EBX/RBX Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
182H	386	MSR_MCG_RCX	0, 1, 2, 3, 4, 6	Unique	Machine Check ECX/RCX Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
183H	387	MSR_MCG_RDX	0, 1, 2, 3, 4, 6	Unique	Machine Check EDX/RDX Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
184H	388	MSR_MCG_RSI	0, 1, 2, 3, 4, 6	Unique	Machine Check ESI/RSI Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
185H	389	MSR_MCG_RDI	0, 1, 2, 3, 4, 6	Unique	Machine Check EDI/RDI Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
186H	390	MSR_MCG_RBP	0, 1, 2, 3, 4, 6	Unique	Machine Check EBP/RBP Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
187H	391	MSR_MCG_RSP	0, 1, 2, 3, 4, 6	Unique	Machine Check ESP/RSP Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
188H	392	MSR_MCG_RFLAGS	0, 1, 2, 3, 4, 6	Unique	Machine Check EFLAGS/RFLAG Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
189H	393	MSR_MCG_RIP	0, 1, 2, 3, 4, 6	Unique	Machine Check EIP/RIP Save State See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
18AH	394	MSR_MCG_MISC	0, 1, 2, 3, 4, 6	Unique	Machine Check Miscellaneous See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		0			DS When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.
		63:1			Reserved
18BH-18FH	395-399	MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5			Reserved

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
190H	400	MSR_MCG_R8	0, 1, 2, 3, 4, 6	Unique	Machine Check R8 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
191H	401	MSR_MCG_R9	0, 1, 2, 3, 4, 6	Unique	Machine Check R9D/R9 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
192H	402	MSR_MCG_R10	0, 1, 2, 3, 4, 6	Unique	Machine Check R10 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
193H	403	MSR_MCG_R11	0, 1, 2, 3, 4, 6	Unique	Machine Check R11 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
194H	404	MSR_MCG_R12	0, 1, 2, 3, 4, 6	Unique	Machine Check R12 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
195H	405	MSR_MCG_R13	0, 1, 2, 3, 4, 6	Unique	Machine Check R13 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
196H	406	MSR_MCG_R14	0, 1, 2, 3, 4, 6	Unique	Machine Check R14 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
197H	407	MSR_MCG_R15	0, 1, 2, 3, 4, 6	Unique	Machine Check R15 See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
198H	408	IA32_PERF_STATUS	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
199H	409	IA32_PERF_CTL	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
19AH	410	IA32_CLOCK_MODULATION	0, 1, 2, 3, 4, 6	Unique	Thermal Monitor Control (R/W) See Table 2-2. See Section 14.8.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	0, 1, 2, 3, 4, 6	Unique	Thermal Interrupt Control (R/W) See Section 14.8.2, "Thermal Monitor," and see Table 2-2.
19CH	412	IA32_THERM_STATUS	0, 1, 2, 3, 4, 6	Shared	Thermal Monitor Status (R/W) See Section 14.8.2, "Thermal Monitor," and see Table 2-2.
19DH	413	MSR_THERM2_CTL			Thermal Monitor 2 Control
			3,	Shared	For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.
			4, 6	Shared	For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.
1A0H	416	IA32_MISC_ENABLE	0, 1, 2, 3, 4, 6	Shared	Enable Miscellaneous Processor Features (R/W)

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
		0			Fast-Strings Enable. See Table 2-2.
		1			Reserved
		2			x87 FPU Fopcode Compatibility Mode Enable
		3			Thermal Monitor 1 Enable See Section 14.8.2, "Thermal Monitor," and see Table 2-2.
		4			Split-Lock Disable When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus.
					This debug feature is specific to the Pentium 4 processor.
		5			Reserved
		6			Third-Level Cache Disable (R/W) When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache.  Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CRO, the page-level cache controls, and/or the MTRRs. See Section 11.5.4, "Disabling and Enabling the L3 Cache."
		7			Performance Monitoring Available (R) See Table 2-2.
		8			Suppress Lock Enable When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.
		9			Prefetch Queue Disable When set, disables the prefetch queue. When clear (default), enables the prefetch queue.
		10			FERR# Interrupt Reporting Enable (R/W) When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled.

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
					When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt.
					This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.
		11			Branch Trace Storage Unavailable (BTS_UNAVILABLE) (R) See Table 2-2. When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.
		12			PEBS_UNAVILABLE: Processor Event Based Sampling Unavailable (R) See Table 2-2. When set, the processor does not support processor event-based sampling (PEBS); when clear, PEBS is supported.
		13	3		TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		17:14			Reserved
		18	3, 4, 6		ENABLE MONITOR FSM (R/W) See Table 2-2.

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
		19			Adjacent Cache Line Prefetch Disable (R/W) When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector.
					Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		21:20			Reserved
		22	3, 4, 6		Limit CPUID MAXVAL (R/W) See Table 2-2. Setting this can cause unexpected behavior to software that depends on the availability of CPUID leaves greater than 3.
		23		Shared	xTPR Message Disable (R/W) See Table 2-2.
		24			L1 Data Cache Context Mode (R/W) When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 11.5.6, "L1 Data Cache Context Mode." When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].
		33:25			Reserved
		34		Unique	XD Bit Disable (R/W) See Table 2-2.
		63:35			Reserved
1A1H	417	MSR_PLATFORM_BRV	3, 4, 6	Shared	Platform Feature Requirements (R)
		17:0			Reserved

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		18			PLATFORM Requirements When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.
		63:19			Reserved
1D7H	471	MSR_LER_FROM_LIP	0, 1, 2, 3, 4, 6	Unique	Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13.3, "Last Exception Records."
		31:0			From Linear IP Linear address of the last branch instruction.
		63:32			Reserved
1D7H	471	63:0		Unique	From Linear IP Linear address of the last branch instruction (If IA-32e mode is active).
1D8H	472	MSR_LER_TO_LIP	0, 1, 2, 3, 4, 6	Unique	Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13.3, "Last Exception Records."
		31:0			From Linear IP Linear address of the target of the last branch instruction.
		63:32			Reserved
1D8H	472	63:0		Unique	From Linear IP Linear address of the target of the last branch instruction (If IA-32e mode is active).
1D9H	473	MSR_DEBUGCTLA	0, 1, 2, 3, 4, 6	Unique	Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.13.1, "MSR_DEBUGCTLA MSR."
1DAH	474	MSR_LASTBRANCH_TOS	0, 1, 2, 3, 4, 6	Unique	Last Branch Record Stack TOS (R/O) Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record). See Section 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture"; and addresses 1DBH-1DEH and 680H-68FH.



**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
1DBH	475	MSR_LASTBRANCH_0	0, 1, 2	Unique	<p>Last Branch Record 0 (R/O)</p> <p>One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took.</p> <p>MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH.</p>
					See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
1DCH	477	MSR_LASTBRANCH_1	0, 1, 2	Unique	<p>Last Branch Record 1</p> <p>See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
1DDH	477	MSR_LASTBRANCH_2	0, 1, 2	Unique	<p>Last Branch Record 2</p> <p>See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
1DEH	478	MSR_LASTBRANCH_3	0, 1, 2	Unique	<p>Last Branch Record 3</p> <p>See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
200H	512	IA32_MTRR_PHYSBASE0	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Base MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
201H	513	IA32_MTRR_PHYSMASK0	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
202H	514	IA32_MTRR_PHYSBASE1	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
203H	515	IA32_MTRR_PHYSMASK1	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
204H	516	IA32_MTRR_PHYSBASE2	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
205H	517	IA32_MTRR_PHYSMASK2	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
206H	518	IA32_MTRR_PHYSBASE3	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
207H	519	IA32_MTRR_PHYSMASK3	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
208H	520	IA32_MTRR_PHYSBASE4	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
209H	521	IA32_MTRR_PHYSMASK4	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
20AH	522	IA32_MTRR_PHYSBASE5	0, 1, 2, 3, 4, 6	Shared	Variable Range Mask MTRR See Section 11.11.2.3, "Variable Range MTRRs."
20BH	523	IA32_MTRR_PHYSMASK5	0, 1, 2, 3, 4, 6	Shared	Variable Range Mask MTRR See Section 11.11.2.3, "Variable Range MTRRs."
20CH	524	IA32_MTRR_PHYSBASE6	0, 1, 2, 3, 4, 6	Shared	Variable Range Mask MTRR See Section 11.11.2.3, "Variable Range MTRRs."
20DH	525	IA32_MTRR_PHYSMASK6	0, 1, 2, 3, 4, 6	Shared	Variable Range Mask MTRR See Section 11.11.2.3, "Variable Range MTRRs."
20EH	526	IA32_MTRR_PHYSBASE7	0, 1, 2, 3, 4, 6	Shared	Variable Range Mask MTRR See Section 11.11.2.3, "Variable Range MTRRs."
20FH	527	IA32_MTRR_PHYSMASK7	0, 1, 2, 3, 4, 6	Shared	Variable Range Mask MTRR See Section 11.11.2.3, "Variable Range MTRRs."
250H	592	IA32_MTRR_FIX64K_00000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
258H	600	IA32_MTRR_FIX16K_80000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
259H	601	IA32_MTRR_FIX16K_A0000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
268H	616	IA32_MTRR_FIX4K_C0000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
269H	617	IA32_MTRR_FIX4K_C8000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
26AH	618	IA32_MTRR_FIX4K_D0000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
26BH	619	IA32_MTRR_FIX4K_D8000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
26CH	620	IA32_MTRR_FIX4K_E0000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
26DH	621	IA32_MTRR_FIX4K_E8000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
26EH	622	IA32_MTRR_FIX4K_F0000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
26FH	623	IA32_MTRR_FIX4K_F8000	0, 1, 2, 3, 4, 6	Shared	Fixed Range MTRR See Section 11.11.2.2, "Fixed Range MTRRs."
277H	631	IA32_PAT	0, 1, 2, 3, 4, 6	Unique	Page Attribute Table See Section 11.11.2.2, "Fixed Range MTRRs."
2FFH	767	IA32_MTRR_DEF_TYPE	0, 1, 2, 3, 4, 6	Shared	Default Memory Types (R/W) See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
300H	768	MSR_BPU_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
301H	769	MSR_BPU_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
302H	770	MSR_BPU_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
303H	771	MSR_BPU_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
304H	772	MSR_MS_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
305H	773	MSR_MS_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
306H	774	MSR_MS_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
307H	775	MSR_MS_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
308H	776	MSR_FLAME_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
309H	777	MSR_FLAME_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30AH	778	MSR_FLAME_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30BH	779	MSR_FLAME_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30CH	780	MSR_IQ_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30DH	781	MSR_IQ_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30EH	782	MSR_IQ_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30FH	783	MSR_IQ_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
310H	784	MSR_IQ_COUNTER4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
311H	785	MSR_IQ_COUNTER5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
360H	864	MSR_BPU_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
361H	865	MSR_BPU_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
362H	866	MSR_BPU_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
363H	867	MSR_BPU_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
364H	868	MSR_MS_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
365H	869	MSR_MS_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
366H	870	MSR_MS_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
367H	871	MSR_MS_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
368H	872	MSR_FLAME_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
369H	873	MSR_FLAME_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36AH	874	MSR_FLAME_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36BH	875	MSR_FLAME_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36CH	876	MSR_IQ_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36DH	877	MSR_IQ_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36EH	878	MSR_IQ_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36FH	879	MSR_IQ_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
370H	880	MSR_IQ_CCCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
371H	881	MSR_IQ_CCCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
3A0H	928	MSR_BSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A1H	929	MSR_BSU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A2H	930	MSR_FSB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A3H	931	MSR_FSB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A4H	932	MSR_FIRM_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A5H	933	MSR_FIRM_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A6H	934	MSR_FLAME_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A7H	935	MSR_FLAME_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
3A8H	936	MSR_DAC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A9H	937	MSR_DAC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AAH	938	MSR_MOB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ABH	939	MSR_MOB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ACH	940	MSR_PMH_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ADH	941	MSR_PMH_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AEH	942	MSR_SAA_T_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AFH	943	MSR_SAA_T_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B0H	944	MSR_U2L_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B1H	945	MSR_U2L_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B2H	946	MSR_BPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B3H	947	MSR_BPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B4H	948	MSR_IS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B5H	949	MSR_IS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B6H	950	MSR_ITLB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B7H	951	MSR_ITLB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B8H	952	MSR_CRU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B9H	953	MSR_CRU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BAH	954	MSR_IQ_ESCR0	0, 1, 2	Shared	See Section 18.6.3.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.
3BBH	955	MSR_IQ_ESCR1	0, 1, 2	Shared	See Section 18.6.3.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
3BCH	956	MSR_RAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BDH	957	MSR_RAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BEH	958	MSR_SSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C0H	960	MSR_MS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C1H	961	MSR_MS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C2H	962	MSR_TBPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C3H	963	MSR_TBPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C4H	964	MSR_TC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C5H	965	MSR_TC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C8H	968	MSR_IX_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C9H	969	MSR_IX_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CAH	970	MSR_ALF_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CBH	971	MSR_ALF_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CCH	972	MSR_CRU_ESCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CDH	973	MSR_CRU_ESCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3E0H	992	MSR_CRU_ESCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3E1H	993	MSR_CRU_ESCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3F0H	1008	MSR_TC_PRECISE_EVENT	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3F1H	1009	MSR_PEBS_ENABLE	0, 1, 2, 3, 4, 6	Shared	Processor Event Based Sampling (PEBS) (R/W) Controls the enabling of processor event sampling and replay tagging.
		12:0			See <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a> .
		23:13			Reserved
		24			UOP Tag Enables replay tagging when set.

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
		25			ENABLE_PEBS_MY_THR (R/W) Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.6.4.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is called ENABLE_PEBS in IA-32 processors that do not support Intel Hyper-Threading Technology.
		26			ENABLE_PEBS_OTH_THR (R/W) Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.6.4.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is reserved for IA-32 processors that do not support Intel Hyper-Threading Technology.
		63:27			Reserved
3F2H	1010	MSR_PEBS_MATRIX_VERT	0, 1, 2, 3, 4, 6	Shared	See <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a> .
400H	1024	IA32_MCO_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
401H	1025	IA32_MCO_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCI_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCI_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MCO_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCI_MISC MSRs." The IA32_MCO_MISC MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
405H	1029	IA32_MC1_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCI_STATUS MSRS."

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
406H	1030	IA32_MC1_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC		Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC1_MISC MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.



Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
40FH	1039	IA32_MC3_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRIV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC4_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	3, 4, 6	Unique	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	3, 4, 6	Unique	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	3, 4, 6	Unique	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
483H	1155	IA32_VMX_EXIT_CTL	3, 4, 6	Unique	Capability Reporting Register of VM-Exit Controls (R/O) See Appendix A.4, "VM-Exit Controls," and see Table 2-2.

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
484H	1156	IA32_VMX_ENTRY_CTL5	3, 4, 6	Unique	Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, "VM-Entry Controls," and see Table 2-2.
485H	1157	IA32_VMX_MISC	3, 4, 6	Unique	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Appendix A.6, "Miscellaneous Data," and see Table 2-2.
486H	1158	IA32_VMX_CRO_FIXED0	3, 4, 6	Unique	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
487H	1159	IA32_VMX_CRO_FIXED1	3, 4, 6	Unique	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
488H	1160	IA32_VMX_CR4_FIXED0	3, 4, 6	Unique	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.
489H	1161	IA32_VMX_CR4_FIXED1	3, 4, 6	Unique	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.
48AH	1162	IA32_VMX_VMCS_ENUM	3, 4, 6	Unique	Capability Reporting Register of VMCS Field Enumeration (R/O) See Appendix A.9, "VMCS Enumeration," and see Table 2-2.
48BH	1163	IA32_VMX_PROCBASED_CTL52	3, 4, 6	Unique	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
600H	1536	IA32_DS_AREA	0, 1, 2, 3, 4, 6	Unique	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	3, 4, 6	Unique	Last Branch Record 0 (R/W) One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the source instruction for one of the last 16 branches, exceptions, or interrupts taken by the processor.

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
					The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH, which performed the same function for early releases. See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
681H	1665	MSR_LASTBRANCH_1_FROM_IP	3, 4, 6	Unique	Last Branch Record 1 See description of MSR_LASTBRANCH_0 at 680H.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	3, 4, 6	Unique	Last Branch Record 2 See description of MSR_LASTBRANCH_0 at 680H.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	3, 4, 6	Unique	Last Branch Record 3 See description of MSR_LASTBRANCH_0 at 680H.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	3, 4, 6	Unique	Last Branch Record 4 See description of MSR_LASTBRANCH_0 at 680H.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	3, 4, 6	Unique	Last Branch Record 5 See description of MSR_LASTBRANCH_0 at 680H.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	3, 4, 6	Unique	Last Branch Record 6 See description of MSR_LASTBRANCH_0 at 680H.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	3, 4, 6	Unique	Last Branch Record 7 See description of MSR_LASTBRANCH_0 at 680H.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	3, 4, 6	Unique	Last Branch Record 8 See description of MSR_LASTBRANCH_0 at 680H.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	3, 4, 6	Unique	Last Branch Record 9 See description of MSR_LASTBRANCH_0 at 680H.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	3, 4, 6	Unique	Last Branch Record 10 See description of MSR_LASTBRANCH_0 at 680H.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	3, 4, 6	Unique	Last Branch Record 11 See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	3, 4, 6	Unique	Last Branch Record 12 See description of MSR_LASTBRANCH_0 at 680H.

Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	3, 4, 6	Unique	Last Branch Record 13 See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	3, 4, 6	Unique	Last Branch Record 14 See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	3, 4, 6	Unique	Last Branch Record 15 See description of MSR_LASTBRANCH_0 at 680H.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	3, 4, 6	Unique	Last Branch Record 0 (R/W) One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the destination instruction for one of the last 16 branches, exceptions, or interrupts that the processor took. See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	3, 4, 6	Unique	Last Branch Record 1 See description of MSR_LASTBRANCH_0 at 6C0H.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	3, 4, 6	Unique	Last Branch Record 2 See description of MSR_LASTBRANCH_0 at 6C0H.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	3, 4, 6	Unique	Last Branch Record 3 See description of MSR_LASTBRANCH_0 at 6C0H.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	3, 4, 6	Unique	Last Branch Record 4 See description of MSR_LASTBRANCH_0 at 6C0H.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	3, 4, 6	Unique	Last Branch Record 5 See description of MSR_LASTBRANCH_0 at 6C0H.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	3, 4, 6	Unique	Last Branch Record 6 See description of MSR_LASTBRANCH_0 at 6C0H.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	3, 4, 6	Unique	Last Branch Record 7 See description of MSR_LASTBRANCH_0 at 6C0H.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	3, 4, 6	Unique	Last Branch Record 8 See description of MSR_LASTBRANCH_0 at 6C0H.

**Table 2-50. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	3, 4, 6	Unique	Last Branch Record 9 See description of MSR_LASTBRANCH_0 at 6COH.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	3, 4, 6	Unique	Last Branch Record 10 See description of MSR_LASTBRANCH_0 at 6COH.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	3, 4, 6	Unique	Last Branch Record 11 See description of MSR_LASTBRANCH_0 at 6COH.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	3, 4, 6	Unique	Last Branch Record 12 See description of MSR_LASTBRANCH_0 at 6COH.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	3, 4, 6	Unique	Last Branch Record 13 See description of MSR_LASTBRANCH_0 at 6COH.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	3, 4, 6	Unique	Last Branch Record 14 See description of MSR_LASTBRANCH_0 at 6COH.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	3, 4, 6	Unique	Last Branch Record 15 See description of MSR_LASTBRANCH_0 at 6COH.
C000_0080H		IA32_EFER	3, 4, 6	Unique	Extended Feature Enables See Table 2-2.
C000_0081H		IA32_STAR	3, 4, 6	Unique	System Call Target Address (R/W) See Table 2-2.
C000_0082H		IA32_LSTAR	3, 4, 6	Unique	IA-32e Mode System Call Target Address (R/W) See Table 2-2.
C000_0084H		IA32_FMASK	3, 4, 6	Unique	System Call Flag Mask (R/W) See Table 2-2.
C000_0100H		IA32_FS_BASE	3, 4, 6	Unique	Map of BASE Address of FS (R/W) See Table 2-2.
C000_0101H		IA32_GS_BASE	3, 4, 6	Unique	Map of BASE Address of GS (R/W) See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	3, 4, 6	Unique	Swap Target of BASE Address of GS (R/W) See Table 2-2.

**NOTES**

1. For HT-enabled processors, there may be more than one logical processors per physical unit. If an MSR is Shared, this means that one MSR is shared between logical processors. If an MSR is unique, this means that each logical processor has its own MSR.

## 2.19.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-51 apply to Intel® Xeon® Processor MP with up to 8MB level three cache. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 3 or 4 (see CPUID instruction for more details).

**Table 2-51. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache**

Register Address	Register Name Fields and Flags	Model Availability	Shared/Unique	Bit Description
107CCH	MSR_IFSB_BUSQ0	3, 4	Shared	IFSB BUSQ Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CDH	MSR_IFSB_BUSQ1	3, 4	Shared	IFSB BUSQ Event Control and Counter Register (R/W)
107CEH	MSR_IFSB_SNPQ0	3, 4	Shared	IFSB SNPQ Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH	MSR_IFSB_SNPQ1	3, 4	Shared	IFSB SNPQ Event Control and Counter Register (R/W)
107DOH	MSR_EFSB_DRDY0	3, 4	Shared	EFSB DRDY Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D1H	MSR_EFSB_DRDY1	3, 4	Shared	EFSB DRDY Event Control and Counter Register (R/W)
107D2H	MSR_IFSB_CTL6	3, 4	Shared	IFSB Latency Event Control Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D3H	MSR_IFSB_CNTR7	3, 4	Shared	IFSB Latency Event Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."

The MSRs listed in Table 2-52 apply to Intel® Xeon® Processor 7100 series. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 6 (See CPUID instruction for more details.). The performance monitoring MSRs listed in Table 2-52 are shared between logical processors in the same core, but are replicated for each core.

**Table 2-52. MSRs Unique to Intel® Xeon® Processor 7100 Series**

Register Address	Register Name Fields and Flags	Model Availability	Shared/Unique	Bit Description
107CCH	MSR_EMON_L3_CTR_CTL0	6	Shared	GBUSQ Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."

**Table 2-52. MSRs Unique to Intel® Xeon® Processor 7100 Series (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique	Bit Description
107CDH		MSR_EMON_L3_CTR_CTL1	6	Shared	GBUSQ Event Control and Counter Register (R/W)
107CEH		MSR_EMON_L3_CTR_CTL2	6	Shared	GSNPQ Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH		MSR_EMON_L3_CTR_CTL3	6	Shared	GSNPQ Event Control and Counter Register (R/W)
107D0H		MSR_EMON_L3_CTR_CTL4	6	Shared	FSB Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D1H		MSR_EMON_L3_CTR_CTL5	6	Shared	FSB Event Control and Counter Register (R/W)
107D2H		MSR_EMON_L3_CTR_CTL6	6	Shared	FSB Event Control and Counter Register (R/W)
107D3H		MSR_EMON_L3_CTR_CTL7	6	Shared	FSB Event Control and Counter Register (R/W)

## 2.20 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 2-53. The column "Shared/Unique" applies to Intel Core Duo processor. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

**Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/Unique	Bit Description
Hex	Dec			
0H	0	P5_MC_ADDR	Unique	See Section 2.23, "MSRs in Pentium Processors," and see Table 2-2.
1H	1	P5_MC_TYPE	Unique	See Section 2.23, "MSRs in Pentium Processors," and see Table 2-2.
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and see Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	Platform ID (R) See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location," and see Table 2-2.

Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
2AH	42	MSR_EBL_CR_POWERON	Shared	Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved
		1		Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		MCERR# Drive Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		Address Parity Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		6: 5		Reserved
		7		BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled
		9		Execute BIST (R/O) 1 = Enabled; 0 = Disabled
		10		MCERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled
		11		Reserved
		12		BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled
		13		Reserved
		14		1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		APIC Cluster ID (R/O)
		18		System Bus Frequency (R/O) 0 = 100 MHz 1 = Reserved
		19		Reserved
		21: 20		Symmetric Arbitration ID (R/O)
26:22	Clock Frequency Ratio (R/O)			



**Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Unique	Control Features in IA-32 Processor (R/W) See Table 2-2.
40H	64	MSR_LASTBRANCH_0	Unique	Last Branch Record 0 (R/W) One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the 'to' address. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."</li> </ul>
41H	65	MSR_LASTBRANCH_1	Unique	Last Branch Record 1 (R/W) See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	Unique	Last Branch Record 2 (R/W) See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	Unique	Last Branch Record 3 (R/W) See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	Unique	Last Branch Record 4 (R/W) See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	Unique	Last Branch Record 5 (R/W) See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	Unique	Last Branch Record 6 (R/W) See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	Unique	Last Branch Record 7 (R/W) See description of MSR_LASTBRANCH_0.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	BIOS Update Trigger Register (W) See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	BIOS Update Signature ID (RO) See Table 2-2.
C1H	193	IA32_PMC0	Unique	Performance Counter Register See Table 2-2.
C2H	194	IA32_PMC1	Unique	Performance Counter Register See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	Scaleable Bus Speed (RO) This field indicates the scaleable bus clock speed:
		2:0		<ul style="list-style-type: none"> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> </ul> 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.
		63:3		Reserved

Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
E7H	231	IA32_MPERF	Unique	Maximum Performance Frequency Clock Count (RW) See Table 2-2.
E8H	232	IA32_APERF	Unique	Actual Performance Frequency Clock Count (RW) See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	Control Register 3 Used to configure the L2 Cache.
		0		L2 Hardware Enabled (RO) 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved
		8		L2 Enabled (R/W) 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved
		23		L2 Not Present (RO) 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	Global Machine Check Status
		0		RIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
		1		EIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.

**Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		2		MCIP When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFEVTSELO	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	Clock Modulation (R/W) See Table 2-2.
19BH	411	IA32_THERM_INTERRUPT	Unique	Thermal Interrupt Control (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Unique	Thermal Monitor Status (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor".
19DH	413	MSR_THERM2_CTL	Unique	Thermal Monitor 2 Control
		15:0		Reserved
		16		TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16		Reserved
1A0H	416	IA32_MISC_ENABLE		Enable Miscellaneous Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		2:0		Reserved
		3	Unique	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2.
		6:4		Reserved
		7	Shared	Performance Monitoring Available (R) See Table 2-2.

Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		9:8		Reserved
		10	Shared	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	Branch Trace Storage Unavailable (RO) See Table 2-2.
		12		Reserved
		13	Shared	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		15:14		Reserved
		16	Shared	Enhanced Intel SpeedStep Technology Enable (R/W) 1 = Enhanced Intel SpeedStep Technology enabled
		18	Shared	ENABLE MONITOR FSM (R/W) See Table 2-2.
		19		Reserved
		22	Shared	Limit CPUID Maxval (R/W) See Table 2-2. Setting this bit may cause behavior in software that depends on the availability of CPUID leaves greater than 2.
		33:23		Reserved
		34	Shared	XD Bit Disable (R/W) See Table 2-2.
		63:35		Reserved
1C9H	457	MSR_LASTBRANCH_TOS	Unique	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_O_FROM_IP (at 40H).

**Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Unique	Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Unique	Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
200H	512	MTRRphysBase0	Unique	Memory Type Range Registers
201H	513	MTRRphysMask0	Unique	Memory Type Range Registers
202H	514	MTRRphysBase1	Unique	Memory Type Range Registers
203H	515	MTRRphysMask1	Unique	Memory Type Range Registers
204H	516	MTRRphysBase2	Unique	Memory Type Range Registers
205H	517	MTRRphysMask2	Unique	Memory Type Range Registers
206H	518	MTRRphysBase3	Unique	Memory Type Range Registers
207H	519	MTRRphysMask3	Unique	Memory Type Range Registers
208H	520	MTRRphysBase4	Unique	Memory Type Range Registers
209H	521	MTRRphysMask4	Unique	Memory Type Range Registers
20AH	522	MTRRphysBase5	Unique	Memory Type Range Registers
20BH	523	MTRRphysMask5	Unique	Memory Type Range Registers
20CH	524	MTRRphysBase6	Unique	Memory Type Range Registers
20DH	525	MTRRphysMask6	Unique	Memory Type Range Registers
20EH	526	MTRRphysBase7	Unique	Memory Type Range Registers
20FH	527	MTRRphysMask7	Unique	Memory Type Range Registers
250H	592	MTRRfix64K_00000	Unique	Memory Type Range Registers
258H	600	MTRRfix16K_80000	Unique	Memory Type Range Registers
259H	601	MTRRfix16K_A0000	Unique	Memory Type Range Registers
268H	616	MTRRfix4K_C0000	Unique	Memory Type Range Registers
269H	617	MTRRfix4K_C8000	Unique	Memory Type Range Registers
26AH	618	MTRRfix4K_D0000	Unique	Memory Type Range Registers
26BH	619	MTRRfix4K_D8000	Unique	Memory Type Range Registers
26CH	620	MTRRfix4K_E0000	Unique	Memory Type Range Registers
26DH	621	MTRRfix4K_E8000	Unique	Memory Type Range Registers
26EH	622	MTRRfix4K_F0000	Unique	Memory Type Range Registers
26FH	623	MTRRfix4K_F8000	Unique	Memory Type Range Registers

Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	Default Memory Types (R/W) See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MCO_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	MSR_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	MSR_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
413H	1043	MSR_MC3_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
414H	1044	MSR_MC5_CTL	Unique	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
415H	1045	MSR_MC5_STATUS	Unique	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
416H	1046	MSR_MC5_ADDR	Unique	Machine Check Error Reporting Register - contains the address of the code or data memory location that produced the machine-check error if the ADDR flag in the IA32_MCi_STATUS register is set.
417H	1047	MSR_MC5_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
480H	1152	IA32_VMX_BASIC	Unique	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information". (If CPUID.01H:ECX.[bit 5])
481H	1153	IA32_VMX_PINBASED_CTL	Unique	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls". (If CPUID.01H:ECX.[bit 5])
482H	1154	IA32_VMX_PROCBASED_CTL	Unique	Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls". (If CPUID.01H:ECX.[bit 5])
483H	1155	IA32_VMX_EXIT_CTL	Unique	Capability Reporting Register of VM-Exit Controls (R/O) See Appendix A.4, "VM-Exit Controls". (If CPUID.01H:ECX.[bit 5])
484H	1156	IA32_VMX_ENTRY_CTL	Unique	Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, "VM-Entry Controls". (If CPUID.01H:ECX.[bit 5])
485H	1157	IA32_VMX_MISC	Unique	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Appendix A.6, "Miscellaneous Data". (If CPUID.01H:ECX.[bit 5])
486H	1158	IA32_VMX_CRO_FIXED0	Unique	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Appendix A.7, "VMX-Fixed Bits in CRO". (If CPUID.01H:ECX.[bit 5])

**Table 2-53. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
487H	1159	IA32_VMX_CR0_FIXED1	Unique	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Appendix A.7, “VMX-Fixed Bits in CR0”. (If CPUID.01H:ECX.[bit 5])
488H	1160	IA32_VMX_CR4_FIXED0	Unique	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Appendix A.8, “VMX-Fixed Bits in CR4”. (If CPUID.01H:ECX.[bit 5])
489H	1161	IA32_VMX_CR4_FIXED1	Unique	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Appendix A.8, “VMX-Fixed Bits in CR4”. (If CPUID.01H:ECX.[bit 5])
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	Capability Reporting Register of VMCS Field Enumeration (R/O) See Appendix A.9, “VMCS Enumeration”. (If CPUID.01H:ECX.[bit 5])
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	Unique	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, “VM-Execution Controls”. (If CPUID.01H:ECX.[bit 5] and IA32_VMX_PROCBASED_CTLSS[bit 63])
600H	1536	IA32_DS_AREA	Unique	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, “Debug Store (DS) Mechanism.”
		31:0		DS Buffer Management Area Linear address of the first byte of the DS buffer management area.
		63:32		Reserved
C000_0080H		IA32_EFER	Unique	See Table 2-2.
		10:0		Reserved
		11		Execute Disable Bit Enable
		63:12		Reserved

## 2.21 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.22 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

**Table 2-54. MSRs in Pentium M Processors**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.23, “MSRs in Pentium Processors.”



**Table 2-54. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
1H	1	P5_MC_TYPE	See Section 2.23, "MSRs in Pentium Processors."
10H	16	IA32_TIME_STAMP_COUNTER	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Platform ID (R) See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
2AH	42	MSR_EBL_CR_POWERON	Processor Hard Power-On Configuration (R/W) Enables and disables processor features. (R) Indicates current processor configuration.
		0	Reserved
		1	Data Error Checking Enable (R) 0 = Disabled Always 0 on the Pentium M processor.
		2	Response Error Checking Enable (R) 0 = Disabled Always 0 on the Pentium M processor.
		3	MCERR# Drive Enable (R) 0 = Disabled Always 0 on the Pentium M processor.
		4	Address Parity Enable (R) 0 = Disabled Always 0 on the Pentium M processor.
		6:5	Reserved
		7	BINIT# Driver Enable (R) 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		8	Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled
		9	Execute BIST (R/O) 1 = Enabled; 0 = Disabled
		10	MCERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		11	Reserved
		12	BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		13	Reserved

Table 2-54. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		14	1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes Always 0 on the Pentium M processor.
		15	Reserved
		17:16	APIC Cluster ID (R/O) Always 00B on the Pentium M processor.
		18	System Bus Frequency (R/O) 0 = 100 MHz 1 = Reserved Always 0 on the Pentium M processor.
		19	Reserved
		21:20	Symmetric Arbitration ID (R/O) Always 00B on the Pentium M processor.
		26:22	Clock Frequency Ratio (R/O)
40H	64	MSR_LASTBRANCH_0	Last Branch Record 0 (R/W) One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the to address. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H.</li> <li>▪ Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)".</li> </ul>
41H	65	MSR_LASTBRANCH_1	Last Branch Record 1 (R/W) See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	Last Branch Record 2 (R/W) See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	Last Branch Record 3 (R/W) See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	Last Branch Record 4 (R/W) See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	Last Branch Record 5 (R/W) See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	Last Branch Record 6 (R/W) See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	Last Branch Record 7 (R/W) See description of MSR_LASTBRANCH_0.
119H	281	MSR_BBL_CR_CTL	Control Register Used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response.
		63:0	Reserved
11EH	281	MSR_BBL_CR_CTL3	Control register 3 Used to configure the L2 Cache.

**Table 2-54. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		0	L2 Hardware Enabled (RO) 1 = If the L2 is hardware-enabled. 0 = Indicates if the L2 is hardware-disabled.
		4:1	Reserved
		5	ECC Check Enable (RO) This bit enables ECC checking on the cache data bus. ECC is always generated on write cycles. 0 = Disabled (default) 1 = Enabled For the Pentium M processor, ECC checking on the cache data bus is always enabled.
		7:6	Reserved
		8	L2 Enabled (R/W) 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9	Reserved
		23	L2 Not Present (RO) 0 = L2 Present 1 = L2 Not Present
		63:24	Reserved
179H	377	IA32_MCG_CAP	Read-only register that provides information about the machine-check architecture of the processor.
		7:0	Count (RO) Indicates the number of hardware unit error reporting banks available in the processor.
		8	IA32_MCG_CTL Present (RO) 1 = Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH. 0 = Not supported.
		63:9	Reserved
17AH	378	IA32_MCG_STATUS	Global Machine Check Status
		0	RIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
		1	EIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.

Table 2-54. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		2	MCIP When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3	Reserved
198H	408	IA32_PERF_STATUS	See Table 2-2.
199H	409	IA32_PERF_CTL	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation (R/W). See Table 2-2. See Section 14.8.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Thermal Monitor Status (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor."
19DH	413	MSR_THERM2_CTL	Thermal Monitor 2 Control
		15:0	Reserved
		16	TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16	Reserved
1A0H	416	IA32_MISC_ENABLE	Enable Miscellaneous Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.
		2:0	Reserved

**Table 2-54. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		3	Automatic Thermal Control Circuit Enable (R/W) 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows processor clocks to be automatically modulated based on the processor's thermal sensor operation. 0 = Disabled (default). The automatic thermal control circuit enable bit determines if the thermal control circuit (TCC) will be activated when the processor's internal thermal sensor determines the processor is about to exceed its maximum operating temperature. When the TCC is activated and TM1 is enabled, the processors clocks will be forced to a 50% duty cycle. BIOS must enable this feature. The bit should not be confused with the on-demand thermal control circuit enable bit.
		6:4	Reserved
		7	Performance Monitoring Available (R) 1 = Performance monitoring enabled. 0 = Performance monitoring disabled.
		9:8	Reserved
		10	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor. 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage.
			Branch Trace Storage Unavailable (RO) 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported
		12	Processor Event Based Sampling Unavailable (RO) 1 = Processor does not support processor event based sampling (PEBS); 0 = PEBS is supported. The Pentium M processor does not support PEBS.
		15:13	Reserved
		16	Enhanced Intel SpeedStep Technology Enable (R/W) 1 = Enhanced Intel SpeedStep Technology enabled. On the Pentium M processor, this bit may be configured to be read-only.
		22:17	Reserved
		23	xTPR Message Disable (R/W) When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority. The default is processor specific.
		63:24	Reserved

Table 2-54. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
1C9H	457	MSR_LASTBRANCH_TOS	Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: <ul style="list-style-type: none"> <li>MSR_LASTBRANCH_0_FROM_IP (at 40H).</li> <li>Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)".</li> </ul>
1D9H	473	MSR_DEBUGCTLB	Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."
1DDH	477	MSR_LER_TO_LIP	Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.16.2, "Last Branch and Last Exception MSRs."
1DEH	478	MSR_LER_FROM_LIP	Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.16.2, "Last Branch and Last Exception MSRs."
2FFH	767	IA32_MTRR_DEF_TYPE	Default Memory Types (R/W) Sets the memory type for the regions of physical memory that are not mapped by the MTRRs. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MCO_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	See Section 14.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	See Chapter 15.3.2.2, "IA32_MCi_STATUS MSRS."

**Table 2-54. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
40AH	1034	IA32_MC2_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	MSR_MC4_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	MSR_MC3_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	MSR_MC3_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	MSR_MC3_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
600H	1536	IA32_DS_AREA	DS Save Area (R/W) See Table 2-2. Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
		31:0	DS Buffer Management Area Linear address of the first byte of the DS buffer management area.
		63:32	Reserved

## 2.22 MSRS IN THE P6 FAMILY PROCESSORS

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as "architectural" and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

**Table 2-55. MSRs in the P6 Family Processors**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.23, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.23, "MSRs in Pentium Processors."
10H	16	TSC	See Section 17.17, "Time-Stamp Counter."

Table 2-55. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
17H	23	IA32_PLATFORM_ID	Platform ID (R) The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.
		49:0	Reserved
		52:50	Platform Id (R) Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
		56:53	L2 Cache Latency Read.
		59:57	Reserved
		60	Clock Frequency Ratio Read.
		63:61	Reserved
		1BH	27
7:0	Reserved		
8	Boot Strap Processor Indicator Bit 1 = BSP		
10:9	Reserved		
11	APIC Global Enable Bit - Permanent till reset 1 = Enabled 0 = Disabled		
31:12	APIC Base Address.		
63:32	Reserved		
2AH	42	EBL_CR_POWERON	Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.
		0	Reserved <sup>1</sup>
		1	Data Error Checking Enable (R/W) 1 = Enabled 0 = Disabled
		2	Response Error Checking Enable FRCERR Observation Enable (R/W) 1 = Enabled 0 = Disabled
		3	AERR# Drive Enable (R/W) 1 = Enabled 0 = Disabled



**Table 2-55. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		4	BERR# Enable for Initiator Bus Requests (R/W) 1 = Enabled 0 = Disabled
		5	Reserved
		6	BERR# Driver Enable for Initiator Internal Errors (R/W) 1 = Enabled 0 = Disabled
		7	BINIT# Driver Enable (R/W) 1 = Enabled 0 = Disabled
		8	Output Tri-state Enabled (R) 1 = Enabled 0 = Disabled
		9	Execute BIST (R) 1 = Enabled 0 = Disabled
		10	AERR# Observation Enabled (R) 1 = Enabled 0 = Disabled
		11	Reserved
		12	BINIT# Observation Enabled (R) 1 = Enabled 0 = Disabled
		13	In Order Queue Depth (R) 1 = 1 0 = 8
		14	1-MByte Power on Reset Vector (R) 1 = 1MByte 0 = 4GBytes
		15	FRC Mode Enable (R) 1 = Enabled 0 = Disabled
		17:16	APIC Cluster ID (R)
		19:18	System Bus Frequency (R) 00 = 66MHz 10 = 100Mhz 01 = 133MHz 11 = Reserved
		21: 20	Symmetric Arbitration ID (R)
		25:22	Clock Frequency Ratio (R)
		26	Low Power Mode Enable (R/W)

Table 2-55. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		27	Clock Frequency Ratio
		63:28	Reserved <sup>1</sup>
33H	51	MSR_TEST_CTRL	Test Control Register
		29:0	Reserved
		30	Streaming Buffer Disable
		31	Disable LOCK# Assertion for split locked access.
79H	121	BIOS_UPDT_TRIG	BIOS Update Trigger Register.
88H	136	BBL_CR_D0[63:0]	Chunk 0 data register D[63:0]: used to write to and read from the L2
89H	137	BBL_CR_D1[63:0]	Chunk 1 data register D[63:0]: used to write to and read from the L2
8AH	138	BBL_CR_D2[63:0]	Chunk 2 data register D[63:0]: used to write to and read from the L2
8BH	139	BIOS_SIGN/BBL_CR_D3[63:0]	BIOS Update Signature Register or Chunk 3 data register D[63:0] Used to write to and read from the L2 depending on the usage model.
C1H	193	PerfCtr0 (PERFCTR0)	Performance Counter Register See Table 2-2.
C2H	194	PerfCtr1 (PERFCTR1)	Performance Counter Register See Table 2-2.
FEH	254	MTRRcap	Memory Type Range Registers
116H	278	BBL_CR_ADDR [63:0]	Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses.
		BBL_CR_ADDR [63:32]	Reserved,
		BBL_CR_ADDR [31:3]	Address bits [35:3]
		BBL_CR_ADDR [2:0]	Reserved Set to 0.
118H	280	BBL_CR_DECC[63:0]	Data ECC register D[7:0]: used to write ECC and read ECC to/from L2
119H	281	BBL_CR_CTL	Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response
		BL_CR_CTL[63:22]	Reserved
		BBL_CR_CTL[21]	Processor number <sup>2</sup> Disable = 1 Enable = 0 Reserved

**Table 2-55. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		BBL_CR_CTL[20:19] BBL_CR_CTL[18] BBL_CR_CTL[17] BBL_CR_CTL[16] BBL_CR_CTL[15:14] BBL_CR_CTL[13:12]  BBL_CR_CTL[11:10]  BBL_CR_CTL[9:8] BBL_CR_CTL[7] BBL_CR_CTL[6:5]  BBL_CR_CTL[4:0]	User supplied ECC Reserved L2 Hit Reserved State from L2 Modified - 11, Exclusive - 10, Shared - 01, Invalid - 00 Way from L2 Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11 Way to L2 Reserved State to L2  L2 Command 01100 Data Read w/ LRU update (RLU) 01110 Tag Read w/ Data Read (TRR) 01111 Tag Inquire (TI) 00010 L2 Control Register Read (CR) 00011 L2 Control Register Write (CW) 010 + MESI encode Tag Write w/ Data Read (TWR) 111 + MESI encode Tag Write w/ Data Write (TWW) 100 + MESI encode Tag Write (TW)
11AH	282	BBL_CR_TRIG	Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.
11BH	283	BBL_CR_BUSY	Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY
11EH	286	BBL_CR_CTL3  BBL_CR_CTL3[63:26] BBL_CR_CTL3[25] BBL_CR_CTL3[24] BBL_CR_CTL3[23]  BBL_CR_CTL3[22:20] 111 110 101 100 011 010 001 000  BBL_CR_CTL3[19] BBL_CR_CTL3[18]	Control register 3: used to configure the L2 Cache  Reserved Cache bus fraction (read only) Reserved L2 Hardware Disable (read only)  L2 Physical Address Range support 64GBytes 32GBytes 16GBytes 8GBytes 4GBytes 2GBytes 1GBytes 512MBytes  Reserved Cache State error checking enable (read/write)

Table 2-55. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		BBL_CR_CTL3[17:13] 00001 00010 00100 01000 10000  BBL_CR_CTL3[12:11] BBL_CR_CTL3[10:9] 00 01 10 11  BBL_CR_CTL3[8] BBL_CR_CTL3[7] BBL_CR_CTL3[6] BBL_CR_CTL3[5] BBL_CR_CTL3[4:1] BBL_CR_CTL3[0]	Cache size per bank (read/write) 256KBytes 512KBytes 1MByte 2MByte 4MBytes  Number of L2 banks (read only) L2 Associativity (read only) Direct Mapped 2 Way 4 Way Reserved  L2 Enabled (read/write) CRTN Parity Check Enable (read/write) Address Parity Check Enable (read/write) ECC Check Enable (read/write) L2 Cache Latency (read/write) L2 Configured (read/write )
174H	372	SYSENTER_CS_MSR	CS register target for CPL 0 code
175H	373	SYSENTER_ESP_MSR	Stack pointer for CPL 0 stack
176H	374	SYSENTER_EIP_MSR	CPL 0 code entry point
179H	377	MCG_CAP	Machine Check Global Control Register
17AH	378	MCG_STATUS	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
17BH	379	MCG_CTL	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
186H	390	PerfEvtSel0 (EVNTSEL0)	Performance Event Select Register 0 (R/W)
		7:0	Event Select Refer to Performance Counter section for a list of event encodings.
		15:8	UMASK (Unit Mask) Unit mask register set to 0 to enable all count options.
		16	USER Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	OS Controls the counting of events at Privilege level of 0.

**Table 2-55. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		18	E Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	PC Enabled the signaling of performance counter overflow via BPO pin
		20	INT Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		22	ENABLE Enables the counting of performance events in both counters 1 = Enable 0 = Disable
		23	INV Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted
		31:24	CMASK (Counter Mask)
187H	391	PerfEvtSel1 (EVNTSEL1)	Performance Event Select for Counter 1 (R/W)
		7:0	Event Select Refer to Performance Counter section for a list of event encodings.
		15:8	UMASK (Unit Mask) Unit mask register set to 0 to enable all count options.
		16	USER Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	OS Controls the counting of events at Privilege level of 0.
		18	E Occurrence/Duration Mode Select. 1 = Occurrence 0 = Duration
		19	PC Enabled the signaling of performance counter overflow via BPO pin.

Table 2-55. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		20	INT Enables the signaling of counter overflow via input to APIC. 1 = Enable 0 = Disable
		23	INV Inverts the result of the CMASK condition. 1 = Inverted 0 = Non-Inverted
		31:24	CMASK (Counter Mask)
1D9H	473	DEBUGCTLMR	Enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode.
		0	Enable/Disable Last Branch Records
		1	Branch Trap Flag
		2	Performance Monitoring/Break Point Pins
		3	Performance Monitoring/Break Point Pins
		4	Performance Monitoring/Break Point Pins
		5	Performance Monitoring/Break Point Pins
		6	Enable/Disable Execution Trace Messages
31:7	Reserved		
1DBH	475	LASTBRANCHFROMIP	32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.
1DCH	476	LASTBRANCHTOIP	32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.
1DDH	477	LASTINTFROMIP	Last INT from IP
1DEH	478	LASTINTTOIP	Last INT to IP
200H	512	MTRRphysBase0	Memory Type Range Registers
201H	513	MTRRphysMask0	Memory Type Range Registers
202H	514	MTRRphysBase1	Memory Type Range Registers
203H	515	MTRRphysMask1	Memory Type Range Registers
204H	516	MTRRphysBase2	Memory Type Range Registers
205H	517	MTRRphysMask2	Memory Type Range Registers
206H	518	MTRRphysBase3	Memory Type Range Registers
207H	519	MTRRphysMask3	Memory Type Range Registers
208H	520	MTRRphysBase4	Memory Type Range Registers
209H	521	MTRRphysMask4	Memory Type Range Registers
20AH	522	MTRRphysBase5	Memory Type Range Registers

**Table 2-55. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
20BH	523	MTRRphysMask5	Memory Type Range Registers
20CH	524	MTRRphysBase6	Memory Type Range Registers
20DH	525	MTRRphysMask6	Memory Type Range Registers
20EH	526	MTRRphysBase7	Memory Type Range Registers
20FH	527	MTRRphysMask7	Memory Type Range Registers
250H	592	MTRRfix64K_00000	Memory Type Range Registers
258H	600	MTRRfix16K_80000	Memory Type Range Registers
259H	601	MTRRfix16K_A0000	Memory Type Range Registers
268H	616	MTRRfix4K_C0000	Memory Type Range Registers
269H	617	MTRRfix4K_C8000	Memory Type Range Registers
26AH	618	MTRRfix4K_D0000	Memory Type Range Registers
26BH	619	MTRRfix4K_D8000	Memory Type Range Registers
26CH	620	MTRRfix4K_E0000	Memory Type Range Registers
26DH	621	MTRRfix4K_E8000	Memory Type Range Registers
26EH	622	MTRRfix4K_F0000	Memory Type Range Registers
26FH	623	MTRRfix4K_F8000	Memory Type Range Registers
2FFH	767	MTRRdefType	Memory Type Range Registers
		2:0	Default memory type
		10	Fixed MTRR enable
		11	MTRR Enable
400H	1024	MCO_CTL	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
401H	1025	MCO_STATUS	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCI_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
		15:0	MC_STATUS_MCACOD
		31:16	MC_STATUS_MSCOD
		57	MC_STATUS_DAM
		58	MC_STATUS_ADDRV
		59	MC_STATUS_MISCV
		60	MC_STATUS_EN. (Note: For MCO_STATUS only, this bit is hardcoded to 1.)
		61	MC_STATUS_UC
		62	MC_STATUS_O
		63	MC_STATUS_V
402H	1026	MCO_ADDR	
403H	1027	MCO_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

Table 2-55. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
404H	1028	MC1_CTL	
405H	1029	MC1_STATUS	Bit definitions same as MCO_STATUS.
406H	1030	MC1_ADDR	
407H	1031	MC1_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
408H	1032	MC2_CTL	
409H	1033	MC2_STATUS	Bit definitions same as MCO_STATUS.
40AH	1034	MC2_ADDR	
40BH	1035	MC2_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
40CH	1036	MC4_CTL	
40DH	1037	MC4_STATUS	Bit definitions same as MCO_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.
40EH	1038	MC4_ADDR	Defined in MCA architecture but not implemented in P6 Family processors.
40FH	1039	MC4_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
410H	1040	MC3_CTL	
411H	1041	MC3_STATUS	Bit definitions same as MCO_STATUS.
412H	1042	MC3_ADDR	
413H	1043	MC3_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

#### NOTES

- 1.Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
- 2.The processor number feature may be disabled by setting bit 21 of the BBL\_CR\_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL\_CR\_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
- 3.The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

## 2.23 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5\_MC\_ADDR, P5\_MC\_TYPE, and TSC MSRs (named IA32\_P5\_MC\_ADDR, IA32\_P5\_MC\_TYPE, and IA32\_TIME\_STAMP\_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 2.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.



**Table 2-56. MSRs in the Pentium Processor**

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
1H	1	P5_MC_TYPE	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
10H	16	TSC	See Section 17.17, "Time-Stamp Counter."
11H	17	CESR	See Section 18.6.9.1, "Control and Event Select Register (CESR)."
12H	18	CTRO	Section 18.6.9.3, "Events Counted."
13H	19	CTR1	Section 18.6.9.3, "Events Counted."

## 2.24 MSR INDEX

MSRs of recent processors are indexed here for convenience. IA32 MSRs are excluded from this index.

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_ALF_ESCR0	
0FH .....	See Table 2-50
MSR_ALF_ESCR1	
0FH .....	See Table 2-50
MSR_ANY_CORE_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_ANY_GFXE_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_BO_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
MSR_BO_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_BO_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
MSR_BO_PMON_CTRO	
06_2EH .....	See Table 2-17
MSR_BO_PMON_CTR1	
06_2EH .....	See Table 2-17
MSR_BO_PMON_CTR2	
06_2EH .....	See Table 2-17
MSR_BO_PMON_CTR3	
06_2EH .....	See Table 2-17
MSR_BO_PMON_EVNT_SELO	
06_2EH .....	See Table 2-17

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_BO_PMON_EVNT_SEL1 06_2EH .....	See Table 2-17
MSR_BO_PMON_EVNT_SEL2 06_2EH .....	See Table 2-17
MSR_BO_PMON_EVNT_SEL3 06_2EH .....	See Table 2-17
MSR_BO_PMON_MASK 06_2EH .....	See Table 2-17
MSR_BO_PMON_MATCH 06_2EH .....	See Table 2-17
MSR_B1_PMON_BOX_CTRL 06_2EH .....	See Table 2-17
MSR_B1_PMON_BOX_OVF_CTRL 06_2EH .....	See Table 2-17
MSR_B1_PMON_BOX_STATUS 06_2EH .....	See Table 2-17
MSR_B1_PMON_CTRL0 06_2EH .....	See Table 2-17
MSR_B1_PMON_CTRL1 06_2EH .....	See Table 2-17
MSR_B1_PMON_CTRL2 06_2EH .....	See Table 2-17
MSR_B1_PMON_CTRL3 06_2EH .....	See Table 2-17
MSR_B1_PMON_EVNT_SELO 06_2EH .....	See Table 2-17
MSR_B1_PMON_EVNT_SEL1 06_2EH .....	See Table 2-17
MSR_B1_PMON_EVNT_SEL2 06_2EH .....	See Table 2-17
MSR_B1_PMON_EVNT_SEL3 06_2EH .....	See Table 2-17
MSR_B1_PMON_MASK 06_2EH .....	See Table 2-17
MSR_B1_PMON_MATCH 06_2EH .....	See Table 2-17
MSR_BBL_CR_CTL 06_09H .....	See Table 2-54
MSR_BBL_CR_CTL3 06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54
MSR_BIOS_DEBUG	
06_8CH, 06_8DH .....	See Table 2-45
MSR_BIOS_DONE	
06_7DH, 06_7EH .....	See Table 2-44
MSR_BIOS_MCU_ERRORCODE	
06_7DH, 06_7EH .....	See Table 2-44
06_8CH, 06_8DH .....	See Table 2-45
MSR_BPU_CCCR0	
0FH .....	See Table 2-50
MSR_BPU_CCCR1	
0FH .....	See Table 2-50
MSR_BPU_CCCR2	
0FH .....	See Table 2-50
MSR_BPU_CCCR3	
0FH .....	See Table 2-50
MSR_BPU_COUNTER0	
0FH .....	See Table 2-50
MSR_BPU_COUNTER1	
0FH .....	See Table 2-50
MSR_BPU_COUNTER2	
0FH .....	See Table 2-50
MSR_BPU_COUNTER3	
0FH .....	See Table 2-50
MSR_BPU_ESCR0	
0FH .....	See Table 2-50
MSR_BPU_ESCR1	
0FH .....	See Table 2-50
MSR_BR_DETECT_COUNTER_CONFIG_i	
06_66H.....	See Table 2-42
MSR_BR_DETECT_CTRL	
06_66H.....	See Table 2-42
MSR_BR_DETECT_STATUS	
06_66H.....	See Table 2-42
MSR_BSU_ESCR0	
0FH .....	See Table 2-50
MSR_BSU_ESCR1	
0FH .....	See Table 2-50
MSR_CO_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3FH .....	See Table 2-33
MSR_CO_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_CO_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_CO_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_CO_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_CO_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
MSR_CO_PMON_CTRL0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_CO_PMON_CTRL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_CO_PMON_CTRL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_CO_PMON_CTRL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_CO_PMON_CTRL4	
06_2EH .....	See Table 2-17
MSR_CO_PMON_CTRL5	
06_2EH .....	See Table 2-17
MSR_CO_PMON_EVNT_SELO	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_CO_PMON_EVNT_SEL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_CO_PMON_CTRL1	

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_CO_PMON_EVNT_SEL2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_CO_PMON_CTR2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_CO_PMON_EVNT_SEL3</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_CO_PMON_EVNT_SEL4</b>	
06_2EH .....	See Table 2-17
<b>MSR_CO_PMON_EVNT_SEL5</b>	
06_2EH .....	See Table 2-17
<b>MSR_C1_PMON_BOX_CTRL</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C1_PMON_BOX_FILTER</b>	
06_2DH .....	See Table 2-24
<b>MSR_C1_PMON_BOX_FILTER0</b>	
06_3FH .....	See Table 2-33
<b>MSR_C1_PMON_BOX_FILTER1</b>	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C1_PMON_BOX_OVF_CTRL</b>	
06_2EH .....	See Table 2-17
<b>MSR_C1_PMON_BOX_STATUS</b>	
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
<b>MSR_C1_PMON_CTR0</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C1_PMON_CTR1</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3FH .....	See Table 2-33
MSR_C1_PMON_CTR2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C1_PMON_CTR3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C1_PMON_CTR4	
06_2EH .....	See Table 2-17
MSR_C1_PMON_CTR5	
06_2EH .....	See Table 2-17
MSR_C1_PMON_EVNT_SELO	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C1_PMON_EVNT_SEL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C1_PMON_EVNT_SEL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C1_PMON_EVNT_SEL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C1_PMON_EVNT_SEL4	
06_2EH .....	See Table 2-17
MSR_C1_PMON_EVNT_SEL5	
06_2EH .....	See Table 2-17
MSR_C10_PMON_BOX_FILTER	
06_3EH .....	See Table 2-28
MSR_C10_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C10_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C11_PMON_BOX_FILTER	
06_3EH .....	See Table 2-28

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C11_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C11_PMON_BOX_FILTER1 06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C12_PMON_BOX_FILTER 06_3EH .....	See Table 2-28
MSR_C12_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C12_PMON_BOX_FILTER1 06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C13_PMON_BOX_FILTER 06_3EH .....	See Table 2-28
MSR_C13_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C13_PMON_BOX_FILTER1 06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C14_PMON_BOX_FILTER 06_3EH .....	See Table 2-28
MSR_C14_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C14_PMON_BOX_FILTER1 06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C15_PMON_BOX_CTL 06_3FH .....	See Table 2-33
MSR_C15_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C15_PMON_BOX_FILTER1 06_3FH .....	See Table 2-33
MSR_C15_PMON_BOX_STATUS 06_3FH .....	See Table 2-33
MSR_C15_PMON_CTR0 06_3FH .....	See Table 2-33
MSR_C15_PMON_CTR1 06_3FH .....	See Table 2-33
MSR_C15_PMON_CTR2 06_3FH .....	See Table 2-33
MSR_C15_PMON_CTR3 06_3FH .....	See Table 2-33

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C15_PMON_EVNTSELO 06_3FH .....	See Table 2-33
MSR_C15_PMON_EVNTSEL1 06_3FH .....	See Table 2-33
MSR_C15_PMON_EVNTSEL2 06_3FH .....	See Table 2-33
MSR_C15_PMON_EVNTSEL3 06_3FH .....	See Table 2-33
MSR_C16_PMON_BOX_CTL 06_3FH .....	See Table 2-33
MSR_C16_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C16_PMON_BOX_FILTER1 06_3FH .....	See Table 2-33
MSR_C16_PMON_BOX_STATUS 06_3FH .....	See Table 2-33
MSR_C16_PMON_CTRL0 06_3FH .....	See Table 2-33
MSR_C16_PMON_CTRL3 06_3FH .....	See Table 2-33
MSR_C16_PMON_CTRL2 06_3FH .....	See Table 2-33
MSR_C16_PMON_CTRL3 06_3FH .....	See Table 2-33
MSR_C16_PMON_EVNTSELO 06_3FH .....	See Table 2-33
MSR_C16_PMON_EVNTSEL1 06_3FH .....	See Table 2-33
MSR_C16_PMON_EVNTSEL2 06_3FH .....	See Table 2-33
MSR_C16_PMON_EVNTSEL3 06_3FH .....	See Table 2-33
MSR_C17_PMON_BOX_CTL 06_3FH .....	See Table 2-33
MSR_C17_PMON_BOX_FILTER0 06_3FH .....	See Table 2-33
MSR_C17_PMON_BOX_FILTER1 06_3FH .....	See Table 2-33
MSR_C17_PMON_BOX_STATUS 06_3FH .....	See Table 2-33
MSR_C17_PMON_CTRL0 06_3FH .....	See Table 2-33



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_C17_PMON_CTR1	
06_3FH .....	See Table 2-33
MSR_C17_PMON_CTR2	
06_3FH .....	See Table 2-33
MSR_C17_PMON_CTR3	
06_3FH .....	See Table 2-33
MSR_C17_PMON_EVNTSELO	
06_3FH .....	See Table 2-33
MSR_C17_PMON_EVNTSEL1	
06_3FH .....	See Table 2-33
MSR_C17_PMON_EVNTSEL2	
06_3FH .....	See Table 2-33
MSR_C17_PMON_EVNTSEL3	
06_3FH .....	See Table 2-33
MSR_C2_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_C2_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C2_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C2_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_C2_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
MSR_C2_PMON_CTR0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_CTR1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_CTR2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C2_PMON_CTRL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_CTRL4	
06_2EH .....	See Table 2-17
MSR_C2_PMON_CTRL5	
06_2EH .....	See Table 2-17
MSR_C2_PMON_EVTN_SEL0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_EVTN_SEL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_EVTN_SEL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_EVTN_SEL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C2_PMON_EVTN_SEL4	
06_2EH .....	See Table 2-17
MSR_C2_PMON_EVTN_SEL5	
06_2EH .....	See Table 2-17
MSR_C3_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C3_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_C3_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C3_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C3_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_C3_PMON_BOX_STATUS	

MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_CTRL0</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_CTRL1</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_CTRL2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_CTRL3</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_CTRL4</b>	
06_2EH .....	See Table 2-17
<b>MSR_C3_PMON_CTRL5</b>	
06_2EH .....	See Table 2-17
<b>MSR_C3_PMON_EVTN_SEL0</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_EVTN_SEL1</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_EVTN_SEL2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_EVTN_SEL3</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C3_PMON_EVTN_SEL4</b>	
06_2EH .....	See Table 2-17
<b>MSR_C3_PMON_EVTN_SEL5</b>	
06_2EH .....	See Table 2-17

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C4_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_C4_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C4_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C4_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_C4_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
MSR_C4_PMON_CTRL0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_CTRL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_CTRL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_CTRL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_CTRL4	
06_2EH .....	See Table 2-17
MSR_C4_PMON_CTRL5	
06_2EH .....	See Table 2-17
MSR_C4_PMON_EVNT_SELO	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_EVNT_SEL1	
06_2EH .....	See Table 2-17

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_EVNT_SEL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_EVNT_SEL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C4_PMON_EVNT_SEL4	
06_2EH .....	See Table 2-17
MSR_C4_PMON_EVNT_SEL5	
06_2EH .....	See Table 2-17
MSR_C5_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_C5_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C5_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C5_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_C5_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
MSR_C5_PMON_CTRL0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_CTRL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_CTRL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C5_PMON_CTRL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_CTRL4	
06_2EH .....	See Table 2-17
MSR_C5_PMON_CTRL5	
06_2EH .....	See Table 2-17
MSR_C5_PMON_EVNT_SEL0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_EVNT_SEL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_EVNT_SEL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_EVNT_SEL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C5_PMON_EVNT_SEL4	
06_2EH .....	See Table 2-17
MSR_C5_PMON_EVNT_SEL5	
06_2EH .....	See Table 2-17
MSR_C6_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C6_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_C6_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C6_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C6_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_C6_PMON_BOX_STATUS	

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_CTRL0</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_CTRL1</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_CTRL2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_CTRL3</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_CTRL4</b>	
06_2EH .....	See Table 2-17
<b>MSR_C6_PMON_CTRL5</b>	
06_2EH .....	See Table 2-17
<b>MSR_C6_PMON_EVTN_SEL0</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_EVTN_SEL1</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_EVTN_SEL2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_EVTN_SEL3</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C6_PMON_EVTN_SEL4</b>	
06_2EH .....	See Table 2-17
<b>MSR_C6_PMON_EVTN_SEL5</b>	
06_2EH .....	See Table 2-17

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C7_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C7_PMON_BOX_FILTER	
06_2DH .....	See Table 2-24
MSR_C7_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C7_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C7_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_C7_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
06_3FH .....	See Table 2-33
MSR_C7_PMON_CTRL0	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C7_PMON_CTRL1	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C7_PMON_CTRL2	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C7_PMON_CTRL3	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C7_PMON_CTRL4	
06_2EH .....	See Table 2-17
MSR_C7_PMON_CTRL5	
06_2EH .....	See Table 2-17
MSR_C7_PMON_EVNT_SELO	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
MSR_C7_PMON_EVNT_SEL1	
06_2EH .....	See Table 2-17



MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C7_PMON_EVNT_SEL2</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C7_PMON_EVNT_SEL3</b>	
06_2EH .....	See Table 2-17
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_C7_PMON_EVNT_SEL4</b>	
06_2EH .....	See Table 2-17
<b>MSR_C7_PMON_EVNT_SEL5</b>	
06_2EH .....	See Table 2-17
<b>MSR_C8_PMON_BOX_CTRL</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C8_PMON_BOX_FILTER</b>	
06_3EH .....	See Table 2-28
<b>MSR_C8_PMON_BOX_FILTER0</b>	
06_3FH .....	See Table 2-33
<b>MSR_C8_PMON_BOX_FILTER1</b>	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C8_PMON_BOX_OVF_CTRL</b>	
06_2FH .....	See Table 2-19
<b>MSR_C8_PMON_BOX_STATUS</b>	
06_2FH .....	See Table 2-19
06_3FH .....	See Table 2-33
<b>MSR_C8_PMON_CTR0</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C8_PMON_CTR1</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C8_PMON_CTR2</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C8_PMON_CTR3	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C8_PMON_CTR4	
06_2FH .....	See Table 2-19
MSR_C8_PMON_CTR5	
06_2FH .....	See Table 2-19
MSR_C8_PMON_EVNT_SELO	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C8_PMON_EVNT_SEL1	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C8_PMON_EVNT_SEL2	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C8_PMON_EVNT_SEL3	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C8_PMON_EVNT_SEL4	
06_2FH .....	See Table 2-19
MSR_C8_PMON_EVNT_SEL5	
06_2FH .....	See Table 2-19
MSR_C9_PMON_BOX_CTRL	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C9_PMON_BOX_FILTER	
06_3EH .....	See Table 2-28
MSR_C9_PMON_BOX_FILTER0	
06_3FH .....	See Table 2-33
MSR_C9_PMON_BOX_FILTER1	
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
MSR_C9_PMON_BOX_OVF_CTRL	
06_2FH .....	See Table 2-19
MSR_C9_PMON_BOX_STATUS	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2FH .....	See Table 2-19
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_CTRL0</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_CTRL1</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_CTRL2</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_CTRL3</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_CTRL4</b>	
06_2FH .....	See Table 2-19
<b>MSR_C9_PMON_CTRL5</b>	
06_2FH .....	See Table 2-19
<b>MSR_C9_PMON_EVTN_SEL0</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_EVTN_SEL1</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_EVTN_SEL2</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_EVTN_SEL3</b>	
06_2FH .....	See Table 2-19
06_3EH .....	See Table 2-28
06_3FH .....	See Table 2-33
<b>MSR_C9_PMON_EVTN_SEL4</b>	
06_2FH .....	See Table 2-19
<b>MSR_C9_PMON_EVTN_SEL5</b>	
06_2FH .....	See Table 2-19

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_CC6_DEMOTION_POLICY_CONFIG	
06_37H .....	See Table 2-9
MSR_CONFIG_TDP_CONTROL	
06_3AH .....	See Table 2-25
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_57H, 06_85H .....	See Table 2-48
MSR_CONFIG_TDP_LEVEL1	
06_3AH .....	See Table 2-25
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_57H, 06_85H .....	See Table 2-48
MSR_CONFIG_TDP_LEVEL2	
06_3AH .....	See Table 2-25
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_57H, 06_85H .....	See Table 2-48
MSR_CONFIG_TDP_NOMINAL	
06_3AH .....	See Table 2-25
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_57H, 06_85H .....	See Table 2-48
MSR_CORE_C1_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_66H .....	See Table 2-42
MSR_CORE_C3_RESIDENCY	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
MSR_CORE_C6_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
06_57H, 06_85H .....	See Table 2-48
MSR_CORE_C7_RESIDENCY	
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
MSR_CORE_GFXE_OVERLAP_CO	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_CORE_HDC_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_CORE_PERF_LIMIT_REASONS	
06_5CH, 06_7AH .....	See Table 2-12
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_3F .....	See Table 2-32

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_56H, 06_4FH .....	See Table 2-36
06_57H, 06_85H.....	See Table 2-48
MSR_CORE_THREAD_COUNT	
06_3FH.....	See Table 2-32
MSR_CRASHLOG_CONTROL	
06_7DH, 06_7EH .....	See Table 2-44
MSR_CRU_ESCR0	
0FH.....	See Table 2-50
MSR_CRU_ESCR1	
0FH.....	See Table 2-50
MSR_CRU_ESCR2	
0FH.....	See Table 2-50
MSR_CRU_ESCR3	
0FH.....	See Table 2-50
MSR_CRU_ESCR4	
0FH.....	See Table 2-50
MSR_CRU_ESCR5	
0FH.....	See Table 2-50
MSR_DAC_ESCR0	
0FH.....	See Table 2-50
MSR_DAC_ESCR1	
0FH.....	See Table 2-50
MSR_DRAM_ENERGY_STATUS	
06_5CH, 06_7AH .....	See Table 2-12
06_2DH.....	See Table 2-23
06_3EH, 06_3FH .....	See Table 2-26
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_3F .....	See Table 2-32
06_56H, 06_4FH .....	See Table 2-36
06_6AH, 06_6CH .....	See Table 2-47
06_57H, 06_85H.....	See Table 2-48
MSR_DRAM_PERF_STATUS	
06_5CH, 06_7AH .....	See Table 2-12
06_2DH.....	See Table 2-23
06_3EH, 06_3FH.....	See Table 2-26
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_3F .....	See Table 2-32
06_56H, 06_4FH .....	See Table 2-36
06_6AH, 06_6CH .....	See Table 2-47
06_57H, 06_85H.....	See Table 2-48
MSR_DRAM_POWER_INFO	
06_5CH, 06_7AH .....	See Table 2-12

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2DH .....	See Table 2-23
06_3EH, 06_3FH .....	See Table 2-26
06_3F .....	See Table 2-32
06_56H, 06_4FH .....	See Table 2-36
06_6AH, 06_6CH .....	See Table 2-47
06_57H, 06_85H .....	See Table 2-48
<b>MSR_DRAM_POWER_LIMIT</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_2DH .....	See Table 2-23
06_3EH, 06_3FH .....	See Table 2-26
06_3F .....	See Table 2-32
06_56H, 06_4FH .....	See Table 2-36
06_6AH, 06_6CH .....	See Table 2-47
06_57H, 06_85H .....	See Table 2-48
<b>MSR_EBC_FREQUENCY_ID</b>	
0FH .....	See Table 2-50
<b>MSR_EBC_HARD_POWERON</b>	
0FH .....	See Table 2-50
<b>MSR_EBC_SOFT_POWERON</b>	
0FH .....	See Table 2-50
<b>MSR_EBL_CR_POWERON</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54
<b>MSR_EFSB_DRDY0</b>	
0F_03H, 0F_04H .....	See Table 2-51
<b>MSR_EFSB_DRDY1</b>	
0F_03H, 0F_04H .....	See Table 2-51
<b>MSR_EMON_L3_CTR_CTL0</b>	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
<b>MSR_EMON_L3_CTR_CTL1</b>	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
<b>MSR_EMON_L3_CTR_CTL2</b>	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
<b>MSR_EMON_L3_CTR_CTL3</b>	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_EMON_L3_CTR_CTL4	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
MSR_EMON_L3_CTR_CTL5	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
MSR_EMON_L3_CTR_CTL6	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
MSR_EMON_L3_CTR_CTL7	
06_0FH, 06_17H .....	See Table 2-3
0F_06H .....	See Table 2-52
MSR_EMON_L3_GL_CTL	
06_0FH, 06_17H .....	See Table 2-3
MSR_ERROR_CONTROL	
06_2DH .....	See Table 2-23
06_3EH .....	See Table 2-26
06_3F .....	See Table 2-32
MSR_FAST_UNCORE_MSRS_CAPABILITY	
06_7DH, 06_7EH .....	See Table 2-44
MSR_FAST_UNCORE_MSRS_CTL	
06_7DH, 06_7EH .....	See Table 2-44
MSR_FAST_UNCORE_MSRS_STATUS	
06_7DH, 06_7EH .....	See Table 2-44
MSR_FEATURE_CONFIG	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_25H, 06_2CH .....	See Table 2-18
06_2FH .....	See Table 2-19
06_2AH, 06_2DH .....	See Table 2-20
06_57H, 06_85H .....	See Table 2-48
MSR_FIRM_ESCR0	
0FH .....	See Table 2-50
MSR_FIRM_ESCR1	
0FH .....	See Table 2-50
MSR_FLAME_CCCR0	
0FH .....	See Table 2-50
MSR_FLAME_CCCR1	
0FH .....	See Table 2-50
MSR_FLAME_CCCR2	
0FH .....	See Table 2-50
MSR_FLAME_CCCR3	
0FH .....	See Table 2-50

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_FLAME_COUNTER0 OFH .....	See Table 2-50
MSR_FLAME_COUNTER1 OFH .....	See Table 2-50
MSR_FLAME_COUNTER2 OFH .....	See Table 2-50
MSR_FLAME_COUNTER3 OFH .....	See Table 2-50
MSR_FLAME_ESCRO OFH .....	See Table 2-50
MSR_FLAME_ESCR1 OFH .....	See Table 2-50
MSR_FSB_ESCRO OFH .....	See Table 2-50
MSR_FSB_ESCR1 OFH .....	See Table 2-50
MSR_FSB_FREQ 06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_4CH .....	See Table 2-11
06_0EH .....	See Table 2-53
MSR_GQ_SNOOP_MESF 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_GRAPHICS_PERF_LIMIT_REASONS 06_3CH, 06_45H, 06_46H .....	See Table 2-30
MSR_IFSB_BUSQ0 OF_03H, OF_04H .....	See Table 2-51
MSR_IFSB_BUSQ1 OF_03H, OF_04H .....	See Table 2-51
MSR_IFSB_CNTR7 OF_03H, OF_04H .....	See Table 2-51
MSR_IFSB_CTL6 OF_03H, OF_04H .....	See Table 2-51
MSR_IFSB_SNPQ0 OF_03H, OF_04H .....	See Table 2-51
MSR_IFSB_SNPQ1 OF_03H, OF_04H .....	See Table 2-51
MSR_IQ_CCCR0 OFH .....	See Table 2-50
MSR_IQ_CCCR1 OFH .....	See Table 2-50



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_IQ_CCCR2	
OFH .....	See Table 2-50
MSR_IQ_CCCR3	
OFH .....	See Table 2-50
MSR_IQ_CCCR4	
OFH .....	See Table 2-50
MSR_IQ_CCCR5	
OFH .....	See Table 2-50
MSR_IQ_COUNTER0	
OFH .....	See Table 2-50
MSR_IQ_COUNTER1	
OFH .....	See Table 2-50
MSR_IQ_COUNTER2	
OFH .....	See Table 2-50
MSR_IQ_COUNTER3	
OFH .....	See Table 2-50
MSR_IQ_COUNTER4	
OFH .....	See Table 2-50
MSR_IQ_COUNTER5	
OFH .....	See Table 2-50
MSR_IQ_ESCR0	
OFH .....	See Table 2-50
MSR_IQ_ESCR1	
OFH .....	See Table 2-50
MSR_IS_ESCR0	
OFH .....	See Table 2-50
MSR_IS_ESCR1	
OFH .....	See Table 2-50
MSR_ITLB_ESCR0	
OFH .....	See Table 2-50
MSR_ITLB_ESCR1	
OFH .....	See Table 2-50
MSR_IX_ESCR0	
OFH .....	See Table 2-50
MSR_IX_ESCR1	
OFH .....	See Table 2-50
MSR_LASTBRANCH_0	
OFH .....	See Table 2-50
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54
MSR_LASTBRANCH_0_FROM_IP	
06_OFH, 06_17H .....	See Table 2-3

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH.....	See Table 2-12
06_7AH.....	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_0_TO_IP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH.....	See Table 2-12
06_7AH.....	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_1_FROM_IP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_1_TO_IP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_10_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_10_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50

MSR Name and CPUID DisplayFamily_DisplayModel	Location
<b>MSR_LASTBRANCH_11_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_11_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_12_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_12_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_13_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_13_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_14_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_14_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_15_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_15_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_16_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_16_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_17_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_17_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_18_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_18_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_19_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_19_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_2</b>	
0FH .....	See Table 2-50
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54

MSR Name and CPUID DisplayFamily_DisplayModel	Location
<b>MSR_LASTBRANCH_2_FROM_IP</b>	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 2-7
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
0FH.....	See Table 2-50
<b>MSR_LASTBRANCH_2_TO_IP</b>	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 2-7
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
0FH.....	See Table 2-50
<b>MSR_LASTBRANCH_20_FROM_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_LASTBRANCH_20_TO_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_LASTBRANCH_21_FROM_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_LASTBRANCH_21_TO_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_LASTBRANCH_22_FROM_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_LASTBRANCH_22_TO_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_LASTBRANCH_23_FROM_IP</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LASTBRANCH_23_TO_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_24_FROM_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_24_TO_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_25_FROM_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_25_TO_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_26_FROM_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_26_TO_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_27_FROM_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_27_TO_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_28_FROM_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_28_TO_IP	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_LASTBRANCH_29_FROM_IP	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_29_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_3</b>	
0FH .....	See Table 2-50
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54
<b>MSR_LASTBRANCH_3_FROM_IP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_3_TO_IP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_30_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_30_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_31_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_LASTBRANCH_31_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_LASTBRANCH_4</b>	
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
<b>MSR_LASTBRANCH_4_FROM_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_4_TO_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_5</b>	
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
<b>MSR_LASTBRANCH_5_FROM_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_5_TO_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_6</b>	
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
<b>MSR_LASTBRANCH_6_FROM_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15



## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_6_TO_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_7</b>	
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54
<b>MSR_LASTBRANCH_7_FROM_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_7_TO_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_8_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_8_TO_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_9_FROM_IP</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_9_TO_IP</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
0FH .....	See Table 2-50
<b>MSR_LASTBRANCH_TOS</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_57H, 06_85H .....	See Table 2-48
06_0EH .....	See Table 2-53
06_09H .....	See Table 2-54
<b>MSR_LASTBRANCH_INFO_0</b>	
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_1</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_10</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_11</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_12</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_13</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_14</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH .....	See Table 2-13
<b>MSR_LBR_INFO_15</b>	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_16	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_17	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_18	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_19	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_2	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_20	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_21	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_22	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_23	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_24	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_25	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_26</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_27</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_28</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_29</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_3</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_30</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_31</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_4</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_5</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_6</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_7</b>	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_8</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_INFO_9</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
06_7AH.....	See Table 2-13
<b>MSR_LBR_SELECT</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_57H, 06_85H.....	See Table 2-48
<b>MSR_LER_FROM_LIP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
0FH.....	See Table 2-50
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
<b>MSR_LER_TO_LIP</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
0FH.....	See Table 2-50
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
<b>MSR_MO_PMON_ADDR_MASK</b>	
06_2EH.....	See Table 2-17
<b>MSR_MO_PMON_ADDR_MATCH</b>	
06_2EH.....	See Table 2-17
<b>MSR_MO_PMON_BOX_CTRL</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 2-17
MSR_MO_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_MO_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTRL0	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTRL1	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTRL2	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTRL3	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTRL4	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTRL5	
06_2EH.....	See Table 2-17
MSR_MO_PMON_DSP	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SELO	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SEL1	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SEL2	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SEL3	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SEL4	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SEL5	
06_2EH.....	See Table 2-17
MSR_MO_PMON_ISS	
06_2EH.....	See Table 2-17
MSR_MO_PMON_MAP	
06_2EH.....	See Table 2-17
MSR_MO_PMON_MM_CONFIG	
06_2EH.....	See Table 2-17
MSR_MO_PMON_MSC_THR	
06_2EH.....	See Table 2-17
MSR_MO_PMON_PGT	
06_2EH.....	See Table 2-17
MSR_MO_PMON_PLD	

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 2-17
MSR_MO_PMON_TIMESTAMP	
06_2EH.....	See Table 2-17
MSR_MO_PMON_ZDP	
06_2EH.....	See Table 2-17
MSR_M1_PMON_ADDR_MASK	
06_2EH.....	See Table 2-17
MSR_M1_PMON_ADDR_MATCH	
06_2EH.....	See Table 2-17
MSR_M1_PMON_BOX_CTRL	
06_2EH.....	See Table 2-17
MSR_M1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_M1_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_M1_PMON_CTRL0	
06_2EH.....	See Table 2-17
MSR_M1_PMON_CTRL1	
06_2EH.....	See Table 2-17
MSR_M1_PMON_CTRL2	
06_2EH.....	See Table 2-17
MSR_M1_PMON_CTRL3	
06_2EH.....	See Table 2-17
MSR_M1_PMON_CTRL4	
06_2EH.....	See Table 2-17
MSR_M1_PMON_CTRL5	
06_2EH.....	See Table 2-17
MSR_M1_PMON_DSP	
06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL0	
06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL1	
06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL2	
06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL3	
06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL4	
06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL5	
06_2EH.....	See Table 2-17
MSR_M1_PMON_ISS	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 2-17
MSR_M1_PMON_MAP	
06_2EH.....	See Table 2-17
MSR_M1_PMON_MM_CONFIG	
06_2EH.....	See Table 2-17
MSR_M1_PMON_MSC_THR	
06_2EH.....	See Table 2-17
MSR_M1_PMON_PGT	
06_2EH.....	See Table 2-17
MSR_M1_PMON_PLD	
06_2EH.....	See Table 2-17
MSR_M1_PMON_TIMESTAMP	
06_2EH.....	See Table 2-17
MSR_M1_PMON_ZDP	
06_2EH.....	See Table 2-17
IA32_MCO_MISC / MSR_MCO_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
MSR_MCO_RESIDENCY	
06_57H, 06_85H.....	See Table 2-48
IA32_MC1_MISC / MSR_MC1_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
IA32_MC10_ADDR / MSR_MC10_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC10_CTL / MSR_MC10_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC10_MISC / MSR_MC10_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38



<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>IA32_MC10_STATUS / MSR_MC10_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC11_ADDR / MSR_MC11_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC11_CTL / MSR_MC11_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC11_MISC / MSR_MC11_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC11_STATUS / MSR_MC11_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC12_ADDR / MSR_MC12_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC12_CTL / MSR_MC12_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_4FH.....	See Table 2-38
<b>IA32_MC12_MISC / MSR_MC12_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC12_STATUS / MSR_MC12_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC13_ADDR / MSR_MC13_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC13_CTL / MSR_MC13_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC13_MISC / MSR_MC13_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC13_STATUS / MSR_MC13_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC14_ADDR / MSR_MC14_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_4FH.....	See Table 2-38
<b>IA32_MC14_CTL / MSR_MC14_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC14_MISC / MSR_MC14_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC14_STATUS / MSR_MC14_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC15_ADDR / MSR_MC15_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC15_CTL / MSR_MC15_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC15_MISC / MSR_MC15_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC15_STATUS / MSR_MC15_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_4FH.....	See Table 2-38
<b>IA32_MC16_ADDR / MSR_MC16_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC16_CTL / MSR_MC16_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC16_MISC / MSR_MC16_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC16_STATUS / MSR_MC16_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC17_ADDR / MSR_MC17_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC17_CTL / MSR_MC17_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC17_MISC / MSR_MC17_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC17_STATUS / MSR_MC17_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC18_ADDR / MSR_MC18_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC18_CTL / MSR_MC18_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC18_MISC / MSR_MC18_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC18_STATUS / MSR_MC18_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC19_ADDR / MSR_MC19_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC19_CTL / MSR_MC19_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC19_MISC / MSR_MC19_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC19_STATUS / MSR_MC19_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC2_MISC / MSR_MC2_MISC</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
<b>IA32_MC20_ADDR / MSR_MC20_ADDR</b>	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC20_CTL / MSR_MC20_CTL</b>	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC20_MISC / MSR_MC20_MISC</b>	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
IA32_MC20_STATUS / MSR_MC20_STATUS	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC21_ADDR / MSR_MC21_ADDR	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC21_CTL / MSR_MC21_CTL	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC21_MISC / MSR_MC21_MISC	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC21_STATUS / MSR_MC21_STATUS	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC22_ADDR / MSR_MC22_ADDR	
06_3EH.....	See Table 2-26
IA32_MC22_CTL / MSR_MC22_CTL	
06_3EH.....	See Table 2-26
IA32_MC22_MISC / MSR_MC22_MISC	
06_3EH.....	See Table 2-26
IA32_MC22_STATUS / MSR_MC22_STATUS	
06_3EH.....	See Table 2-26
IA32_MC23_ADDR / MSR_MC23_ADDR	
06_3EH.....	See Table 2-26
IA32_MC23_CTL / MSR_MC23_CTL	
06_3EH.....	See Table 2-26
IA32_MC23_MISC / MSR_MC23_MISC	
06_3EH.....	See Table 2-26
IA32_MC23_STATUS / MSR_MC23_STATUS	
06_3EH.....	See Table 2-26
IA32_MC24_ADDR / MSR_MC24_ADDR	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 2-26
IA32_MC24_CTL / MSR_MC24_CTL	
06_3EH.....	See Table 2-26
IA32_MC24_MISC / MSR_MC24_MISC	
06_3EH.....	See Table 2-26
IA32_MC24_STATUS / MSR_MC24_STATUS	
06_3EH.....	See Table 2-26
IA32_MC25_ADDR / MSR_MC25_ADDR	
06_3EH.....	See Table 2-26
IA32_MC25_CTL / MSR_MC25_CTL	
06_3EH.....	See Table 2-26
IA32_MC25_MISC / MSR_MC25_MISC	
06_3EH.....	See Table 2-26
IA32_MC25_STATUS / MSR_MC25_STATUS	
06_3EH.....	See Table 2-26
IA32_MC26_ADDR / MSR_MC26_ADDR	
06_3EH.....	See Table 2-26
IA32_MC26_CTL / MSR_MC26_CTL	
06_3EH.....	See Table 2-26
IA32_MC26_MISC / MSR_MC26_MISC	
06_3EH.....	See Table 2-26
IA32_MC26_STATUS / MSR_MC26_STATUS	
06_3EH.....	See Table 2-26
IA32_MC27_ADDR / MSR_MC27_ADDR	
06_3EH.....	See Table 2-26
IA32_MC27_CTL / MSR_MC27_CTL	
06_3EH.....	See Table 2-26
IA32_MC27_MISC / MSR_MC27_MISC	
06_3EH.....	See Table 2-26
IA32_MC27_STATUS / MSR_MC27_STATUS	
06_3EH.....	See Table 2-26
IA32_MC28_ADDR / MSR_MC28_ADDR	
06_3EH.....	See Table 2-26
IA32_MC28_CTL / MSR_MC28_CTL	
06_3EH.....	See Table 2-26
IA32_MC28_MISC / MSR_MC28_MISC	
06_3EH.....	See Table 2-26
IA32_MC28_STATUS / MSR_MC28_STATUS	
06_3EH.....	See Table 2-26
IA32_MC29_ADDR / MSR_MC29_ADDR	
06_3EH.....	See Table 2-27
IA32_MC29_CTL / MSR_MC29_CTL	



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH.....	See Table 2-27
IA32_MC29_MISC / MSR_MC29_MISC	
06_3EH.....	See Table 2-27
IA32_MC29_STATUS / MSR_MC29_STATUS	
06_3EH.....	See Table 2-27
IA32_MC3_ADDR / MSR_MC3_ADDR	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
IA32_MC3_CTL / MSR_MC3_CTL	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
IA32_MC3_MISC / MSR_MC3_MISC	
06_0FH, 06_17H .....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_0EH.....	See Table 2-53
IA32_MC3_STATUS / MSR_MC3_STATUS	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
IA32_MC30_ADDR / MSR_MC30_ADDR	
06_3EH.....	See Table 2-27
IA32_MC30_CTL / MSR_MC30_CTL	
06_3EH.....	See Table 2-27
IA32_MC30_MISC / MSR_MC30_MISC	
06_3EH.....	See Table 2-27
IA32_MC30_STATUS / MSR_MC30_STATUS	
06_3EH.....	See Table 2-27
IA32_MC31_ADDR / MSR_MC31_ADDR	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 2-27
IA32_MC31_CTL / MSR_MC31_CTL	
06_3EH.....	See Table 2-27
IA32_MC31_MISC / MSR_MC31_MISC	
06_3EH.....	See Table 2-27
IA32_MC31_STATUS / MSR_MC31_STATUS	
06_3EH.....	See Table 2-27
IA32_MC4_ADDR / MSR_MC4_ADDR	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
IA32_MC4_CTL / MSR_MC4_CTL	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
IA32_MC4_CTL2 / MSR_MC4_CTL2	
06_2AH, 06_2DH.....	See Table 2-20
IA32_MC4_STATUS / MSR_MC4_STATUS	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
MSR_MC5_ADDR / MSR_MC5_ADDR	
06_0FH, 06_17H.....	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_57H, 06_85H.....	See Table 2-48

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_0EH.....	See Table 2-53
<b>IA32_MC5_CTL / MSR_MC5_CTL</b>	
06_0FH, 06_17H.....	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
<b>IA32_MC5_MISC / MSR_MC5_MISC</b>	
06_0FH, 06_17H.....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_0EH.....	See Table 2-53
<b>IA32_MC5_STATUS / MSR_MC5_STATUS</b>	
06_0FH, 06_17H.....	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_57H, 06_85H.....	See Table 2-48
06_0EH.....	See Table 2-53
<b>IA32_MC6_ADDR / MSR_MC6_ADDR</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC6_CTL / MSR_MC6_CTL</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_4FH.....	See Table 2-38
<b>MSR_MC6_DEMOTION_POLICY_CONFIG</b>	
06_37H.....	See Table 2-9
<b>IA32_MC6_MISC / MSR_MC6_MISC</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>MSR_MC6_RESIDENCY_COUNTER</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 2-7
06_37H.....	See Table 2-9
06_57H, 06_85H.....	See Table 2-48
<b>IA32_CORE_CAPABILITIES (Note there are no architecturally defined bits; all bits are model-specific)</b>	
06_86H.....	See Table 2-14
06_8CH, 06_8DH.....	See Table 2-45
<b>IA32_MC6_STATUS / MSR_MC6_STATUS</b>	
06_0FH, 06_17H.....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC7_ADDR / MSR_MC7_ADDR</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC7_CTL / MSR_MC7_CTL</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC7_MISC / MSR_MC7_MISC</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC7_STATUS / MSR_MC7_STATUS</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC8_ADDR / MSR_MC8_ADDR</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC8_CTL / MSR_MC8_CTL</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC8_MISC / MSR_MC8_MISC</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC8_STATUS / MSR_MC8_STATUS</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
<b>IA32_MC9_ADDR / MSR_MC9_ADDR</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>IA32_MC9_CTL / MSR_MC9_CTL</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC9_MISC / MSR_MC9_MISC</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>IA32_MC9_STATUS / MSR_MC9_STATUS</b>	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>MSR_MCG_MISC</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R10</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R11</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R12</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R13</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R14</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R15</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R8</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_R9</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_RAX</b>	
0FH.....	See Table 2-50
<b>MSR_MCG_RBP</b>	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 2-50
MSR_MCG_RBX	
0FH.....	See Table 2-50
MSR_MCG_RCX	
0FH.....	See Table 2-50
MSR_MCG_RDI	
0FH.....	See Table 2-50
MSR_MCG_RDX	
0FH.....	See Table 2-50
MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5	
0FH.....	See Table 2-50
MSR_MCG_RFLAGS	
0FH.....	See Table 2-50
MSR_MCG_RIP	
0FH.....	See Table 2-50
MSR_MCG_RSI	
0FH.....	See Table 2-50
MSR_MCG_RSP	
0FH.....	See Table 2-50
MSR_MISC_FEATURE_CONTROL	
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
MSR_MISC_PWR_MGMT	
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
MSR_MOB_ESCRO	
0FH.....	See Table 2-50
MSR_MOB_ESCR1	
0FH.....	See Table 2-50
MSR_MS_CCCRO	
0FH.....	See Table 2-50
MSR_MS_CCCR1	
0FH.....	See Table 2-50
MSR_MS_CCCR2	
0FH.....	See Table 2-50
MSR_MS_CCCR3	
0FH.....	See Table 2-50
MSR_MS_COUNTER0	
0FH.....	See Table 2-50
MSR_MS_COUNTER1	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
0FH.....	See Table 2-50
<b>MSR_MS_COUNTER2</b>	
0FH.....	See Table 2-50
<b>MSR_MS_COUNTER3</b>	
0FH.....	See Table 2-50
<b>MSR_MS_ESCRO</b>	
0FH.....	See Table 2-50
<b>MSR_MS_ESCR1</b>	
0FH.....	See Table 2-50
<b>MSR_MTRRCAP</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH....	See Table 2-39
<b>MSR_OFFCORE_RSP_0</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
<b>MSR_OFFCORE_RSP_1</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_25H, 06_2CH.....	See Table 2-18
06_2FH.....	See Table 2-19
06_2AH, 06_2DH.....	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
<b>MSR_PACKAGE_ENERGY_TIME_STATUS</b>	
06_6AH, 06_6CH.....	See Table 2-47
<b>MSR_PCIE_PLL_RATIO</b>	
06_3FH.....	See Table 2-32
<b>MSR_PCU_PMON_BOX_CTL</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_PCU_PMON_BOX_FILTER</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_PCU_PMON_BOX_STATUS</b>	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
<b>MSR_PCU_PMON_CTR0</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_PCU_PMON_CTR1</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_PCU_PMON_CTR2</b>	



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_CTR3	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSELO	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSEL1	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSEL2	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSEL3	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PEBS_DATA_CFG	
06_7DH, 06_7EH.....	See Table 2-44
MSR_PEBS_ENABLE	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 2-7
06_5CH.....	See Table 2-12
06_7AH.....	See Table 2-13
06_86H.....	See Table 2-14
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3EH.....	See Table 2-27
06_57H, 06_85H.....	See Table 2-48
0FH.....	See Table 2-50
MSR_PEBS_FRONTEND	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH....	See Table 2-39
MSR_PEBS_LD_LAT	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
MSR_PEBS_MATRIX_VERT	
0FH.....	See Table 2-50
MSR_PEBS_NUM_ALT	
06_2DH.....	See Table 2-23
MSR_PERF_CAPABILITIES	
06_0FH, 06_17H.....	See Table 2-3

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_PERF_GLOBAL_CTRL	
06_0FH, 06_17H .....	See Table 2-3
MSR_PERF_GLOBAL_OVF_CTRL	
06_0FH, 06_17H .....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
MSR_PERF_GLOBAL_STATUS	
06_0FH, 06_17H .....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
MSR_PERF_METRICS	
06_7DH, 06_7EH .....	See Table 2-44
MSR_PERF_STATUS	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
06_2AH, 06_2DH .....	See Table 2-20
MSR_PKG_C10_RESIDENCY	
06_5CH, 06_7AH .....	See Table 2-12
06_45H.....	See Table 2-30 and Table 2-31
06_4FH.....	See Table 2-38
MSR_PKG_C2_RESIDENCY	
06_27H.....	See Table 2-5
06_5CH, 06_7AH .....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
MSR_PKG_C3_RESIDENCY	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
06_66H.....	See Table 2-42
06_57H, 06_85H.....	See Table 2-48
MSR_PKG_C4_RESIDENCY	
06_27H.....	See Table 2-5
MSR_PKG_C6_RESIDENCY	
06_27H.....	See Table 2-5
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
06_57H, 06_85H .....	See Table 2-48
MSR_PKG_C7_RESIDENCY	
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_57H, 06_85H .....	See Table 2-48
<b>MSR_PKG_C8_RESIDENCY</b>	
06_45H.....	See Table 2-31
06_4FH.....	See Table 2-38
<b>MSR_PKG_C9_RESIDENCY</b>	
06_45H.....	See Table 2-31
06_4FH.....	See Table 2-38
<b>MSR_PKG_CST_CONFIG_CONTROL</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 2-7
06_4CH.....	See Table 2-11
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_3AH.....	See Table 2-25
06_3EH.....	See Table 2-26
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_45H.....	See Table 2-31
06_3F.....	See Table 2-32
06_3DH.....	See Table 2-35
06_56H, 06_4FH .....	See Table 2-36
06_57H, 06_85H .....	See Table 2-48
<b>MSR_PKG_ENERGY_STATUS</b>	
06_37H, 06_4AH, 06_4CH, 06_5AH, 06_5DH .....	See Table 2-8
06_5CH, 06_7AH, 06_86H .....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3DH, 06_3EH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH	See Table 2-20
06_57H, 06_85H .....	See Table 2-48
<b>MSR_PKG_HDC_CONFIG</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_PKG_HDC_DEEP_RESIDENCY</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_PKG_HDC_SHALLOW_RESIDENCY</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_PKG_PERF_STATUS</b>	
06_5CH, 06_7AH, 06_86H .....	See Table 2-12
06_2DH.....	See Table 2-23
06_3AH, 06_3EH .....	See Table 2-26
06_3CH, 06_3DH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH .....	See Table 2-29
06_57H, 06_85H.....	See Table 2-48

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_PKG_POWER_INFO</b>	
06_4DH.....	See Table 2-10
06_5CH, 06_7AH, 06_86H.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3DH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
<b>MSR_PKG_POWER_LIMIT</b>	
06_37H, 06_4AH, 06_4CH, 06_5AH, 06_5DH.....	See Table 2-8
06_4DH.....	See Table 2-10
06_5CH, 06_7AH, 06_86H.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3DH, 06_3EH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH	See Table 2-20
06_57H, 06_85H.....	See Table 2-48
<b>MSR_PKG_C_IRTL1</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-29
<b>MSR_PKG_C_IRTL2</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-29
<b>MSR_PKG_C3_IRTL</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_2AH, 06_2DH.....	See Table 2-20
<b>MSR_PKG_C6_IRTL</b>	
06_2AH, 06_2DH.....	See Table 2-20
<b>MSR_PKG_C7_IRTL</b>	
06_2AH.....	See Table 2-21
<b>MSR_PLATFORM_BRV</b>	
0FH.....	See Table 2-50
<b>MSR_PLATFORM_ENERGY_COUNTER</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_PLATFORM_ID</b>	
06_0FH, 06_17H.....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 2-7
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
<b>MSR_PLATFORM_INFO</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3AH.....	See Table 2-25
06_3EH.....	See Table 2-26

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3CH, 06_45H, 06_46H .....	See Table 2-29 and Table 2-30
06_56H, 06_4FH .....	See Table 2-36
06_57H.....	See Table 2-48
<b>MSR_PLATFORM_POWER_LIMIT</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
<b>MSR_PMG_IO_CAPTURE_BASE</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_4CH.....	See Table 2-11
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_3AH.....	See Table 2-25
06_3EH.....	See Table 2-26
06_57H.....	See Table 2-48
<b>MSR_PMH_ESCRO</b>	
0FH.....	See Table 2-50
<b>MSR_PMH_ESCR1</b>	
0FH.....	See Table 2-50
<b>MSR_PMON_GLOBAL_CONFIG</b>	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
<b>MSR_PMON_GLOBAL_CTL</b>	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
<b>MSR_PMON_GLOBAL_STATUS</b>	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
<b>MSR_POWER_CTL</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
<b>MSR_PPO_ENERGY_STATUS</b>	
06_37H, 06_4AH, 06_5AH, 06_5DH .....	See Table 2-8
06_5CH, 06_7AH .....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20
06_57H.....	See Table 2-48
<b>MSR_PPO_POLICY</b>	
06_2AH, 06_45H .....	See Table 2-21
<b>MSR_PPO_POWER_LIMIT</b>	
06_4CH.....	See Table 2-11
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 2-20

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_57H.....	See Table 2-48
<b>MSR_PP1_ENERGY_STATUS</b>	
06_5CH, 06_7AH.....	See Table 2-12
06_2AH, 06_45H.....	See Table 2-21
06_3CH, 06_45H, 06_46H.....	See Table 2-30
<b>MSR_PP1_POLICY</b>	
06_2AH, 06_45H.....	See Table 2-21
06_3CH, 06_45H, 06_46H.....	See Table 2-30
<b>MSR_PP1_POWER_LIMIT</b>	
06_2AH, 06_45H.....	See Table 2-21
06_3CH, 06_45H, 06_46H.....	See Table 2-30
<b>MSR_PPERF</b>	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39
<b>MSR_PPIN</b>	
06_3EH.....	See Table 2-26
06_56H, 06_4FH.....	See Table 2-36
<b>MSR_PPIN_CTL</b>	
06_3EH.....	See Table 2-26
06_56H, 06_4FH.....	See Table 2-36
<b>MSR_PRMRR_BASE_0</b>	
06_7DH, 06_7EH.....	See Table 2-44
<b>MSR_PRMRR_PHYS_BASE</b>	
06_8EH, 06_9EH.....	See Table 2-41
<b>MSR_PRMRR_PHYS_MASK</b>	
06_8EH, 06_9EH.....	See Table 2-41
<b>MSR_PRMRR_VALID_CONFIG</b>	
06_8EH, 06_9EH.....	See Table 2-41
<b>MSR_RELOAD_FIXED_CTRx</b>	
06_86H.....	See Table 2-14
<b>MSR_RELOAD_PMCx</b>	
06_86H.....	See Table 2-14
<b>MSR_RING_RATIO_LIMIT</b>	
06_8EH, 06_9EH.....	See Table 2-41
<b>MSR_RO_PMON_BOX_CTRL</b>	
06_2EH.....	See Table 2-17
<b>MSR_RO_PMON_BOX_OVF_CTRL</b>	
06_2EH.....	See Table 2-17
<b>MSR_RO_PMON_BOX_STATUS</b>	
06_2EH.....	See Table 2-17
<b>MSR_RO_PMON_CTRL0</b>	
06_2EH.....	See Table 2-17

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_RO_PMON_CTR1 06_2EH.....	See Table 2-17
MSR_RO_PMON_CTR2 06_2EH.....	See Table 2-17
MSR_RO_PMON_CTR3 06_2EH.....	See Table 2-17
MSR_RO_PMON_CTR4 06_2EH.....	See Table 2-17
MSR_RO_PMON_CTR5 06_2EH.....	See Table 2-17
MSR_RO_PMON_CTR6 06_2EH.....	See Table 2-17
MSR_RO_PMON_CTR7 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SELO 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL1 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL2 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL3 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL4 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL5 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL6 06_2EH.....	See Table 2-17
MSR_RO_PMON_EVNT_SEL7 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P0 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P1 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P2 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P3 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P4 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P5 06_2EH.....	See Table 2-17

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_RO_PMON_IPERFO_P6 06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P7 06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P0 06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P1 06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P2 06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P3 06_2EH.....	See Table 2-17
MSR_R1_PMON_BOX_CTRL 06_2EH.....	See Table 2-17
MSR_R1_PMON_BOX_OVF_CTRL 06_2EH.....	See Table 2-17
MSR_R1_PMON_BOX_STATUS 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR10 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR11 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR12 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR13 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR14 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR15 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR8 06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR9 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL10 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL11 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL12 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL13 06_2EH.....	See Table 2-17



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R1_PMON_EVNT_SEL14 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL15 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL8 06_2EH.....	See Table 2-17
MSR_R1_PMON_EVNT_SEL9 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P10 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P11 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P12 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P13 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P14 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P15 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P8 06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P9 06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P4 06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P5 06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P6 06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P7 06_2EH.....	See Table 2-17
MSR_RAPL_POWER_UNIT 06_37H, 06_4AH, 06_5AH, 06_5DH.....	See Table 2-8
06_4DH.....	See Table 2-10
06_5CH, 06_7AH.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 2-20
06_3FH.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-36
06_57H.....	See Table 2-48
MSR_RAT_ESCR0 0FH.....	See Table 2-50

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_RAT_ESCR1	
0FH.....	See Table 2-50
MSR_RING_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H.....	See Table 2-30
MSR_SO_PMON_BOX_CTRL	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_SO_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_SO_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_SO_PMON_CTRL0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_CTRL1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_CTRL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_CTRL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_EVTN_SEL0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_EVTN_SEL1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_EVTN_SEL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_EVTN_SEL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_MASK	
06_2EH.....	See Table 2-17
MSR_SO_PMON_MATCH	
06_2EH.....	See Table 2-17
MSR_S1_PMON_BOX_CTRL	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_S1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_S1_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_S1_PMON_CTRL0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_CTRL1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_CTRL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_CTRL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SEL0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SEL1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SEL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SEL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_MASK	
06_2EH.....	See Table 2-17
MSR_S1_PMON_MATCH	
06_2EH.....	See Table 2-17
MSR_S2_PMON_BOX_CTL	
06_3FH.....	See Table 2-33
MSR_S2_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_S2_PMON_CTRL0	
06_3FH.....	See Table 2-33

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_S2_PMON_CTR1 06_3FH.....	See Table 2-33
MSR_S2_PMON_CTR2 06_3FH.....	See Table 2-33
MSR_S2_PMON_CTR3 06_3FH.....	See Table 2-33
MSR_S2_PMON_EVNTSELO 06_3FH.....	See Table 2-33
MSR_S2_PMON_EVNTSEL1 06_3FH.....	See Table 2-33
MSR_S2_PMON_EVNTSEL2 06_3FH.....	See Table 2-33
MSR_S2_PMON_EVNTSEL3 06_3FH.....	See Table 2-33
MSR_S3_PMON_BOX_CTL 06_3FH.....	See Table 2-33
MSR_S3_PMON_BOX_FILTER 06_3FH.....	See Table 2-33
MSR_S3_PMON_CTR0 06_3FH.....	See Table 2-33
MSR_S3_PMON_CTR1 06_3FH.....	See Table 2-33
MSR_S3_PMON_CTR2 06_3FH.....	See Table 2-33
MSR_S3_PMON_CTR3 06_3FH.....	See Table 2-33
MSR_S3_PMON_EVNTSELO 06_3FH.....	See Table 2-33
MSR_S3_PMON_EVNTSEL1 06_3FH.....	See Table 2-33
MSR_S3_PMON_EVNTSEL2 06_3FH.....	See Table 2-33
MSR_S3_PMON_EVNTSEL3 06_3FH.....	See Table 2-33
MSR_SAAT_ESCR0 0FH.....	See Table 2-50
MSR_SAAT_ESCR1 0FH.....	See Table 2-50
MSR_SGXOWNERPOCHO 06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH.....	See Table 2-39

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_SGXOWNEREPOCH1	
06_5CH, 06_7AH .....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_SMI_COUNT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_57H .....	See Table 2-48
MSR_SMM_BLOCKED	
06_5CH, 06_7AH .....	See Table 2-12
06_3CH, 06_45H, 06_46H .....	See Table 2-30
MSR_SMM_DELAYED	
06_5CH, 06_7AH .....	See Table 2-12
06_3CH, 06_45H, 06_46H .....	See Table 2-30
MSR_SMM_FEATURE_CONTROL	
06_5CH, 06_7AH .....	See Table 2-12
06_3CH, 06_45H, 06_46H .....	See Table 2-30
MSR_SMM_MCA_CAP	
06_5CH, 06_7AH .....	See Table 2-12
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_3FH .....	See Table 2-32
06_56H, 06_4FH .....	See Table 2-36
06_57H .....	See Table 2-48
MSR_SMRR_PHYSBASE	
06_0FH, 06_17H .....	See Table 2-3
MSR_SMRR_PHYSMASK	
06_0FH, 06_17H .....	See Table 2-3
MSR_SSU_ESCR0	
0FH .....	See Table 2-50
MSR_TBPU_ESCR0	
0FH .....	See Table 2-50
MSR_TBPU_ESCR1	
0FH .....	See Table 2-50
MSR_TC_ESCR0	
0FH .....	See Table 2-50
MSR_TC_ESCR1	
0FH .....	See Table 2-50
MSR_TC_PRECISE_EVENT	
0FH .....	See Table 2-50
MSR_TEMPERATURE_TARGET	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
06_2AH, 06_2DH .....	See Table 2-20
06_3EH.....	See Table 2-26
06_56H, 06_4FH.....	See Table 2-36
06_57H.....	See Table 2-48
<b>MSR_THERM2_CTL</b>	
06_0FH, 06_17H .....	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 2-4
0FH.....	See Table 2-50
06_0EH.....	See Table 2-53
06_09H.....	See Table 2-54
<b>MSR_THREAD_ID_INFO</b>	
06_3FH.....	See Table 2-32
<b>MSR_TRACE_HUB_STH ACPIBAR_BASE</b>	
06_8EH, 06_9EH .....	See Table 2-41
<b>MSR_TURBO_ACTIVATION_RATIO</b>	
06_5CH, 06_7AH .....	See Table 2-12
06_3AH.....	See Table 2-25
06_3CH, 06_45H, 06_46H .....	See Table 2-29
06_57H.....	See Table 2-48
<b>MSR_TURBO_GROUP_CORECNT</b>	
06_5CH, 06_7AH .....	See Table 2-12
<b>MSR_TURBO_POWER_CURRENT_LIMIT</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 2-15
<b>MSR_TURBO_RATIO_LIMIT</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH .....	See Table 2-6
06_4DH.....	See Table 2-10
06_5CH, 06_7AH .....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH .....	See Table 2-15
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
06_2EH.....	See Table 2-17
06_25H, 06_2CH .....	See Table 2-18
06_2FH.....	See Table 2-19
06_2AH, 06_45H .....	See Table 2-21
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26 and Table 2-27
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_3FH.....	See Table 2-32
06_3DH.....	See Table 2-35
06_56H, 06_4FH .....	See Table 2-36
06_55H.....	See Table 2-46

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_57H.....	See Table 2-48
<b>MSR_TURBO_RATIO_LIMIT1</b>	
06_3EH.....	See Table 2-26 and Table 2-27
06_3FH.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-36
<b>MSR_TURBO_RATIO_LIMIT2</b>	
06_3FH.....	See Table 2-32
<b>MSR_TURBO_RATIO_LIMIT3</b>	
06_56H.....	See Table 2-37
06_4FH.....	See Table 2-38
<b>MSR_TURBO_RATIO_LIMIT_CORES</b>	
06_55H.....	See Table 2-46
<b>MSR_U_PMON_BOX_STATUS</b>	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
<b>MSR_U_PMON_CTR</b>	
06_2EH.....	See Table 2-17
<b>MSR_U_PMON_CTR0</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_U_PMON_CTR1</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_U_PMON_EVNT_SEL</b>	
06_2EH.....	See Table 2-17
<b>MSR_U_PMON_EVNTSELO</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_U_PMON_EVNTSEL1</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_U_PMON_GLOBAL_CTRL</b>	
06_2EH.....	See Table 2-17
<b>MSR_U_PMON_GLOBAL_OVF_CTRL</b>	
06_2EH.....	See Table 2-17
<b>MSR_U_PMON_GLOBAL_STATUS</b>	
06_2EH.....	See Table 2-17
<b>MSR_U_PMON_UCLK_FIXED_CTL</b>	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
<b>MSR_U_PMON_UCLK_FIXED_CTR</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2DH .....	See Table 2-24
06_3FH .....	See Table 2-33
<b>MSR_U2L_ESCR0</b>	
0FH .....	See Table 2-50
<b>MSR_U2L_ESCR1</b>	
0FH .....	See Table 2-50
<b>MSR_UNC_ARB_PERFCTR0</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_ARB_PERFCTR1</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_ARB_PERFEVTSELO</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_ARB_PERFEVTSEL1</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_CBO_0_PERFCTR0</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_CBO_0_PERFCTR1</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_CBO_0_PERFCTR2</b>	
06_2AH .....	See Table 2-22
<b>MSR_UNC_CBO_0_PERFCTR3</b>	
06_2AH .....	See Table 2-22
<b>MSR_UNC_CBO_0_PERFEVTSELO</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
<b>MSR_UNC_CBO_0_PERFEVTSEL1</b>	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40



## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNC_CBO_0_PERFEVTSEL2 06_2AH .....	See Table 2-22
MSR_UNC_CBO_0_PERFEVTSEL3 06_2AH .....	See Table 2-22
MSR_UNC_CBO_0_UNIT_STATUS 06_2AH .....	See Table 2-22
MSR_UNC_CBO_1_PERFCTR0 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_1_PERFCTR1 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_1_PERFCTR2 06_2AH .....	See Table 2-22
MSR_UNC_CBO_1_PERFCTR3 06_2AH .....	See Table 2-22
MSR_UNC_CBO_1_PERFEVTSELO 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_1_PERFEVTSEL1 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_1_PERFEVTSEL2 06_2AH .....	See Table 2-22
MSR_UNC_CBO_1_PERFEVTSEL3 06_2AH .....	See Table 2-22
MSR_UNC_CBO_1_UNIT_STATUS 06_2AH .....	See Table 2-22
MSR_UNC_CBO_2_PERFCTR0 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_2_PERFCTR1 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_2_PERFCTR2 06_2AH .....	See Table 2-22

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNC_CBO_2_PERFCTR3	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_2_PERFEVTSELO	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_2_PERFEVTSEL1	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_2_PERFEVTSEL2	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_2_PERFEVTSEL3	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_2_UNIT_STATUS	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_3_PERFCTR0	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_3_PERFCTR1	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_3_PERFCTR2	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_3_PERFCTR3	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_3_PERFEVTSELO	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_3_PERFEVTSEL1	
06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_CBO_3_PERFEVTSEL2	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_3_PERFEVTSEL3	
06_2AH .....	See Table 2-22
MSR_UNC_CBO_3_UNIT_STATUS	
06_2AH .....	See Table 2-22

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNC_CBO_4_PERFCTR0 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFCTR1 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFCTR2 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFCTR3 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSELO 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSEL1 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSEL2 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSEL3 06_2AH .....	See Table 2-22
MSR_UNC_CBO_4_UNIT_STATUS 06_2AH .....	See Table 2-22
MSR_UNC_CBO_CONFIG 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_PERF_FIXED_CTR 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_PERF_FIXED_CTRL 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_PERF_GLOBAL_CTRL 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNC_PERF_GLOBAL_STATUS 06_2AH .....	See Table 2-22
06_3CH, 06_45H, 06_46H .....	See Table 2-30
06_4EH, 06_5EH .....	See Table 2-40
MSR_UNCORE_ADDR_OPCODE_MATCH 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_FIXED_CTR_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNCORE_FIXED_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERF_GLOBAL_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERF_GLOBAL_OVF_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERF_GLOBAL_STATUS 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL0 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL1 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL2 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL3 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL4 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL5 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL6 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PERFEVTSEL7 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC0 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC1 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC2 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC3 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC4 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC5 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
06_2EH .....	See Table 2-17
MSR_UNCORE_PMC6 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PMC7 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 2-16
MSR_UNCORE_PRMRR_BASE	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_UNCORE_PRMRR_MASK	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MSR_UNCORE_PRMRR_PHYS_BASE	
06_8EH, 06_9EH .....	See Table 2-41
MSR_UNCORE_PRMRR_PHYS_MASK	
06_8EH, 06_9EH .....	See Table 2-41
MSR_VR_CURRENT_CONFIG	
06_8CH, 06_8DH .....	See Table 2-45
MSR_W_PMON_BOX_CTRL	
06_2EH .....	See Table 2-17
MSR_W_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 2-17
MSR_W_PMON_BOX_STATUS	
06_2EH .....	See Table 2-17
MSR_W_PMON_CTRL0	
06_2EH .....	See Table 2-17
MSR_W_PMON_CTRL1	
06_2EH .....	See Table 2-17
MSR_W_PMON_CTRL2	
06_2EH .....	See Table 2-17
MSR_W_PMON_CTRL3	
06_2EH .....	See Table 2-17
MSR_W_PMON_EVNT_SELO	
06_2EH .....	See Table 2-17
MSR_W_PMON_EVNT_SEL1	
06_2EH .....	See Table 2-17
MSR_W_PMON_EVNT_SEL2	
06_2EH .....	See Table 2-17
MSR_W_PMON_EVNT_SEL3	
06_2EH .....	See Table 2-17
MSR_W_PMON_FIXED_CTRL	
06_2EH .....	See Table 2-17
MSR_W_PMON_FIXED_CTRL_CTL	
06_2EH .....	See Table 2-17
MSR_WEIGHTED_CORE_CO	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_6CH .....	See Table 2-39
MTRRfix16K_80000	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MTRRfix16K_A0000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_C0000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_C8000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_D0000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_D8000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_E0000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_E8000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_F0000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix4K_F8000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRfix64K_00000	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase0	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase1	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase2	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase3	
06_OEH .....	See Table 2-53
P6 Family .....	See Table 2-55

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MTRRphysBase4	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase5	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase6	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysBase7	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask0	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask1	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask2	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask3	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask4	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask5	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask6	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MTRRphysMask7	
06_0EH .....	See Table 2-53
P6 Family .....	See Table 2-55
MSR_TEST_CTRL	
06_86H .....	See Table 2-14
06_7DH, 06_7EH .....	See Table 2-44
P6 Family .....	See Table 2-55