Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

October 2006

Notice: The Intel® 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: 252046-018

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I^{2}C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I^{2}C bus/protocol and was developed by Intel. Implementations of the I^{2}C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel, Pentium, Celeron, Intel SpeedStep, Intel Xeon, Intel 64, and the Intel logo, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2002–2006, Intel Corporation. All rights reserved.

Contents

Preface 5
Summary Table of Changes 6
Documentation Changes.. 8

Revision History

Version	Description	Date
-001	• Initial Release	November 2002
-002	• Added 1-10 Documentation Changes. • Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	• Added 9 -17 Documentation Changes. • Removed Documentation Change #6 - References to bits Gen and Len Deleted. • Removed Documentation Change #4 - VIF Information Added to CLI Discussion.	February 2003
-004	• Removed Documentation changes 1-17. • Added Documentation changes 1-24.	June 2003
-005	• Removed Documentation Changes 1-24. • Added Documentation Changes 1-15.	September 2003
-006	• Added Documentation Changes 16- 34.	November 2003
-007	• Updated Documentation changes 14, 16, 17, and 28. • Added Documentation Changes 35-45.	January 2004
-008	• Removed Documentation Changes 1-45. • Added Documentation Changes 1-5.	March 2004
-009	• Added Documentation Changes 7-27.	May 2004
-010	• Removed Documentation Changes 1-27. • Added Documentation Changes 1.	August 2004
-011	• Added Documentation Changes 2-28.	November 2004
-012	• Removed Documentation Changes 1-28. • Added Documentation Changes 1-16.	March 2005
-013	• Updated title. • There are no Documentation Changes for this revision of the document.	July 2005
-014	• Added Documentation Changes 1-21.	September 2005
-015	• Removed Documentation Changes 1-21. • Added Documentation Changes 1-20.	March 9, 2006
-016	• Added Documentation changes 21-23.	March 27, 2006
-017	• Removed Documentation Changes 1-23. • Added Documentation Changes 1-36.	September 2006
-018	• Added Documentation Changes 37-42.	October 2006

Preface

This document is an update to the specifications contained in the Affected Documents/ Related Documents table below. This document is a compilation of documentation changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Affected Documents/Related Documents

Document Title	Document Number
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture	253665
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M	253666
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z	253667
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide	253668
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide	253669

Nomenclature

Documentation Changes include errors or omissions from the current published specifications. These changes will be incorporated in the next release of the Software Development Manual.

Summary Table of Changes

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

Codes Used in Summary Table

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

Summary Table of Documentation Changes (Sheet 1 of 2)

Number	Documentation Changes
1	MOV—Move to/from Control Registers
2	XTPR Update Control information added
3	Table on reserved bit checking has been corrected
4	MSR_THERM2_CTL description updated
5	Updated CPUID input format
6	Flag information added for POPF/POPFD/POPFQ
7	Restriction added for total size field, microcode update format
8	Flag check corrected
9	REP/REPE/REPZ/REPNE/REPNZ summary table updated
10	Documentation of CMASK bit range corrected
11	Note defines additional restrictions on APIC DFR programming
12	Tables documenting MCA error codes updated
13	PUSHA/PUSHAD information updated
14	VMCALL pseudocode updated
15	Information on code fetches in uncacheable memory updated
16	PUSH description updated
17	IRET/IRETD pseudocode updated
18	BSR summary table updated
19	SYSCALL and SYSRET pseudocode updated
20	VMX Debug exceptions paragraph deleted
21	FLD list of exceptions updated
22	CPUID register reference corrected
23	UCOMISS range corrected in pseudocode
24	CR information updated
25	VMXON opcode corrected
26	MSR references updated
27	CR0.WP coverage updated
28	Illegal register address flag description updated
29	CPUID call reference corrected
30	Note describing semaphore restrictions added

Summary Table of Documentation Changes (Sheet 2 of 2)

Number	Documentation Changes
31	DAS pseudocode updated
32	Entries added to CACHE-TLB table
33	Updated MOV to CR8 information
34	Information on ENTER instruction updated
35	Microcode update sections improved
36	Incorrect calls to CPUID.1:ECX[bit 9] have been corrected
37	String operation and EFLAGS.RF interactions clarified
38	MSR references corrected
39	Description of VM-entry checks on VM-execution control fields updated
40	STI, MOVSS/POPSS blocking behavior clarified
41	Correction for information on PEBS record size
42	Guest SMBASE entry added to table

Documentation Changes

1. MOV—Move to/from Control Registers

In Chapter 3, "MOV—Move to/from Control Registers" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*; the summary table has been corrected. The updated table is reproduced below. Change bars mark corrected lines.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 22 /*r*	MOV CR0,*r32*	N.E.	Valid	Move *r32* to CR0.
0F 22 /*r*	MOV CR0,*r64*	Valid	N.E.	Move *r64* to extended CR0.
0F 22 /*r*	MOV CR2,*r32*	N.E.	Valid	Move *r32* to CR2.
0F 22 /*r*	MOV CR2,*r64*	Valid	N.E.	Move *r64* to extended CR2.
0F 22 /*r*	MOV CR3,*r32*	N.E.	Valid	Move *r32* to CR3.
0F 22 /*r*	MOV CR3,*r64*	Valid	N.E.	Move *r64* to extended CR3.
0F 22 /*r*	MOV CR4,*r32*	N.E.	Valid	Move *r32* to CR4.
0F 22 /*r*	MOV CR4,*r64*	Valid	N.E.	Move *r64* to extended CR4.
0F 20 /*r*	MOV *r32*,CR0	N.E.	Valid	Move CR0 to *r32.*
0F 20 /*r*	MOV *r64*,CR0	Valid	N.E.	Move extended CR0 to *r64*.
0F 20 /*r*	MOV *r32*,CR2	N.E.	Valid	Move CR2 to *r32.*
0F 20 /*r*	MOV *r64*,CR2	Valid	N.E.	Move extended CR2 to *r64*.
0F 20 /*r*	MOV *r32*,CR3	N.E.	Valid	Move CR3 to *r32.*
0F 20 /*r*	MOV *r64*,CR3	Valid	N.E.	Move extended CR3 to *r64*.
0F 20 /*r*	MOV *r32*,CR4	N.E.	Valid	Move CR4 to *r32.*
0F 20 /*r*	MOV *r64*,CR4	Valid	N.E.	Move extended CR4 to *r64*.
0F 20 /*r*	MOV *r32*,CR8	N.E.	N.E.	Move CR8 to *r32.*
REX + 0F 20 /*r*	MOV *r64*,CR8	Valid	N.E.	Move extended CR8 to *r64*.[1]

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 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

2. XTPR Update Control information added

Figure 3-6 and Table 3-15 in Chapter 3, "CPUID—CPU Identification" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, information on the CPUID xTPR update control bit has been added.

The information is reproduced below.

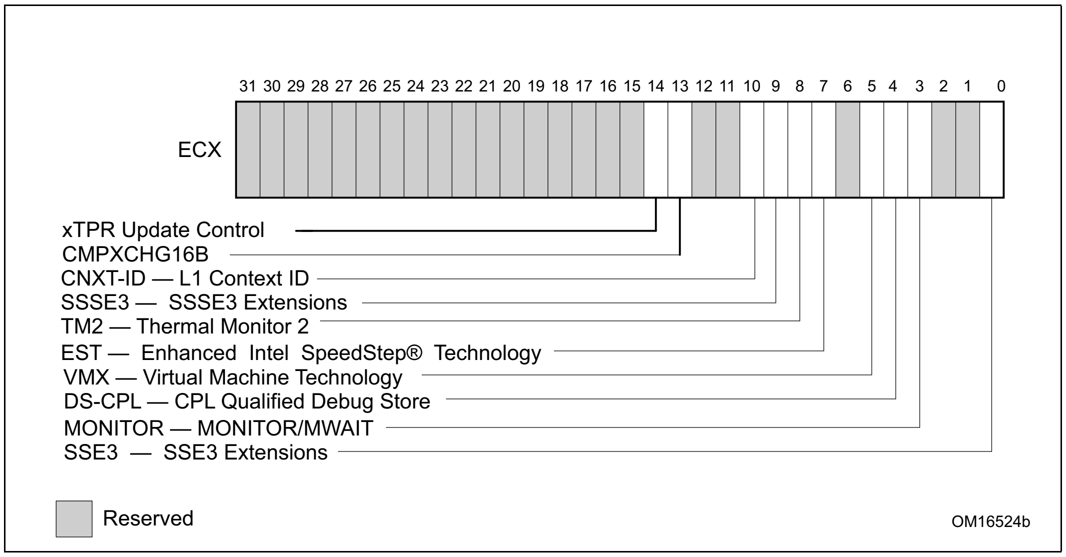

Figure 3-6. Extended Feature Information Returned in the ECX Register

Table 3-15. More on Extended Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
....	....	.....[not all lines in table are shown]
13	CMPXCHG16B	**CMPXCHG16B Available**. 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	**xTPR Update Control**. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
31 - 15	Reserved	Reserved

Information on the CPUID xTPR update control function has also been added to the discussions of IA32_MISC_ENABLES[bit 23] in Appendix B (Tables B-1, B-2, B-5) of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

The added information is reproduced below.

Register Address		Register Name Fields and Flags		Model Avail-ability	Shared/ Unique	Bit Description
Hex	Dec					
			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.

3. Table on reserved bit checking has been corrected

Table 3-5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* has been corrected. The corrections were to check bit values. The table is reproduced below. See the change bars for impacted lines.

Mode	Paging Mode	Paging Structure	Check Bits
32-bit	4-KByte pages (PAE = 0, PSE = 0)	PDE and PT	No reserved bits checked
	4-MByte page (PAE = 0, PSE = 1)	PDE	Bit [21]
	4-KByte page (PAE = 0, PSE = 1)	PDE	No reserved bits checked
	4-KByte and 4-MByte page (PAE = 0, PSE = 1)	PTE	No reserved bits checked
	4-KByte and 2-MByte pages (PAE = 1, PSE = x)	PDP table entry	Bits [63:40] & [8:5] & [2:1]
	2-MByte page (PAE = 1, PSE = x)	PDE	**Bits [62:40] & [20:13]**
	4-KByte pages (PAE = 1, PSE = x)	PDE	**Bits [62:40]**
	4-KByte pages (PAE = 1, PSE = x)	PTE	**Bits [62:40]**
64-bit	4-KByte and 2-MByte pages (PAE = 1, PSE = x)	PML4E	**Bits [51:40]**
	4-KByte and 2-MByte pages (PAE = 1, PSE = x)	PDPTE	**Bits [51:40]**
	2-MByte page (PAE = 1, PSE = x)	PDE, 2-MByte page	**Bits [51:40] & [20:13]**
	4-KByte pages (PAE = 1, PSE = x)	PDE, 4-KByte page	**Bits [51:40]**
	4-KByte pages (PAE = 1, PSE = x)	PTE	**Bits [51:40]**

NOTE:
x = Bit does not impact behavior.

4. MSR_THERM2_CTL description updated

In Table B-2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, the description for MSR_THERM2_CTL(address 19DH) has been updated. The update targets Family F processors. See the table segment below.

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique	Bit Description
Hex	Dec				
....	...	....	....	....	....
19DH	413	IMSR_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.

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique	Bit Description
Hex	Dec				
			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.

5. Updated CPUID input format

Some CPUID inputs require two input values. Documentation of the CPUID input format has been updated to reflect this requirement.

See the summary table below (from Chapter 3, “CPUID—CPU Identification”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (and, in some cases, ECX).

.--

In addition, the input format is summarized in a figure located in the first chapter of each volume; this figure is reproduced below.

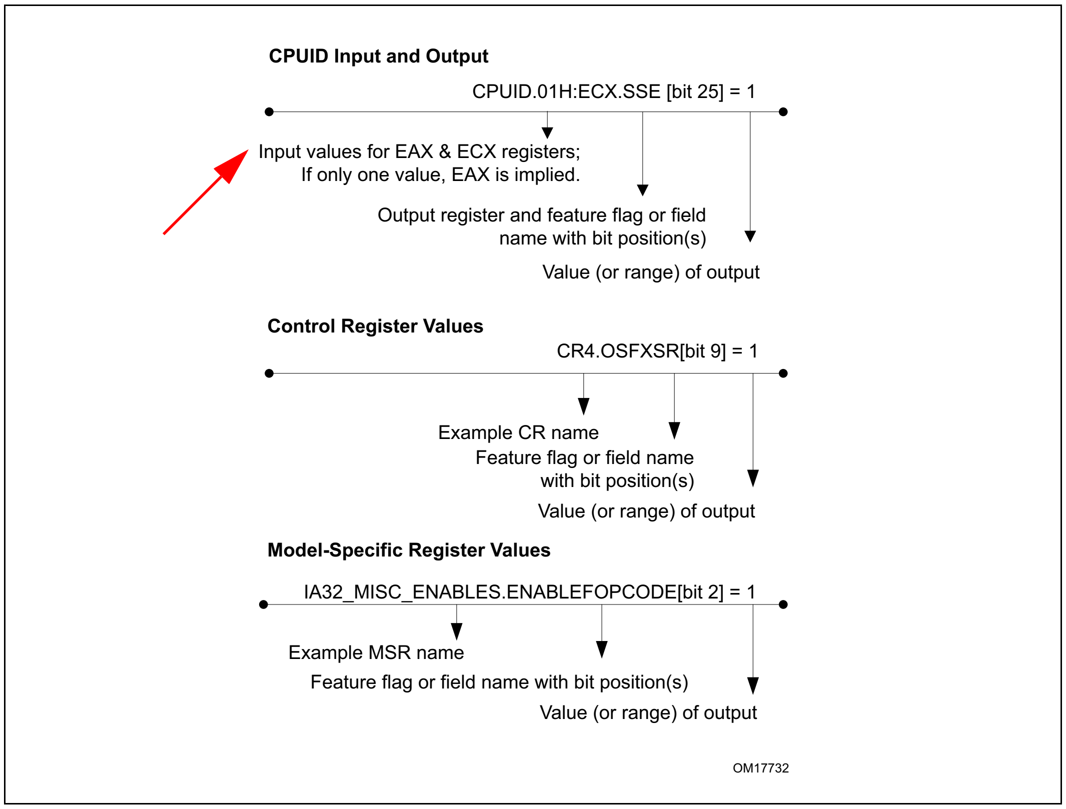
CPUID Input and Output
CPUID.01H:ECX.SSE [bit 25] = 1
Input values for EAX & ECX registers;
If only one value, EAX is implied.
Output register and feature flag or field
name with bit position(s)
Value (or range) of output
Control Register Values
CR4.OSFXSR[bit 9] = 1
Example CR name
Feature flag or field name
with bit position(s)
Value (or range) of output
Model-Specific Register Values
IA32_MISC_ENABLES.ENABLEFOPCODE[bit 2] = 1
Example MSR name
Feature flag or field name with bit position(s)
Value (or range) of output
OM17732

Figure 18-11. Syntax for CPUID, CR, and MSR Data Presentation

----

From Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*), below is an example of a CPUID feature that requires two inputs. See the change bars and the footnote.

7.7. Detecting Hardware Multi-Threading Support and Topology

Use the CPUID instruction to detect the presence of hardware multi-threading support in a physical processor. The following can be interpreted:

- **Hardware Multi-Threading feature flag (CPUID.1:EDX[28] = 1)** — Indicates when set that the physical package is capable of supporting Hyper-Threading Technology and/or multiple cores.
- **Logical processors per Package (CPUID.1:EBX[23:16])** — Indicates the maximum number of logical processors in a physical package. This represents the hardware capability as the processor has been manufactured.[1]

1. Operating system and BIOS may implement features that reduce the number of logical processors available in a platform to applications at runtime to less than the number of physical packages times the number of hardware-capable logical processors per package.

- **Cores per Package**[1] **(CPUID.(EAX=4, ECX=0**[2]**):EAX[31:26] + 1 = Y)** — Indicates the maximum number of processor cores (Y) in the physical package

The CPUID feature flag may indicate support for hardware multi-threading when only one logical processor available in the package. In this case, the decimal value represented by bits 16 through 23 in the EBX register will have a value of 1.

Software should note that the number of logical processors enabled by system software may be less than the value of “logical processors per package”. Similarly, the number of cores enabled by system software may be less than the value of “cores per package”.

.. ..Text omitted here... ... .

6. Flag information added for POPF/POPFD/POPFQ

In Chapter 4, “POPF/POPFD/POPFQ—Pop Stack” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B,* the flag information has been updated. The section is reprinted below with changes marked by change bars.

POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9D	POPF	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	N.E.	Valid	Pop top of stack into EFLAGS.
REX.W + 9D	POPFQ	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF[3], VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

1. Software must check CPUID for its support of leaf 4 when implementing support for multi-core. If CPUID leaf 4 is not available at runtime, software should handle the situation as if there is only one core per package.
2. Maximum number of cores in the physical package must be queried by executing CPUID with EAX =1 and a valid ECX input value. ECX input values start from 0.
3. RF is always zero after execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

When operating in protected mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and VIP, VIF, and VM. Here, the IOPL flags are unaffected, the VIP and VIF flags are cleared, and the VM flag is unaffected. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode, the IOPL must be equal to 3 to use POPF/POPFD instructions; VM, RF, IOPL, VIP, and VIF are unaffected. If the IOPL is less than 3, POPF/POPFD causes a general-protection exception (#GP).

In 64-bit mode, use REX.W to pop the top of stack to RFLAGS. The mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack, loads the lower 32 bits into RFLAGS, and zero extends the upper bits of RFLAGS.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS registers.

Operation

```
IF VM = 0 (* Not in Virtual-8086 Mode *)
   THEN IF CPL = 0
       THEN
           IF OperandSize = 32;
               THEN
                   EFLAGS ← Pop(); (* 32-bit pop *)
                   (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                   VIP and VIF are cleared; RF, VM, and all reserved bits are unaffected. *)
               ELSE IF (Operandsize = 64)
                   RFLAGS = Pop(); (* 64-bit pop *)
                   (* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP
                   and VIF are cleared; RF, VM, and all reserved bits are unaffected.*)
               ELSE (* OperandSize = 16 *)
                   EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                   (* All non-reserved flags can be modified. *)
           FI;
       ELSE (* CPL > 0 *)
           IF OperandSize = 32
               THEN
                   IF CPL > IOPL
                       THEN
                           EFLAGS ← Pop(); (* 32-bit pop *)
                           (* All non-reserved bits except IF, IOPL, RF, VIP, and
                           VIF can be modified; IF, IOPL, RF, VM, and all reserved
                           bits are unaffected; VIP and VIF are cleared. *)
                       ELSE
                           EFLAGS ← Pop(); (* 32-bit pop *)
                           (* All non-reserved bits except IOPL, RF, VIP, and VIF can be
                            modified; IOPL, RF, VM, and all reserved bits are
                            unaffected; VIP and VIF are cleared. *)
                   FI;
               ELSE IF (Operandsize = 64)
                   IF CPL > IOPL
                       THEN
                           RFLAGS ← Pop(); (* 64-bit pop *)
                           (* All non-reserved bits except IF, IOPL, RF, VIP, and
                           VIF can be modified; IF, IOPL, RF, VM, and all reserved
```

```
                                bits are unaffected; VIP and VIF are cleared. *)
                            ELSE
                                RFLAGS ← Pop(); (* 64-bit pop *)
                                (* All non-reserved bits except IOPL, RF, VIP, and VIF can be
                        modified; IOPL, RF, VM, and all reserved bits are
                        unaffected; VIP and VIF are cleared. *)
                        FI;
                    ELSE (* OperandSize = 16 *)
                        EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                        (* All non-reserved bits except IOPL can be modified; IOPL and all
                        reserved bits are unaffected. *)
                FI;
            FI;
        ELSE (* In Virtual-8086 Mode *)
            IF IOPL = 3
                THEN IF OperandSize = 32
                    THEN
                        EFLAGS ← Pop();
                        (* All non-reserved bits except VM, RF, IOPL, VIP, and VIF can be
                        modified; VM, RF, IOPL, VIP, VIF, and all reserved bits are unaffected. *)
                    ELSE
                        EFLAGS[15:0] ← Pop(); FI;
                        (* All non-reserved bits except IOPL can be modified;
                        IOPL and all reserved bits are unaffected. *)
            ELSE (* IOPL < 3 *)
                #GP(0); (* Trap to virtual-8086 monitor. *)
            FI;
        FI;
FI;
```

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
	If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack 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

7. Restriction added for total size field, microcode update format

In Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the total size field of the microcode update header must be in multiples of 1024 bytes (1 KBytes). This is now indicated in several locations. See the example below.

9.11.1 Microcode Update

. ...Text omitted here... ... .

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)).

.. ..Text omitted here... ... .

8. Flag check corrected

In Section 22.3.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*), a flag check was incorrectly indicated in the following section. The section is reproduced below, with the correction.

22.3.1.4 Checks on Guest RIP and RFLAGS

The following checks are performed on fields in the guest-state area corresponding to RIP and RFLAGS:

- RIP. The following checks are performed on processors that support Intel 64 Technology:
  - — Bits 63:32 must be 0 if the "IA-32e mode guest" VM-entry control is 0 or if the L bit (bit 13) in the access-rights field for CS is 0.
  - — If the processor supports N < 64 linear-address bits, bits 63:N must be identical if the "IA-32e mode guest" VM-entry control is 1 and the L bit in the access-rights field for CS is 1.[1] (No check applies if the processor supports 64 linear-address bits.)

- RFLAGS.
  - — Reserved bits 63:22 (bits 31:22 on processors that do not support Intel 64 Technology), bit 15, bit 5 and bit 3 must be 0 in the field, and reserved bit 1 must be 1.
  - — On processors that support Intel 64 Technology, the VM flag (bit 17) must be 0 if the “IA-32e mode guest” VM-entry control is 1.

The **IF flag (RFLAGS[bit 9]) must be 1** if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) is external interrupt.

.. ..Text omitted here... ... .

9. REP/REPE/REPZ/REPNE/REPNZ summary table updated

In Chapter 4, “REP/REPE/ REPZ/ REPNE/ REPNZ—Repeat String Operation Prefix”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, the summary table has been corrected. The updated table is reproduced below. Change bars mark corrected lines.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS *m8*, DX	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS *m8*, DX	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS *m16*, DX	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI.]
F3 6D	REP INS *m32*, DX	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS *r/m32*, DX	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS *m8, m8*	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS *m8, m8*	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS *m16, m16*	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m32, m32*	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS *m64, m64*	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, *r/m8*	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, *r/m8**	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, *r/m16*	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 6F	REP OUTS DX, *r/ m32*	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, *r/ m32*	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS AL	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS RAX	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS *m8*	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS *m8*	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS *m16*	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS *m32*	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS *m64*	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS *m8, m8*	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS *m8, m8*	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS *m16, m16*	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m32, m32*	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS *m64, m64*	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS *m8*	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS *m8*	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS *m16*	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS *m32*	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].
F3 REX.W AF	REPE SCAS *m64*	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS *m8, m8*	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS *m8, m8*	Valid	N.E.	Find matching bytes in [RDI] and [RSI].

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 A7	REPNE CMPS *m16, m16*	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m32, m32*	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS *m64, m64*	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS *m8*	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS *m8*	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS *m16*	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m32*	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS *m64*	Valid	N.E.	Find RAX, starting at [RDI].

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

10. Documentation of CMASK bit range corrected

In Figure 18-13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* (in our new edition), the figure showing the layout of MSR IA32PERFEVTSELx has been corrected to show the correct range of CMASK. The same change has been made to the paragraph text associated with the figure. See below.

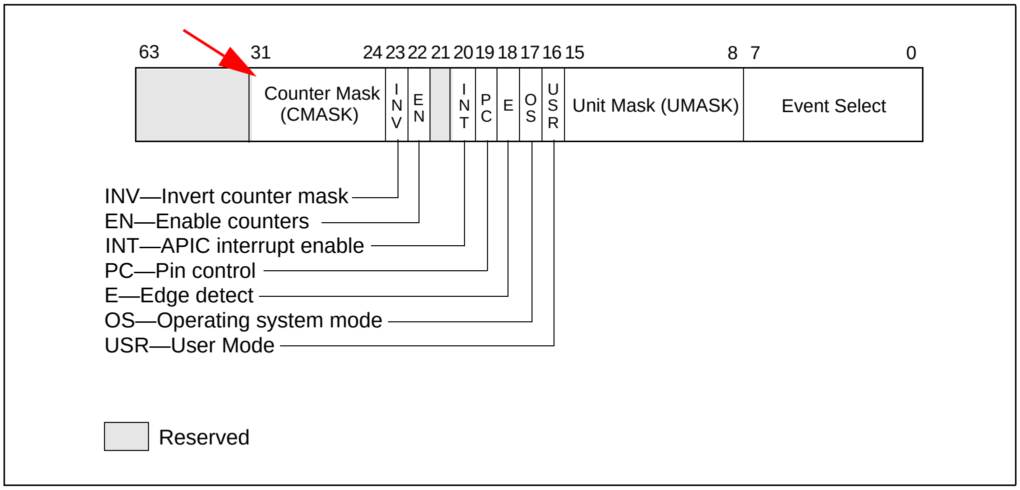

Figure 18-13. Layout of IA32_PERFEVTSELx MSRs

.. ..Text omitted here... ... .

- **Counter mask (CMASK) field (bits 24 through 31) —** When this field is not zero, the 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.

.. ..Text omitted here... ... .

11. Note defines additional restrictions on APIC DFR programming

In Section 8.6.2.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the note has been updated to capture a programming recommendation. The note is reproduced below. See the change bars.

8.6.2.2 Logical Destination Mode

.. ..Text omitted here... ... .

- The hierarchical cluster destination model can be used with Pentium 4, Intel Xeon, P6 family, or Pentium processors. With this model, a hierarchical network can be created by connecting different flat clusters via independent system or APIC buses. This scheme requires a cluster manager within each cluster, which is responsible for handling message passing between system or APIC buses. One cluster contains up to 4 agents. Thus 15 cluster managers, each with 4 agents, can form a network of up to 60 APIC agents. Note that hierarchical APIC networks requires a special cluster manager device, which is not part of the local or the I/O APIC units.

NOTE

All processors that have their APIC software enabled (using the spurious vector enable/disable bit) must have their DFRs (Destination Format Registers) programmed identically.

The default mode for DFR is flat mode. If you are using cluster mode, DFRs must be programmed before the APIC is software enabled. Since some chipsets do not accurately track a system view of the logical mode, program DFRs as soon as possible after starting the processor.

8.6.2.3 Broadcast/Self Delivery Mode

The destination shorthand field of the ICR allows the delivery mode to be by-passed in favor of broadcasting the IPI to all the processors on the system bus and/or back to itself (see Section 8.6.1, "Interrupt Command Register (ICR)"). Three destination shorthands are supported: self, all excluding self, and all including self. The destination mode is ignored when a destination shorthand is used.

... .Text omitted here... ... .

12. Tables documenting MCA error codes updated

Tables 14-15 and 14-16 in Section 14.7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, have been updated. Information in the tables is grouped differently. See the change bars below.

14.7. 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_MC*i*_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_MC*i*_STATUS register depending on the implementation of the machine-check architecture of the processor.

The MCA error codes are architecturally defined for IA-32 processors. However, the specific IA32_MC*i*_STATUS register that a code is 'written to' is model specific. To determine the cause of a machine-check exception, the machine-check exception handler must read the VAL flag for each IA32_MC*i*_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.

14.7.1 Simple Error Codes

Table 14-15 shows the simple error codes. These unique codes indicate global error information.

Table 14-15. 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.[1]
FRC Error	0000 0000 0000 0100	FRC (functional redundancy check) master/ slave error
Internal Timer Error	0000 0100 0000 0000	Internal timer error.
Internal Unclassified	0000 01xx xxxx xxxx	Internal unclassified errors. [2]

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. This is because no additional information is included in the machine check register.

14.7.2 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 memory hierarchy, and type of request. Table 14-16 shows the general form of the compound error codes.

Table 14-16. IA32_MCi_Status [15:0] Compound Error Code Encoding

Type	Form	Interpretation
Generic Memory Hierarchy	000F 0000 0000 11LL	Generic memory hierarchy error
TLB Errors	000F 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory Hierarchy Errors	000F 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_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 Section 14.7.2.1, "Correction Report Filtering (F) Bit" through Section 14.7.2.5, "Bus and Interconnect Errors".

.. ..Text omitted here... ... .

13. PUSHA/PUSHAD information updated

In the Description subsection, "PUSHA/PUSHAD—Push All General-Purpose Registers", in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, the language has been updated to correct an error (EBP was listed twice in the earlier version). See the change bar and bold text.

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
60	PUSHA	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Description

Pushes the contents of the general-purpose registers onto the stack. **The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16).** These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used.

Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

.. ..Text omitted here... ... .

14. VMCALL pseudocode updated

In the Operations subsection, "VMCALL—Call to VM Monitor", in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, the pseudocode has been updated. The change accommodates an architectural decision made pertaining to the handling of SMM.

The subsection is reprinted below. See the change bars.

Operation

```
IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF in SMM or if the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail(VMCALL executed in VMX root operation);
ELSIF (RFLAGS.VM = 1) OR (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN perform an SMM VM exit (see Section 24.16.2
     of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields are not valid (see Section 24.16.6.1 of the Intel® 64 and IA-32 Architec-
tures Software Developer's Manual, Volume 3B)
    THEN VMfailValid(VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
    read revision identifier in MSEG;
    IF revision identifier does not match that supported by processor
        THEN
            leave SMM;
            VMfailValid(VMCALL with incorrect MSEG revision identifier);
        ELSE
```

```
                read SMM-monitor features field in MSEG (see Section 24.16.6.2,
                in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
                IF features field is invalid
                    THEN
                        leave SMM;
                        VMfailValid(VMCALL with invalid SMM-monitor features);
                    ELSE activate dual-monitor treatment of SMIs and SMM (see Section 24.16.6
                    in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
                FI;
            FI;
FI;
```

15. Information on code fetches in uncacheable memory updated

In Section 10.3.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* the language has been updated to address confusion.

The section is reproduced below.

10.3.3 Code Fetches in Uncacheable Memory

Programs may execute code from uncacheable (UC) memory, but the implications are different from accessing data in UC memory. When doing code fetches, the processor never transitions from cacheable code to UC code speculatively. It also never speculatively fetches branch targets that result in UC code.

The processor may fetch the same UC cache line multiple times in order to decode an instruction once. It may decode consecutive UC instructions in a cacheline without fetching between each instruction. It may also fetch additional cachelines from the same or a consecutive 4-KByte page in order to decode one non-speculative UC instruction (this can be true even when the instruction is contained fully in one line).

Because of the above and because cacheline sizes may change in future processors, software should avoid placing memory-mapped I/O with read side effects in the same page or in a subsequent page used to execute UC code.

16. PUSH description updated

In the Description subsection, "PUSH—Push Word, Doubleword or Quadword Onto the Stack", in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, the language describing the instructions in real-address mode has been corrected.

The subsection is reprinted below. See the change bar.

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16, 32 or 64 bits). The operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2, 4 or 8 bytes).

In non-64-bit modes: if the address-size and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4. If both attributes are 16, the 16-bit SP register (stack pointer) is decremented by 2.

If the source operand is an immediate and its size is less than the address size of the stack, a sign-extended value is pushed on the stack. If the source operand is the FS or GS and its size is less than the address size of the stack, the zero-extended value is pushed on the stack.

The B flag in the stack segment's segment descriptor determines the stack's address-size attribute. The D flag in the current code segment's segment descriptor (with prefixes), determines the operand-size attribute and the address-size attribute of the source operand. Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned stack pointer (a stack pointer that is not be aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus if a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

In 64-bit mode, the instruction's default operation size is 64 bits. In a push, the 64-bit RSP register (stack pointer) is decremented by 8. A 66H override causes 16-bit operation. Note that pushing a 16-bit operand can result in the stack pointer misaligned to 8-byte boundary.

.. ..Text omitted here... ... .

17. IRET/IRETD pseudocode updated

In the Operation subsection, "IRET/IRETD—Interrupt Return", in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, the pseudocode has been updated to correct an error.

The subsection is reprinted below. See the change bar.

Operation

```
IF PE = 0
   THEN
       GOTO REAL-ADDRESS-MODE;
   ELSE
       IF (IA32_EFER.LMA = 0)
               THEN (* Protected mode *)
                   GOTO PROTECTED-MODE;
               ELSE (* IA-32e mode *)
                   GOTO IA-32e-MODE;
       FI;
FI;
REAL-ADDRESS-MODE;
   IF OperandSize = 32
       THEN
           IF top 12 bytes of stack not within stack limits
               THEN #SS; FI;
           tempEIP ← 4 bytes at end of stack
```

```
                IF tempEIP[31:16] is not zero THEN #GP(0); FI;
                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 *)
                IF top 6 bytes of stack are not within stack limits
                    THEN #SS; FI;
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop();
        FI;
        END;
    PROTECTED-MODE:
        IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
            THEN
                GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
        FI;
        IF NT = 1
            THEN
                GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
        FI;
        IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack are not within stack limits
                            THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
                tempEIP ← tempEIP AND FFFFH;
                tempEFLAGS ← tempEFLAGS AND FFFFH;
        FI;
        IF tempEFLAGS(VM) = 1 and CPL = 0
            THEN
                GOTO RETURN-TO-VIRTUAL-8086-MODE;
                (* PE = 1, VM = 1 in EFLAGS image *)
            ELSE
                GOTO PROTECTED-MODE-RETURN;
                (* PE = 1, VM = 0 in EFLAGS image *)
        FI;
    IA-32e-MODE:
        IF NT = 1
            THEN #GP(0);
        ELSE IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
```

```
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
            ELSE IF OperandSize = 16
                THEN
                    IF top 6 bytes of stack are not within stack limits
                            THEN #SS(0); FI;
                    tempEIP ← Pop();
                    tempCS ← Pop();
                    tempEFLAGS ← Pop();
                    tempEIP ← tempEIP AND FFFFH;
                    tempEFLAGS ← tempEFLAGS AND FFFFH;
                FI;
            ELSE (* OperandSize = 64 *)
                THEN
                        tempRIP ← Pop();
                        tempCS ← Pop();
                        tempEFLAGS ← Pop();
                        tempRSP ← Pop();
                        tempSS ← Pop();
    FI;
    GOTO IA-32e-MODE-RETURN;

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
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                IF instruction pointer not within code segment limits
                    THEN #GP(0); FI;
                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 *)
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack are not within stack limits
                    THEN #SS(0); FI;
                IF instruction pointer not within code segment limits
                    THEN #GP(0); FI;
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
            FI;
        ELSE
            #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;
```

```
RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, VM = 1 in flag image *)
    IF top 24 bytes of stack are not within stack segment limits
        THEN #SS(0); FI;
    IF instruction pointer not within code segment limits
        THEN #GP(0); FI;
    CS ← tempCS;
    EIP ← tempEIP;
    EFLAGS ← tempEFLAGS;
    TempESP ← Pop();
    TempSS ← Pop();
    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 *)
    SS:ESP ← TempSS:TempESP;
    CPL ← 3;
    (* Resume execution in Virtual-8086 mode *)
END;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    Read segment selector in link field of current TSS;
    IF local/global bit is set to local
    or index not within GDT limits
        THEN #TS (TSS selector); FI;
    Access TSS for task specified in link field of current TSS;
    IF TSS descriptor type is not TSS or if the TSS is marked not busy
        THEN #TS (TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    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 code segment limit
        THEN #GP(0); FI;
END;

PROTECTED-MODE-RETURN: (* PE = 1, VM = 0 in flags image *)
    IF return code segment selector is NULL
        THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond descriptor table limit
        THEN GP(selector); FI;
    Read segment descriptor pointed to by the return code segment selector;
    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 not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
```

```
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, VM = 0 in flags image, RPL = CPL *)
   IF new mode ≠ 64-Bit Mode
        THEN
            IF tempEIP is not within code segment limits
                THEN #GP(0); FI;
            EIP ← tempEIP;
        ELSE (* new mode = 64-bit mode *)
            IF tempRIP is non-canonical
                    THEN #GP(0); FI;
            RIP ← tempRIP;
   FI;
   CS ← tempCS; (* Segment descriptor information also loaded *)
   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
            THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; FI;
        IF OperandSize = 64
            THEN EFLAGS( VIF, VIP) ← tempEFLAGS; FI;
   FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
   IF OperandSize = 32
        THEN
            IF top 8 bytes on stack are not within limits
                THEN #SS(0); FI;
        ELSE (* OperandSize = 16 *)
            IF top 4 bytes on stack are not within limits
                THEN #SS(0); FI;
   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(SSselector); FI;
   Read segment descriptor pointed to by return segment selector;
   IF stack segment selector RPL ≠ RPL of the return code segment selector
   or the stack segment descriptor does not indicate a a writable data segment;
   or the stack segment DPL ≠ RPL of the return code segment selector
        THEN #GP(SS selector); FI;
   IF stack segment is not present
        THEN #SS(SS selector); FI;
   IF new mode ≠ 64-Bit Mode
```

```
            THEN
                IF tempEIP is not within code segment limits
                    THEN #GP(0); FI;
                EIP ← tempEIP;
            ELSE (* new mode = 64-bit mode *)
                IF tempRIP is non-canonical
                        THEN #GP(0); FI;
                RIP ← tempRIP;
        FI;
        CS ← tempCS;
        EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
        IF OperandSize = 32
            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
                    THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; FI;
                IF OperandSize = 64
                    THEN EFLAGS( VIF, VIP) ← tempEFLAGS; FI;
        FI;
        CPL ← RPL of the return code segment selector;
        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 (* Stored in hidden part of segment register *)
                    THEN (* Segment register invalid *)
                        SegmentSelector ← 0; (* NULL segment selector *)
                FI;
            OD;
    END;
    IA-32e-MODE-RETURN: (* IA32_EFER.LMA = 1, PE = 1, VM = 0 in flags image *)
        IF ( (return code segment selector is NULL) or (return RIP is non-canonical) or
                (SS selector is NULL going back to compatibility mode) or
                (SS selector is NULL going back to CPL3 64-bit mode) or
                (RPL <> CPL going back to non-CPL3 64-bit mode for a NULL SS selector) )
            THEN GP(0); FI;
        IF return code segment selector addrsses descriptor beyond descriptor table limit
            THEN GP(selector); FI;
        Read segment descriptor pointed to by the return code segment selector;
        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 not present
            THEN #NP(selector); FI;
        IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

18. BSR summary table updated

In Chapter 3, “BSR—Bit Scan Reverse”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, the summary table has been updated (/r was added). See the change bar below.

BSR—Bit Scan Reverse

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F BD /r	BSR *r16, r/m16*	Valid	Valid	Bit scan reverse on *r/m16.*
0F BD /r	BSR *r32, r/m32*	Valid	Valid	Bit scan reverse on *r/m32.*
REX.W + 0F BD	BSR *r64, r/m64*	Valid	N.E.	Bit scan reverse on *r/m64.*

19. SYSCALL and SYSRET pseudocode updated

In Chapter 4, “SYSCALL—Fast System Call” and “SYSRET—Return From Fast System Call” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, the pseudocode has been corrected.

See the change bars on the reprinted subsections below.

SYSCALL—Fast System Call

Operation

```
IF ((CS.L ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1))
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
RCX ← RIP;
RIP ← LSTAR_MSR;
R11 ← EFLAGS;
EFLAGS ← (EFLAGS MASKED BY IA32_FMASK);
CPL ← 0;
CS(SEL) ← IA32_STAR_MSR[47:32];
CS(DPL) ← 0;
CS(BASE) ← 0;
CS(LIMIT) ← 0xFFFFF;
CS(GRANULAR) ← 1;
SS(SEL) ← IA32_STAR_MSR[47:32] + 8;
SS(DPL) ← 0;
SS(BASE) ← 0;
SS(LIMIT) ← 0xFFFFF;
SS(GRANULAR) ← 1;
```

SYSRET—Return From Fast System Call

... . Not all text shown... ... ...

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL ≠ 0)
    THEN #GP(0); FI;
IF (RCX ≠ CANONICAL_ADDRESS)
    THEN #GP(0); FI;
IF (OPERAND_SIZE = 64)
    THEN (* Return to 64-Bit Mode *)
        EFLAGS ← R11;
        CPL ← 0x3;
        CS(SEL) ← IA32_STAR[63:48] + 16;
        CS(PL) ← 0x3;
        SS(SEL) ← IA32_STAR[63:48] + 8;
        SS(PL) ← 0x3;
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        EFLAGS ← R11;
        CPL ← 0x3;
        CS(SEL) ← IA32_STAR[63:48] ;
        CS(PL) ← 0x3;
        SS(SEL) ← IA32_STAR[63:48] + 8;
        SS(PL) ← 0x3;
        EIP ← ECX;
FI;
```

20. VMX Debug exceptions paragraph deleted

In Section 27.3.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* the first two paragraphs were deleted.

27.3.1 Debug Exceptions

If a VMM emulates a guest instruction that would encounter a debug trap (single step or data or I/O breakpoint), it should cause that trap to be delivered. The VMM should not inject the debug exception by using vector-on-entry, but should set the appropriate bits in the pending debug exceptions field. This method will give the trap the right priority with respect to other events. (If the exception bitmap was programmed to cause VM exits on debug exceptions, the debug trap will cause a VM exit. At this point, the trap can be injected with vector-on-entry with the proper priority.)

There is a valid pending debug exception if the BS bit (see Table 20-4) is set, regardless of the values of RFLAGS.TF or IA32_DEBUGCTL.BTF. The values of these bits do not impact the delivery of pending debug exceptions.

VMMs should exercise care when emulating a guest write (attempted using WRMSR) to IA32_DEBUGCTL to modify BTF if this is occurring with RFLAGS.TF = 1 and after a MOV SS or POP SS instruction (for example: while debug exceptions are blocked). Note the following:

- Normally, if WRMSR clears BTF while RFLAGS.TF = 1 and with debug exceptions blocked, a single-step trap will occur after WRMSR. A VMM emulating such an instruction should set the BS bit (see Table 20-4) in the pending debug exceptions field before VM entry.
- Normally, if WRMSR sets BTF while RFLAGS.TF = 1 and with debug exceptions blocked, neither a single-step trap nor a taken-branch trap can occur after WRMSR. A VMM emulating such an instruction should clear the BS bit (see Table 20-4) in the pending debug exceptions field before VM entry.

.. ..Text omitted here... ... .

21. FLD list of exceptions updated

In Chapter 3, “FLD—Load Floating Point Value”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, exceptions have been updated. See the change bars.

.. ..Text omitted here... ... .

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).
#D	Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

Protected Mode Exceptions

#GP(0)	If 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.

22. CPUID register reference corrected

In Section 7.5.5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the name of a register (EBX) has been corrected. Part of the section has been reprinted below. See the change bar.

7.5.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

- **Read APIC ID for a local APIC** — Code running on a logical processor can execute a MOV instruction to read the processor's local APIC ID register (see Section 8.4.6, "Local APIC ID"). This is the ID to use for directing physical destination mode interrupts to the processor.
- **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.
- **Read Initial APIC ID** — An APIC ID is assigned to a logical processor during power up and is called the initial APIC ID. This is the APIC ID reported by **CPUID.1:EBX[31:24]** and may be different from the current value read from the local APIC. Use the initial APIC ID to determine the topological relationship between logical processors.

Bits in the initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical level of the multi-threading resource topology in an MP system (See Section 7.10.1, “Hierarchical Mapping of Shared Resources”). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.

.. ..Text omitted here... ... .

23. UCOMISS range corrected in pseudocode

In “UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, a bit range has been corrected in the Operations subsection. See the change bar.

Operation

```
RESULT ← UnorderedCompare(SRC1[31:0] <> SRC2[31:0]) {
(* Set EFLAGS *)
CASE (RESULT) OF
    UNORDERED:          ZF,PF,CF ← 111;
    GREATER_THAN:       ZF,PF,CF ← 000;
    LESS_THAN:          ZF,PF,CF ← 001;
    EQUAL:              ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0;
```

24. CR information updated

In Section 2.5 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, the documentation of the XMXE bit has been added. See the change bars below.

2.5 CONTROL REGISTERS

Control registers (CR0, CR1, CR2, CR3, and CR4; see Figure 2-14) 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:

- Bits 63:32 of CR0 and CR4 are reserved and must be written with zeros. Writing a nonzero value to any of the upper 32 bits results in a general-protection exception, #GP(0).
- All 64 bits of CR2 are writable by software.
- Bits 51:40 of CR3 are reserved and must be 0.
- The MOV CRn instructions do not check that addresses written to CR2 and CR3 are within the linear-address or physical-address limitations of the implementation.
- 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 are described individually. In Figure 2-14, 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 page directory and two flags (PCD and PWT). This register is also known as the page-directory base register (PDBR). 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 page directory must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of the page directory 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 In IA-32e mode, the CR3 register contains the base address of the PML4 table.

  See also: Section 3.8, "36-Bit Physical Addressing Using the PAE Paging Mechanism."
- **CR4** — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. 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.
- **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.

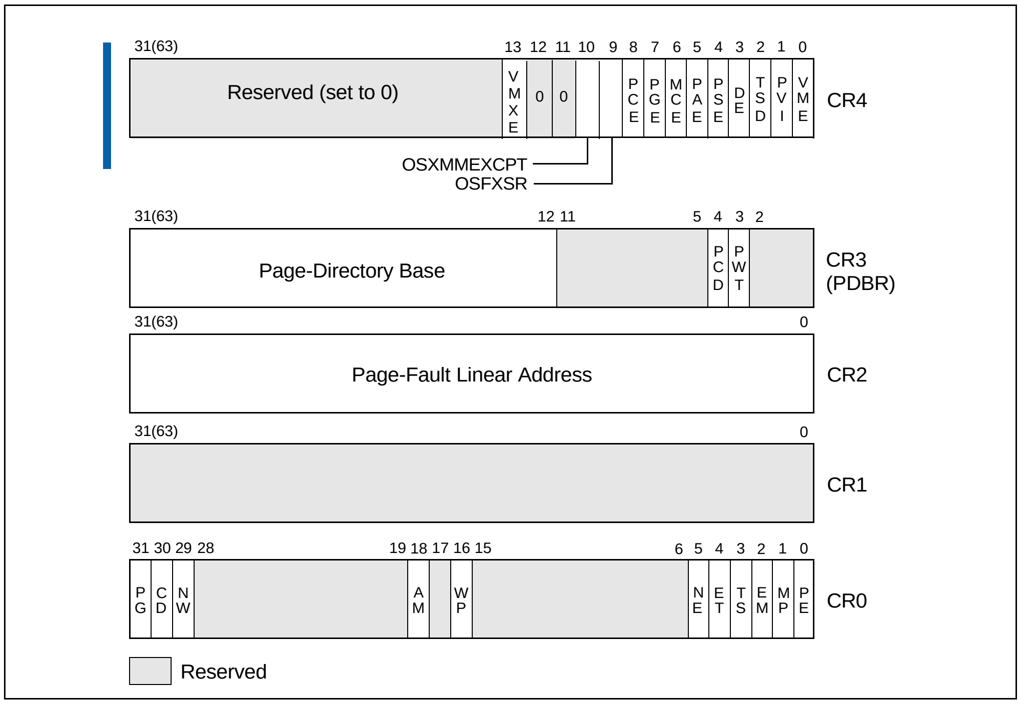

Figure 2-14. Control Registers

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

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: Section 3.6, "Paging (Virtual Memory) Overview."

On IA-32 processors that support Intel 64 Technology, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.

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 10-5. 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 10.5.3, "Preventing Caching," and Section 10.5, "Cache Control."

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 10-5 for detailed information about the affect of the NW flag on caching for other settings of the CD and NW flags.

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.

WP **Write Protect (bit 16 of CR0)** — Inhibits supervisor-level procedures from writing into user-level read-only pages when set; allows supervisor-level procedures to write into user-level read-only pages when clear (regardless of the U/S bit setting; see Section 3.7.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.

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.

See also: “Software Exception Handling” 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*.

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.

TS **Task Switched (bit 3 of CR0)** — Allows the saving of the x87 FPU/MMX/SSE/SSE2/ SSE3 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3 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 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 instruction; with the exception of PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, and CLFLUSH. 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 affect on the execution of x87 FPU/MMX/SSE/SSE2/SSE3 instructions.

Table 2-17 shows the actions taken when the processor encounters an x87 FPU instruction based on the settings of the TS, EM, and MP flags. Table 11-1 and 12-1 show the actions taken when the processor encounters an MMX/SSE/SSE2/SSE3 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 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 instruction; the x87 FPU/MMX/SSE/SSE2/ SSE3 context is never saved.

CR0 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.

EM **Emulation (bit 2 of CR0)** — 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 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 9-2 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-17 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 11-1). Thus, if an IA-32 processor incorporates MMX technology, the EM flag must be set to 0 to enable execution of MMX instructions.

Similarly for SSE/SSE2/SSE3 extensions, when the EM flag is set, execution of most SSE/SSE2/SSE3 instructions causes an invalid opcode exception (#UD) to be generated (see Table 12-1). If an IA-32 processor incorporates the SSE/SSE2/SSE3 extensions, the EM flag must be set to 0 to enable execution of these extensions. SSE/SSE2/SSE3 instructions not affected by the EM flag include: PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, and CLFLUSH.

MP **Monitor Coprocessor (bit 1 of CR0).** — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CR0). 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 9-2 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-17 shows the interaction of the MP, EM, and TS flags.

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.”

PCD **Page-level Cache Disable (bit 4 of CR3)** — Controls caching of the current page directory. When the PCD flag is set, caching of the page-directory is prevented; when the flag is clear, the page-directory can be cached. This flag affects only the processor’s internal caches (both L1 and L2, when present). The processor ignores this flag if paging is not used (the PG flag in register CR0 is clear) or the CD (cache disable) flag in CR0 is set.

See also: Chapter 10, “Memory Cache Control” (for more about the use of the PCD flag) and Section 3.7.6, “Page-Directory and Page-Table Entries” (for a description of a companion PCD flag in page-directory and page-table entries).

PWT **Page-level Writes Transparent (bit 3 of CR3)** — Controls the write-through or write-back caching policy of the current page directory. When the PWT flag is set, write-through caching is enabled; when the flag is clear, write-back caching is enabled. This flag affects only internal caches (both L1 and L2, when present). The processor ignores this flag if paging is not used (the PG flag in register CR0 is clear) or the CD (cache disable) flag in CR0 is set.

See also: Section 10.5, “Cache Control” (for more information about the use of this flag), and Section 3.7.6, “Page-Directory and Page-Table Entries” (for a description of a companion PCD flag in the page-directory and page-table entries).

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 multitasking and multiple-processor environments.

See also: Section 15.3, “Interrupt and Exception Handling in Virtual-8086 Mode.”

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 15.4, “Protected-Mode Virtual Interrupts.”

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.

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 18.2.2, “Debug Registers DR4 and DR5.”

PSE **Page Size Extensions (bit 4 of CR4)** — Enables 4-MByte pages when set; restricts pages to 4 KBytes when clear.

See also: Section 3.6.1, “Paging Options.”

PAE **Physical Address Extension (bit 5 of CR4)** — When set, enables paging mechanism to reference greater-or-equal-than-36-bit physical addresses. When clear, restricts physical addresses to 32 bits. PAE must be enabled to enable IA-32e mode operation. Enabling and disabling IA-32e mode operation also requires modifying CR4.PAE.

See also: Section 3.8, “36-Bit Physical Addressing Using the PAE Paging Mechanism.”

MCE **Machine-Check Enable (bit 6 of CR4)** — Enables the machine-check exception when set; disables the machine-check exception when clear.

See also: Chapter 14, “Machine-Check Architecture.”

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 or 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 3.12, “Translation Lookaside Buffers (TLBs).”

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.

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 instructions, with the exception of the PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, and CLFLUSH.

If this flag is clear, the FXSAVE and FXRSTOR instructions will save and restore the contents of the x87 FPU and MMX instructions, 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, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, and CLFLUSH. The operating system or executive must explicitly set this flag.

NOTE

CPUID feature flags FXSR, SSE, SSE2, and SSE3 indicate availability of the FXSAVE/FXRESTOR instructions, SSE extensions, SSE2 extensions, and SSE3 extensions respectively. The OSFXSR bit provides operating system software with a means of enabling these features and indicating that the operating system supports the features.

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 (#XF) is generated. SIMD floating-point exceptions are only generated by SSE/SSE2/SSE3 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.

XMXE

VMX-Enable Bit (bit 13 of CR4) — Enables VMX operation when set. See Chapter 19, "Introduction to Virtual-Machine Extensions."

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.

25. VMXON opcode corrected

In Chapter 5, "VMXON—Enter VMX Operation", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, the summary chart has been corrected to address a typo. See below.

VMXON—Enter VMX Operation

Opcode	Instruction	Description
F3 0F C7 /6	VMXON m64	Enter VMX root operation.

.. ..Text omitted here... ... .

26. MSR references updated

In Sections 20.6.1 and 20.6.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, two MSR references (IA32_VMX_PINBASED_CTLS, IA32_VMX_PROCBASED_CTLS) have been corrected. The text is reprinted below to show context. See the change bars.

20.6.1 Pin-Based VM-Execution Controls

The pin-based VM-execution controls constitute a 32-bit vector that governs the handling of asynchronous events (for example: interrupts).[1] Table 20-5 lists the controls supported. See Chapter 21 for how these controls affect processor behavior in VMX non-root operation.

1. Some asynchronous events cause VM exits regardless of the settings of the pin-based VM-execution controls (see Section 21.2).

Table 20-5. Definitions of Pin-Based VM-Execution Controls

Bit Position(s)	Name	Description
0	External-interrupt exiting	If this control is 1, external interrupts cause VM exits. Otherwise, they are delivered normally through the guest interrupt-descriptor table (IDT). If this control is 1, the value of RFLAGS.IF does not affect interrupt blocking.
3	NMI exiting	If this control is 1, non-maskable interrupts (NMIs) cause VM exits. Otherwise, they are delivered normally using descriptor 2 of the IDT. This control also determines interactions between IRET and blocking by NMI (see Section 21.3).

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability **MSR IA32_VMX_PINBASED_CTLS**, (see Appendix G.2) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 22.2).

20.6.2 Processor-Based VM-Execution Controls

The processor-based VM-execution controls constitute a 32-bit vector that governs the handling of synchronous events, mainly those caused by the execution of specific instructions.[1] Table 20-6 lists the controls supported. See Chapter 21 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 20-6. Definitions of Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
2	Interrupt-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 20.4.2).
3	Use TSC offsetting	This control determines whether executions of RDTSC and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 20.6.4.1 and Section 21.3).
7	HLT exiting	This control determines whether executions of HLT cause VM exits.
9	INVLPG exiting	This determines whether executions of INVLPG cause VM exits.
10	MWAIT exiting	This control determines whether executions of MWAIT cause VM exits.
11	RDPMC exiting	This control determines whether executions of RDPMC cause VM exits.
12	RDTSC exiting	This control determines whether executions of RDTSC cause VM exits.

1. Some instructions cause VM exits regardless of the settings of the processor-based VM-execution controls (see Section 21.1.2), as do task switches (see Section 21.2).

Table 20-6. Definitions of Processor-Based VM-Execution Controls (Continued)

Bit Position(s)	Name	Description
19	CR8-load exiting	This control determines whether executions of MOV to CR8 cause VM exits. This control must be 0 on processors that do not support Intel 64 Technology.
20	CR8-store exiting	This control determines whether executions of MOV from CR8 cause VM exits. This control must be 0 on processors that do not support Intel 64 Technology.
21	Use TPR shadow	Setting this control to 1 activates the TPR shadow, which is maintained in a page of memory addressed by the virtual-APIC address. See Section 21.3. This control must be 0 on processors that do not support Intel 64 Technology.
23	MOV-DR exiting	This control determines whether executions of MOV DR cause VM exits.
24	Unconditional I/O exiting	This control determines whether executions of I/O instructions (IN, INS/INSB/INSW/INSD, OUT, and OUTS/OUTSB/OUTSW/OUTSD) cause VM exits. This control is ignored if the “use I/O bitmaps” control is 1.
25	Use I/O bitmaps	This control determines whether I/O bitmaps are used to restrict executions of I/O instructions (see Section 20.6.4 and Section 21.1.3). For this control, “0” means “do not use I/O bitmaps” and “1” means “use I/O bitmaps.” If the I/O bitmaps are used, the setting of the “unconditional I/O exiting” control is ignored.
28	Use MSR bitmaps	This control determines whether MSR bitmaps are used to control execution of the RDMSR and WRMSR instructions (see Section 20.6.4 and Section 21.1.3). For this control, “0” means “do not use MSR bitmaps” and “1” means “use MSR bitmaps.” If the MSR bitmaps are not used, all executions of the RDMSR and WRMSR instructions cause VM exits. Not all processors support the 1-setting of this control. Software may consult the VMX capability MSR IA32_VMX_PROCBASED_CTLS (see Appendix G.2) to determine whether that setting is supported.
29	MONITOR exiting	This control determines whether executions of MONITOR cause VM exits.
30	PAUSE exiting	This control determines whether executions of PAUSE cause VM exits.

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability **MSR IA32_VMX_PROCBASED_CTLS** (see Appendix G.2) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 22.2).

27. CR0.WP coverage updated

In Section 2.5 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, the text has been updated to make the language more precise. See the change bars.

2.5 CONTROL REGISTERS

.. ..Text omitted here... ... .

WP **Write Protect (bit 16 of CR0)** — Inhibits supervisor-level procedures from writing into user-level read-only pages when set; allows supervisor-level procedures to write into user-level read-only pages when clear (regardless of the U/S bit setting; see Section 3.7.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.

... ...Text omitted here... ... .

In Section 4.11.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, a similar update has been made. See the change bar below.

4.11.1 Page Type

.. ..Text omitted here... ... .

Starting with the P6 family, Intel processors allow user-mode pages to be write-protected against supervisor-mode access. Setting CR0.WP = 1 enables supervisor-mode sensitivity to user-mode, write protected pages. Supervisor pages which are read-only are not writable from any privilege level (if CR0.WP = 0). This supervisor write-protect feature is useful for implementing a "copy-on-write" strategy used by some operating systems, such as UNIX*, for task creation (also called forking or spawning). When a new task is created.... ... .

.. ..Text omitted here... ... .

28. Illegal register address flag description updated

In Figure 8-2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the language describing the illegal register address flag has been updated. The table is reprinted below. See the change bar.

FLAG	Function
Send Checksum Error	(P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it sent on the APIC bus.
Receive Checksum Error	(P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it received on the APIC bus.
Send Accept Error	(P6 family and Pentium processors only) Set when the local APIC detects that a message it sent was not accepted by any APIC on the APIC bus.
Receive Accept Error	(P6 family and Pentium processors only) Set when the local APIC detects that the message it received was not accepted by any APIC on the APIC bus, including itself.

FLAG	Function
Send Checksum Error	(P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it sent on the APIC bus.
Receive Checksum Error	(P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it received on the APIC bus.
Send Illegal Vector	Set when the local APIC detects an illegal vector in the message that it is sending.
Receive Illegal Vector	Set when the local APIC detects an illegal vector in the message it received, including an illegal vector code in the local vector table interrupts or in a self-interrupt.
Illegal Reg. Address	(Pentium 4, Intel Xeon, and P6 family processors only) Set when the processor is trying to access a register in the processor's local APIC register address space that is reserved (see Table 8-1). Addresses in one the 0x10 byte regions marked reserved are illegal register addresses. The Local APIC Register Map is the address range of the APIC register base address (specified in the IA32_APIC_BASE MSR) plus 4 KBytes.

29. CPUID call reference corrected

In Section 3.3.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, a CPUID reference has been corrected. See the change bar and the bold text.

3.3.1 Physical Address Space for Processors with Intel® 64 Technology

On processors that support Intel 64 Technology (CPUID.80000001:EDX[29] = 1), the size of the physical address range is implementation-specific and indicated by **CPUID.80000008H:EAX[bits 7-0]**. For the format of information returned in EAX, see "CPUID—CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

30. Note describing semaphore restrictions added

In Section 7.1.2.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the language has been added to clarify a usage restriction. See the change bar below.

7.1.2.2 Software Controlled Bus Locking

To explicitly force the LOCK semantics, software can use the LOCK prefix with the following instructions when they are used to modify a memory location. An invalid-opcode exception (#UD) is generated when the LOCK prefix is used with any other instruction or when no write operation is made to memory (that is, when the destination operand is in a register).

- The bit test and modify instructions (BTS, BTR, and BTC).
- The exchange instructions (XADD, CMPXCHG, and CMPXCHG8B).

- The LOCK prefix is automatically assumed for XCHG instruction.
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG.
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area.

Software should access semaphores (shared memory used for signalling between multiple processors) using identical addresses and operand lengths. For example, if one processor accesses a semaphore using a word access, other processors should not access the semaphore using a byte access.

NOTE

Do not implement semaphores using the WC memory type. Do not perform non-temporal stores to a cache line containing a location used to implement a semaphore.

.. ..Text omitted here... ... .

31. DAS pseudocode updated

In Chapter 3, “DAS—Decimal Adjust AL after Subtraction”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, the pseudocode has been corrected. See the change bar below.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ← AL;
        old_CF ← CF;
        CF ← 0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL ← AL – 6;
                CF ← old_CF or (Borrow from AL ← AL − 6);
                AF ← 1;
            ELSE
                AF ← 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ← AL − 60H;
                CF ← 1;
        FI;
FI;
```

32. Entries added to CACHE-TLB table

In Table 3-17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, entries have been added to the table. The table is reprinted below, see the change bars.

- --

Table 3-17. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4 MByte pages, 4-way set associative, 2 entries
03H	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
22H	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
49H	2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
50H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries

Table 3-17. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
51H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
56H	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
5BH	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
66H	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Trace cache: 12 K-μop, 8-way set associative
71H	Trace cache: 16 K-μop, 8-way set associative
72H	Trace cache: 32 K-μop, 8-way set associative
78H	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
82H	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
B0H	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B3H	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
F0H	64-Byte prefetching
F1H	128-Byte prefetching

33. Updated MOV to CR8 information

In Sections 21.1.3 through 21.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* changes have been made to clarify VMX behavior. See the change bars.

21.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:

- **CLTS.** Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - — If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 19.8), in which case CLTS causes a general-protection exception.
  - — If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.
  - — If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit (see Section 21.1.3).
- **IRET.** Behavior of IRET with regard to the blocking by NMI (see Table 20-3) is determined by the setting of the "NMI exiting" VM-execution control:
  - — If the control is 0, IRET operates normally and unblocks NMIs.
  - — If the control is 1, IRET does not affect blocking by NMI.
- **LMSW.** An execution of LMSW that does not cause a VM exit (see Section 21.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. It causes a general-protection exception if it attempts to set any bit to a value not supported in VMX operation (see Section 19.8)
- **MOV from CR0.** The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.

  Note that, depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.
- **MOV from CR4.** The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 returns the value of the CR4 read shadow.

Note that, depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.

- **MOV from CR8.** Behavior of the MOV from CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) is determined by the settings of the “CR8-store exiting” and “use TPR shadow” VM-execution controls:
  - — If both controls are 0, MOV from CR8 operates normally.
  - — If the “CR8-store exiting” VM-execution control is 0 and the “use TPR shadow” VM-execution control is 1, MOV from CR8 reads from the TPR shadow. Specifically, it loads bits 3:0 of its destination operand with the value of bits 7:4 of byte 128 of the page referenced by the virtual-APIC page address (see Section 20.6.7). Bits 63:4 of the destination operand are cleared.
  - — If the “CR8-store exiting” VM-execution control is 1, MOV from CR8 causes a VM exit (see Section 21.1.3); the “use TPR shadow” VM-execution control is ignored in this case.
- **MOV to CR0.** An execution of MOV to CR0 that does not cause a VM exit (see Section 21.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. It causes a general-protection exception if it attempts to set any bit to a value not supported in VMX operation (see Section 19.8).
- **MOV to CR4.** An execution of MOV to CR4 that does not cause a VM exit (see Section 21.1.3) leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit to a value not supported in VMX operation (see Section 19.8).
- **MOV to CR8.** Behavior of the MOV to CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) is determined by the settings of the “CR8-load exiting” and “use TPR shadow” VM-execution controls:
  - — If both controls are 0, MOV to CR8 operates normally.
  - — If the “CR8-load exiting” VM-execution control is 0 and the “use TPR shadow” VM-execution control is 1, MOV to CR8 writes to the TPR shadow. Specifically, it stores bits 3:0 of its source operand into bits 7:4 of byte 128 of the page referenced by the virtual-APIC page address (see Section 20.6.7); bits 3:0 of that byte and bytes 129-131 of that page are cleared. Such a store may cause a VM exit to occur after it completes (see Section 21.1.3).
  - — If the “CR8-load exiting” VM-execution control is 1, MOV to CR8 causes a VM exit (see Section 21.1.3); the “use TPR shadow” VM-execution control is ignored in this case.
- **RDMSR.** Section 21.1.3 identifies when executions of the RDMSR instruction cause VM exits. If an execution of RDMSR does not cause a VM exit and if RCX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the RDMSR instruction is determined by the setting of the “use TSC offsetting” VM-execution control as well as the TSC offset:
  - — If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.
  - — If the control is 1, RDMSR loads EAX:EDX with the sum (using signed addition) of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset (interpreted as a signed value).

- **RDTSC.** Behavior of the RDTSC instruction is determined by the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls as well as the TSC offset:
  - — If both controls are 0, RDTSC operates normally.
  - — If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, RDTSC loads EAX:EDX with the sum (using signed addition) of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset (interpreted as a signed value).
  - — If the "RDTSC exiting" VM-execution control is 1, RDTSC causes a VM exit (see Section 21.1.3).
- **SMSW.** The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.

  Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.
- **WRMSR.** Section 21.1.3 identifies when executions of the WRMSR instruction cause VM exits. If an execution of WRMSR causes neither a fault or a VM exit and if RCX contains 79H (indicating IA32_BIOS_UPDT_TRIG MSR); no microcode update is loaded and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.

34. Information on ENTER instruction updated

In Chapter 3, "ENTER—Make Stack Frame for Procedure Parameters", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* information has been added. The section is reproduced below. See the change bars.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
C8 *iw* 00	ENTER *imm16*, 0	Valid	Valid	Create a stack frame for a procedure.
C8 *iw* 01	ENTER *imm16*,1	Valid	Valid	Create a nested stack frame for a procedure.
C8 *iw* ib	ENTER *imm16*, *imm8*	Valid	Valid	Create a nested stack frame for a procedure.

Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits), EBP (32 bits), or RBP (64 bits) register specifies the current frame pointer and whether SP (16 bits), ESP (32 bits), or RSP (64 bits) specifies the stack pointer. In 64-bit mode, stack-size attribute is always 64-bits.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded.

Operation

```
NestingLevel ← NestingLevel MOD 32
IF 64-Bit Mode (StackSize = 64)
   THEN
      Push(RBP);
      FrameTemp ← RSP;
   ELSE IF StackSize = 32
      THEN
         Push(EBP);
         FrameTemp ← ESP; FI;
   ELSE (* StackSize = 16 *)
         Push(BP);
         FrameTemp ← SP;
FI;
IF NestingLevel = 0
   THEN GOTO CONTINUE;
FI;

IF (NestingLevel > 1)
   THEN FOR i ← 1 to (NestingLevel - 1)
      DO
```

```
                    IF 64-Bit Mode (StackSize = 64)
                        THEN
                            RBP ← RBP - 8;
                            Push([RBP]); (* Quadword push *)
                        ELSE IF OperandSize = 32
                            THEN
                                IF StackSize = 32
                                    EBP ← EBP - 4;
                                    Push([EBP]); (* Doubleword push *)
                                ELSE (* StackSize = 16 *)
                                    BP ← BP - 4;
                                    Push([BP]); (* Doubleword push *)
                                FI;
                            FI;
                        ELSE (* OperandSize = 16 *)
                            IF StackSize = 32
                                THEN
                                    EBP ← EBP - 2;
                                    Push([EBP]); (* Word push *)
                                ELSE (* StackSize = 16 *)
                                    BP ← BP - 2;
                                    Push([BP]); (* Word push *)
                            FI;
                        FI;
   OD;
FI;

IF 64-Bit Mode (StackSize = 64)
   THEN
        Push(FrameTemp); (* Quadword push *)
   ELSE IF OperandSize = 32
        THEN
            Push(FrameTemp); FI; (* Doubleword push *)
   ELSE (* OperandSize = 16 *)
            Push(FrameTemp); (* Word push *)
FI;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
   THEN
            RBP ← FrameTemp;
            RSP ← RSP − Size;
   ELSE IF StackSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP − Size; FI;
   ELSE (* StackSize = 16 *)
            BP ← FrameTemp;
            SP ← SP − Size;
FI;

END;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.

Real-Address Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.

Virtual-8086 Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#SS(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.

35. Microcode update sections improved

In Sections 9.11.3 through 9.11.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* the information has been restructured and the supporting pseudocode has been improved. The applicable sections are reproduced below.

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. Software should not attempt to load a microsoft update where the signatures do not match.

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 9-10. 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 9-7. **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

Example 9-5 shows how to check for a processor signature and platform flags match between the processor and microcode update. The example assumes microcode authentication has been performed prior to attempting to load the microcode update and does not consider the revision of the microcode update.

In most cases, it is not desirable to load a microcode update with a microcode revision that is less than or equal to the processor's current microcode revision.

Example 9-5. Processor Signature and Platform Flags Comparison

```
ProcessorSignature ← CPUID.1:EAX
Flag ← 1 << IA32_PLATFORM_ID[52:50]
uCodeMatch ← FALSE

If (Update.HeaderVersion == 00000001H)
{
      //
      // Check for Processor Signature and Platform Id match
      //
```

```
        If (ProcessorSignature == Update.ProcessorSignature &&
            Flag & Update.ProcessorFlags)
        {
            Load Update
            uCodeMatch = TRUE
        }
        //
        // If the Processor Signature and Platform Id did not match,
        // check for the presence of an Extended Signature Table
        //
        Else If (Update.TotalSize > (Update.DataSize + 48))
        {
            //
            // Assume the Data Size has been used to calculate the
            // location of the Extended Signature Table
            // Update.ProcessorSignature[0] field
            //
            For (N ← 0; N < Update.ExtendedSignatureCount; N++)
            {
                //
                // Check Extended Signature Table Processor Signature
                // and Platform Id for a match.
                //
                If (ProcessorSignature == Update.ProcessorSignature[N] &&
                    Flag & Update.ProcessorFlags[N])
                {
                    Load Update
                    uCodeMatch = TRUE
                    Break
                }
            }
        }
        //
        // The microcode was not a match for the system,
        // a microcode update did not occur
        //
        If ( !uCodeMatch )
        {
            Fail
        }
}
```

36. Incorrect calls to CPUID.1:ECX[bit 9] have been corrected

In Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, incorrect calls to CPUID.1:ECX[bit 9] have been corrected to CPUID.1:ECX[bit 9]. See the table segments reproduced below. Only impacted table rows are reproduced.

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique	Bit Description
Hex	Dec				
...	...	...	...	...	...
9BH	155	IA32_SMM_MONITOR_CTL	3, 4, 6	Unique	SMM Monitor Configuration (R/W). (If CPUID.1:ECX[bit 5]=1 and in SMM)
...	...	....	....	....	....
480H	1152	IA32_VMX_BASIC	3, 4, 6	Unique	BASE Register of VMX Capability Reporting (R/O). (If CPUID.1:ECX[bit 5]=1)
481H	1153	IA32_VMX_PINBASED_CTLS	3, 4, 6	Unique	Capability Reporting Register of Pin-based VMCS Controls(R/O). (If CPUID.1:ECX[bit 5]=1)
482H	1154	IA32_VMX_ PROCBASED_CTLS	3, 4, 6	Unique	Capability Reporting Register of Processor-based VMCS Controls(R/O). (If CPUID.1:ECX[bit 5]=1)
483H	1155	IA32_VMX_EXIT_CTLS	3, 4, 6	Unique	Capability Reporting Register of VM-exit VMCS Controls(R/O). (If CPUID.1:ECX[bit 5]=1)
484H	1156	IA32_VMX_ENTRY_ CTLS	3, 4, 6	Unique	Capability Reporting Register of VM-entry VMCS Controls(R/O). (If CPUID.1:ECX[bit 5]=1)
485H	1157	IA32_VMX_MISC	3, 4, 6	Unique	Capability Reporting Register of Miscellaneous VMCS Controls(R/O). (If CPUID.1:ECX[bit 5]=1)
486H	1158	IA32_VMX_CR0_ FIXED0	3, 4, 6	Unique	Capability Reporting Register of CR0 Bits Fixed to Zero (R/O). (If CPUID.1:ECX[bit 5]=1)
487H	1159	IA32_VMX_CR0_ FIXED1	3, 4, 6	Unique	Capability Reporting Register of CR0 Bits Fixed to One (R/O). (If CPUID.1:ECX[bit 5]=1)
488H	1160	IA32_VMX_CR4_ FIXED0	3, 4, 6	Unique	Capability Reporting Register of CR4 Bits Fixed to Zero (R/O). (If CPUID.1:ECX[bit 5]=1)
489H	1161	IA32_VMX_CR4_ FIXED1	3, 4, 6	Unique	Capability Reporting Register of CR4 Bits Fixed to One(R/O). (If CPUID.1:ECX[bit 5]=1)
48AH	1162	IA32_VMX_VMCS_ ENUM	3, 4, 6	Unique	Capability Reporting Register of VMCS Field Enumeration (R/O). (If CPUID.1:ECX[bit 5]=1)

37. String operation and EFLAGS.RF interactions clarified

In Section 18.3.1.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* a paragraph has been updated to clarify string and EFLAGS.RF interactions. Part of the impacted section is reproduced below. See the change bars.

...Text omitted here....

18.3.1.1 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 (DB0 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 a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see “MOV—Move” and “POP—Pop a Value from the Stack” in Chapters 3 and 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A & 2B*).

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 code breakpoint, CS limit violation 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.

....Text omitted here....

38. MSR references corrected

In Section 20.6.8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* the MSR address references in the bullets. have been corrected. The section is reproduced below. See the change bar.

20.6.8 MSR-Bitmap Address

On processors that support the 1-setting of the "use MSR bitmaps" VM-execution control, the VM-execution control fields include the 64-bit physical address of four contiguous **MSR bitmaps**, which are each 1-KByte in size. This field does not exist on processors that do not support the 1-setting of that control. The four bitmaps are:

- **Read bitmap for low MSRs** (located at the MSR-bitmap address)**.** This contains one bit for each MSR address in the range 00000000H – 00001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Read bitmap for high MSRs** (located at the MSR-bitmap address plus 1024)**.** This contains one bit for each MSR address in the range C0000000H –C0001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Write bitmap for low MSRs** (located at the MSR-bitmap address plus 2048)**.** This contains one bit for each MSR address in the range 00000000H – 00001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.
- **Write bitmap for high MSRs** (located at the MSR-bitmap address plus 3072)**.** This contains one bit for each MSR address in the range C0000000H –C0001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

A logical processor uses these bitmaps if and only if the "use MSR bitmaps" control is 1. If the bitmaps are used, execution of an I/O instruction causes a VM exit if a bit in the I/O bitmaps corresponding to a port it accesses is 1. See Section 21.1.3 for details. If the bitmaps are used, their address must be 4-KByte aligned.

39. Description of VM-entry checks on VM-execution control fields updated

In Section 22.2.1.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, portions of text has been updated for clarity. The applicable section has been reproduced below. See the bullet with the change bar.

22.2.1.1 VM-Execution Control Fields

VM entries perform the following checks on the VM-execution control fields:

- Reserved bits in the pin-based VM-execution controls must be set properly. The reserved settings are indicated in Section 20.6.1. Software may consult the VMX capability MSR IA32_VMX_PINBASED_CTLS to determine the proper settings.
- Reserved bits in the processor-based VM-execution controls must be set properly. The reserved settings are indicated in Section 20.6.2. Software may consult the VMX capability MSR IA32_VMX_PROCBASED_CTLS to determine the proper settings (see Appendix G.2).
- The CR3-target count must not be greater than 4. Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC to determine the number of values supported (see Appendix G.5).
- If the "use I/O bitmaps" VM-execution control is 1, bits 11:0 of each I/O-bitmap address must be 0. On processors that support Intel 64 architecture, neither address should set any bits beyond the processor's physical-address width.[1] On processors that do not support Intel 64 architecture, neither address should set any bits in the range 63:32.
- If the "use TPR shadow" VM-execution control is 1, the virtual-APIC page address must satisfy the following checks:
  - — Bits 11:0 of each virtual-APIC page address must be 0.
  - — On processors that support the Intel 64 architecture, the address should not set any bits beyond the processor's physical-address width.
  - — On processors that do not support the Intel 64 architecture, the address should not set any bits in the range 63:32.

  If the "use TPR shadow" VM-execution control is 1 and all of the above checks are satisfied, bytes 129-131 on the page referenced by the virtual-APIC page address may be cleared by VM entry. This is true even if the VM entry fails. If the bytes are not cleared, they are left unmodified.
- If the "use MSR bitmaps" VM-execution control is 1, bits 11:0 of the MSR-bitmap address must be 0. On processors that support Intel 64 architecture, the address should not set any bits beyond the processor's physical-address width. On processors that do not support Intel 64 architecture, the address should not set any bits in the range 63:32.
- The following check is performed if the "use TPR shadow" VM-execution control is 1: the value of bits 3:0 of the TPR threshold should not be greater than the value of bits 7:4 in byte 128 on the page referenced by the virtual-APIC page address.

40. STI, MOVSS/POPSS blocking behavior clarified

In Chapter 4, "STI—Set Interrupt Flag", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B,* STI's blocking behavior has been emphasized. See the change bar on the reproduced text below.

STI—Set Interrupt Flag

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FB	STI	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

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.

Description

If protected-mode virtual interrupts are not enabled, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized[1]. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts **(and SMIs)** may be blocked for one macroinstruction following an STI.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

....Text omitted here....

In Section 23.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* a note has been updated to emphasize blocking behavior. See the change bar on the reproduced text below.

24.2 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 macroinstruction following an STI, MOVSS or POPSS.

1. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.
   In the following instruction sequence, interrupts may be recognized before RET executes:

```
STI
STI
RET
```

41. Correction for information on PEBS record size

In Section 18.15.5.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B;* PEBS record size was incorrectly indicated. This paragraph has been updated. The updated paragraph is reproduced below.

....Text omitted here....

When IA-32e mode is active, the structure of a branch trace record is similar to that shown in Figure 18-22, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 18-25). The structure of a PEBS record is similar to that shown in Figure 18-23, 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 18-26).

....Text omitted here....

42. Guest SMBASE entry added to table

In Table H-7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* the encoding for Guest SMBASE has been added. See the change bar on the reproduced table below.

Table H-7. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest ES limit	000000000B	00004800H
Guest CS limit	000000001B	00004802H
Guest SS limit	000000010B	00004804H
Guest DS limit	000000011B	00004806H
Guest FS limit	000000100B	00004808H
Guest GS limit	000000101B	0000480AH
Guest LDTR limit	000000110B	0000480CH
Guest TR limit	000000111B	0000480EH
Guest GDTR limit	000001000B	00004810H
Guest IDTR limit	000001001B	00004812H
Guest ES access rights	000001010B	00004814H
Guest CS access rights	000001011B	00004816H
Guest SS access rights	000001100B	00004818H
Guest DS access rights	000001101B	0000481AH
Guest FS access rights	000001110B	0000481CH
Guest GS access rights	000001111B	0000481EH
Guest LDTR access rights	000010000B	00004820H
Guest TR access rights	000010001B	00004822H
Guest interruptibility state	000010010B	00004824H
Guest activity state	000010011B	00004826H

Table H-7. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B) (Continued)

Field Name	Index	Encoding
Guest SMBASE	000010100B	00004828H
Guest IA32_SYSENTER_CS	000010101B	0000482AH