i960® Rx I/O Microprocessor Developer's Manual

Release Date: April, 1997

Order Number: 272736-002

The i960® Rx I/O Processor may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Such errata are not covered by Intel's warranty. Current characterized errata are available on request.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect IL 60056-7641

or call 1-800-879-4683
or visit Intel's website at http:\\www.intel.com

CHAPTER 1
INTRODUCTION

CHAPTER 2
DATA TYPES AND MEMORY ADDRESSING MODES

CHAPTER 3
PROGRAMMING ENVIRONMENT

CHAPTER 4
CACHE AND ON-CHIP DATA RAM

CHAPTER 5
INSTRUCTION SET OVERVIEW

CHAPTER 6
INSTRUCTION SET REFERENCE

CHAPTER 7 PROCEDURE CALLS

CHAPTER 8
INTERRUPTS

**CHAPTER 9
FAULTS**

CHAPTER 10
TRACING AND DEBUGGING

CHAPTER 11
INITIALIZATION AND SYSTEM REQUIREMENTS

CHAPTER 12
CORE PROCESSOR
LOCAL BUS CONFIGURATION

CHAPTER 13
LOCAL BUS

CHAPTER 14
MEMORY CONTROLLER

CHAPTER 15
PCI-TO-PCI BRIDGE UNIT

CHAPTER 16
ADDRESS TRANSLATION UNIT

CHAPTER 17
MESSAGING UNIT

CHAPTER 18
BUS ARBITRATION

CHAPTER 19
TIMERS

CHAPTER 20
DMA CONTROLLER

CHAPTER 21
I^{2}C BUS INTERFACE UNIT

CHAPTER 22
I/O APIC BUS INTERFACE UNIT

CHAPTER 23
TEST FEATURES

APPENDIX A
MACHINE-LEVEL INSTRUCTION FORMATS

APPENDIX B
OPCODES AND EXECUTION TIMES

APPENDIX C
MEMORY-MAPPED REGISTERS

FIGURES

TABLES

1

INTRODUCTION

CHAPTER 1
INTRODUCTION

1.1 INTEL'S i960® Rx I/O PROCESSOR

The i960 Rx I/O Processor integrates a high-performance 80960 "core" into a Peripheral Components Interconnect (PCI) functionality. This integrated processor addresses the needs of intelligent I/O applications and helps reduce intelligent I/O system costs. As indicated in Figure 1-1, the primary functional units include an i960 core processor, PCI to PCI bus bridge, PCI-to-80960 Address Translation Unit, Messaging Unit, Direct Memory Access (DMA) Controller, Memory Controller, Secondary PCI bus Arbitration Unit, I^{2}C Bus Interface Unit, and APIC Bus Interface Unit.

The PCI Bus is an industry standard, high performance, low latency system bus that operates up to 132 Mbyte/sec. The PCI-to-PCI bridge provides a connection path between two independent 32-bit PCI buses and provides the ability to overcome PCI electrical loading limits. The addition of the i960 core processor brings intelligence to the PCI bus bridge.

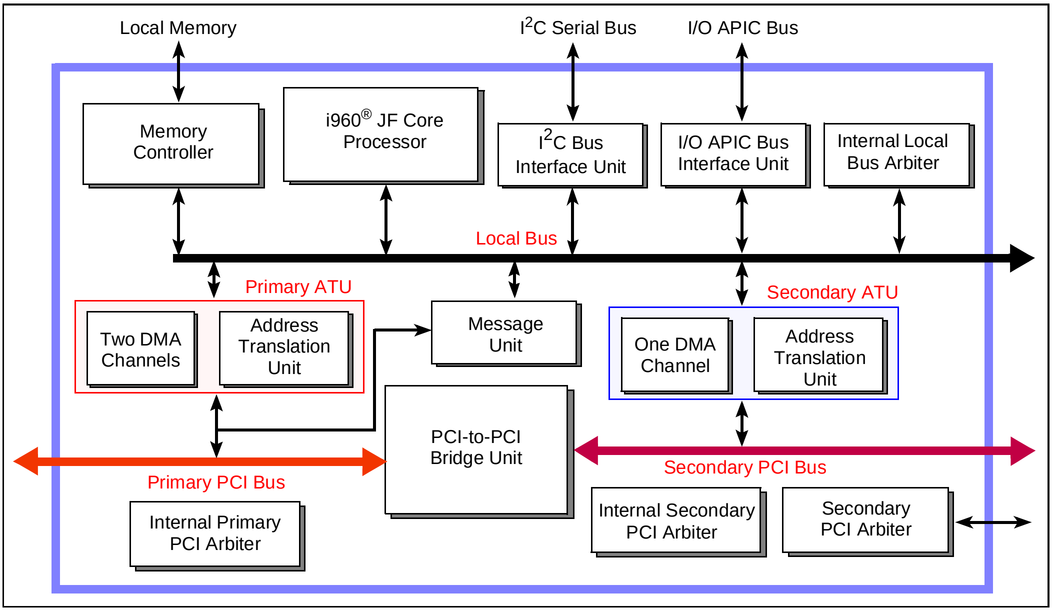


Figure 1-1. i960® Rx I/O Processor Functional Block Diagram

1.2 i960® Rx I/O PROCESSOR FEATURES

The i960 Rx I/O Processor (“80960Rx”) combines the i960® JF processor with powerful new features to create an intelligent I/O processor. This multi-function PCI device is fully compliant with the *PCI Local Bus Specification*, revision 2.1. 80960Rx-specific features include:

- Intelligent I/O (I_2O)
- PCI-to-PCI Bridge Unit
- Private PCI Device Support
- DMA Controller
- Address Translation Unit
- Memory Controller
- I^2C Bus Interface Unit
- I/O APIC Bus Interface Unit
- Secondary PCI Arbitration Unit
- Messaging Unit
- Wind River Systems IxWorks* RTOS Compatibility

Because the 80960Rx’s core processor is based upon the 80960JF, the two i960 family members are object code compatible and can maintain a sustained execution rate of one instruction per clock. The 80960 local bus, a 32-bit multiplexed burst bus, is a high-speed interface to system memory and I/O. A full complement of control signals simplifies the connection of the 80960Rx to external components. Physical and logical memory attributes are programmed via memory-mapped control registers (MMRs), a feature not found on the i960 Kx, Sx or Cx processors. Physical and logical configuration registers enable the processor to operate with all combinations of bus width and data object alignment. See section 1.3, i960® CORE PROCESSOR FEATURES (80960JF) (pg. 1-5) for more information.

The subsections that follow briefly overview each feature. Refer to the appropriate chapter for full technical descriptions.

1.2.1 Intelligent I/O (I_2O)

Addressing the software side of I/O, the i960 Rx I/O Processor supports the industry-standard Intelligent I/O (I_20) interface for PCI applications. This specification was formed by Intel and industry leaders in hardware and software to create a standard interface that increases I/O performance and decreases developer time-to-market. This specification provides a common I/O device driver that is independent to both the specific controlled device and the host operating system. The I_20 architecture facilitates intelligent I/O subsystems by supporting message passing between multiple independent processors. I_20 provides a standard interface to which all peripheral and network adapter card software can be developed, and remain compliant with popular network operating systems. The I_20 architecture improves performance by relieving the host of interrupt-intensive I/O tasks. By providing a standard interface, new technologies can be implemented quickly and uniformly.

1.2.2 PCI-to-PCI Bridge Unit

The PCI-to-PCI bridge unit (referred to as “bridge”) connects two independent PCI buses. It is fully compliant with the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0 published by the PCI Special Interest Group. It allows certain bus transactions on one PCI bus to be forwarded to the other PCI bus. It allows fully independent PCI bus operation (e.g., independent clocks). Dedicated data queues support high-performance bandwidth on the PCI buses. The 80960Rx supports PCI 64-bit Dual Address Cycle (DAC) addressing. The bridge has dedicated PCI configuration space that is accessible through the primary PCI bus. See CHAPTER 15, PCI-TO-PCI BRIDGE UNIT.

1.2.3 Private PCI Device Support

A key 80960Rx feature is that it explicitly supports private PCI devices on the secondary PCI bus without being detected by PCI configuration software. The bridge and Address Translation Unit work together to hide private devices from PCI configuration cycles and allow these devices to use a private PCI address space. The Address Translation Unit uses normal PCI configuration cycles to configure these devices.

1.2.4 DMA Controller

The DMA Controller allows low-latency, high-throughput data transfers between PCI bus agents and 80960 local memory. Three separate DMA channels accommodate data transfers: two for primary PCI bus, one for the secondary PCI bus. The DMA Controller supports chaining and unaligned data transfers. It is programmable through the i960 core processor only, and functions in synchronous mode only. See CHAPTER 20, DMA CONTROLLER.

1.2.5 Address Translation Unit

The Address Translation Unit (ATU) allows PCI transactions direct access to the 80960Rx local memory. The ATU supports transactions between PCI address space and 80960Rx address space. Address translation is controlled through programmable registers accessible from both the PCI interface and the i960 core processor. Dual access to registers allows flexibility in mapping the two address spaces. See CHAPTER 16, ADDRESS TRANSLATION UNIT.

1.2.6 Messaging Unit

The Messaging Unit (MU) provides data transfer between the PCI system and the 80960Rx. It uses interrupts to notify each system when new data arrives. The MU has four messaging mechanisms: Message Registers, Doorbell Registers, Circular Queues and Index Registers. Each allows a host processor or external PCI device and the 80960Rx to communicate through message passing and interrupt generation. See CHAPTER 17, MESSAGING UNIT.

1.2.7 Memory Controller

The Memory Controller allows direct control of external memory systems, including DRAM, SRAM, ROM and flash. It provides a direct connect interface to memory that typically does not require external logic. It features programmable chip selects, a wait state generator and byte parity. External memory can be configured as PCI addressable memory or private 80960Rx memory. See CHAPTER 14, MEMORY CONTROLLER.

1.2.8 I^{2}C Bus Interface Unit

The I^{2}C (Inter-Integrated Circuit) Bus Interface Unit allows the i960 core processor to serve as a master and slave device residing on the I^{2}C bus. The I^{2}C unit uses a serial bus developed by Philips Semiconductor* consisting of a two-pin interface. The bus allows the 80960Rx to interface to other I^{2}C peripherals and microcontrollers for system management functions. It requires a minimum of hardware for an economical system to relay status and reliability information on the I/O subsystem to an external device. See CHAPTER 21, I^{2}C BUS INTERFACE UNIT. Also refer to the document *I^{2}C Peripherals for Microcontrollers* (Philips Semiconductor).

1.2.9 I/O APIC Bus Interface Unit

The I/O APIC Bus Interface Unit provides an interface to the three-wire Advanced Programmable Interrupt Controller (APIC) bus that allows I/O APIC emulation in software. Interrupt messages can be sent on the bus and EOI messages can be received. See CHAPTER 22, I/O APIC BUS INTERFACE UNIT.

1.2.10 Secondary PCI Arbitration Unit

The Secondary PCI Arbitration Unit is the arbiter for the secondary PCI bus. It includes a fairness algorithm with programmable priorities and six PCI request and grant signal pairs. This arbitration unit can also be disabled to allow for external arbitration. See CHAPTER 18, BUS ARBITRATION.

1.2.11 Wind River Systems IxWorks* RTOS

A key feature of the i960 Rx I/O Processor is Wind River System’s IxWorks* Real-Time Operating System (RTOS). With clearly defined Application Program Interfaces (APIs), IxWorks creates a user-friendly environment to write basic device drivers. IxWorks supports NOS-to-driver independence, and allows multiple I/O software to co-exist reliably. In addition, developers get a 30-day evaluation copy of the Tornado* development environment. For more information, contact your local Intel representative.

1.3 i960® CORE PROCESSOR FEATURES (80960JF)

The processing power of the 80960Rx comes from the 80960JF processor core. The 80960JF is a new, scalar implementation of the i960 core architecture. Figure 1-2 shows a block diagram of the 80960JF core processor.

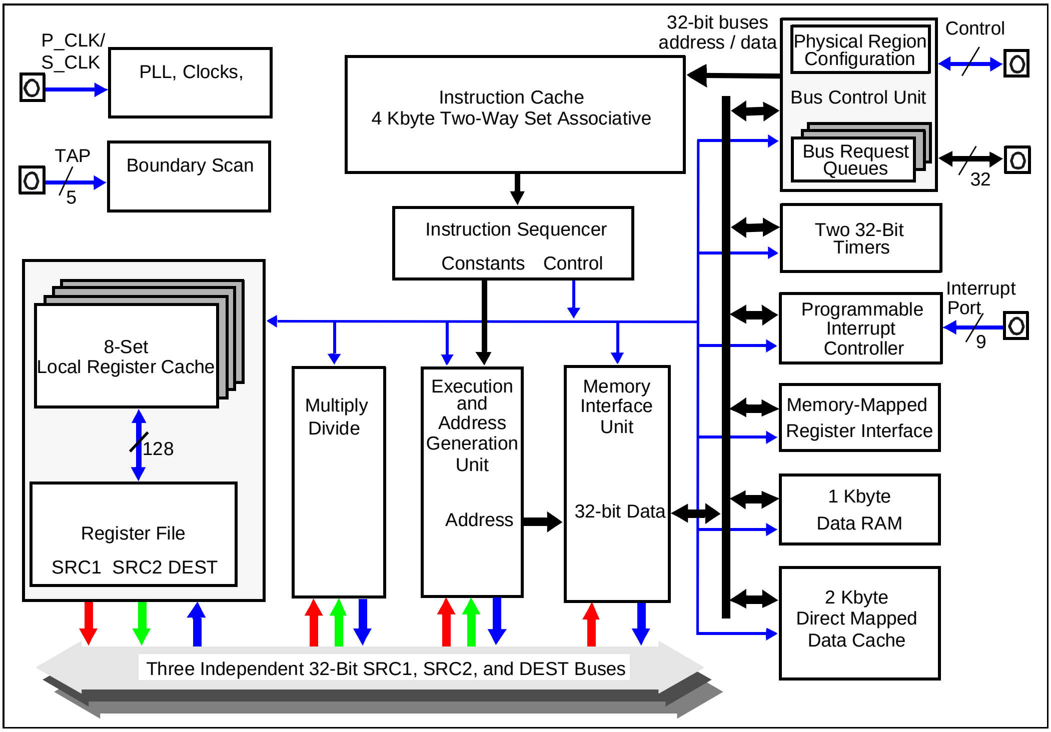


Figure 1-2. 80960JF Core Processor Block Diagram

Factors that contribute to the 80960JF's performance include:

- Single-clock execution of most instructions
- Independent Multiply/Divide Unit
- Efficient instruction pipeline minimizes pipeline break latency
- Register and resource scoreboarding allow overlapped instruction execution
- 128-bit register bus speeds local register caching
- 4 Kbyte two-way set-associative, integrated instruction cache
- 2 Kbyte direct-mapped, integrated data cache
- 1 Kbyte integrated data RAM delivers zero wait state program data

The i960 core processor operates out of its own 32-bit address space, which is independent of the PCI address space. The 80960 local bus memory can be:

- Made visible to the PCI address space
- Kept private to the i960 core processor
- Allocated as a combination of the two

1.3.1 Burst Bus

A 32-bit high-performance bus controller interfaces the i960 core processor to external memory and peripherals. The Bus Control Unit fetches instructions and transfers data on the 80960 local bus at the rate of up to four 32-bit words per six clock cycles.

NOTE: DMA and ATU accesses are limited to 32-bit wide memory regions. Also these units can burst up to a 2 Kbyte boundary with no alignment restrictions.

Users may configure the i960 core processor's bus controller to match an application's fundamental memory organization. Physical bus width is programmable for up to eight regions. Data caching is programmed through a group of logical memory templates and a defaults register. The Bus Control Unit's features include:

- Multiplexed external bus minimizes pin count
- 32-, 16- and 8-bit bus widths simplify I/O interfaces
- External ready control for address-to-data, data-to-data and data-to-next-address wait state types
- Unaligned bus accesses performed transparently
- Three-deep load/store queue decouples the bus from the i960 core processor

For reliability, the 80960Rx conducts an internal self test upon reset. Before executing its first instruction, it performs a local bus confidence test by performing a checksum on the first words of the Initialization Boot Record.

1.3.2 Timer Unit

As described in CHAPTER 19, TIMERS, The Timer Unit (TU) contains two independent 32-bit timers that are capable of counting at software-defined clock rates and generating interrupts. Each is programmed by use of the Timer Unit memory-mapped registers. The timers have a single-shot mode and auto-reload capabilities for continuous operation. Each timer has an independent interrupt request to the 80960Rx's interrupt controller.

1.3.3 Priority Interrupt Controller

CHAPTER 8, INTERRUPTS explains how low interrupt latency is critical to many embedded applications. As part of its highly flexible interrupt mechanism, the 80960Rx exploits several techniques to minimize latency:

- Interrupt vectors and interrupt handler routines can be reserved on-chip
- Register frames for high-priority interrupt handlers can be cached on-chip
- The interrupt stack can be placed in cacheable memory space

1.3.4 Faults and Debugging

The 80960Rx employs a comprehensive fault model. The processor responds to faults by making implicit calls to fault handling routines. Specific information collected for each fault allows the fault handler to diagnose exceptions and recover appropriately.

The processor also has built-in debug capabilities. Via software, the 80960Rx may be configured to detect as many as seven different trace event types. Alternatively, **mark** and **fmark** instructions can generate trace events explicitly in the instruction stream. Hardware breakpoint registers are also available to trap on execution and data addresses. See CHAPTER 9, FAULTS.

1.3.5 On-Chip Cache and Data RAM

As discussed in CHAPTER 4, CACHE AND ON-CHIP DATA RAM, memory subsystems often impose substantial wait state penalties. The 80960Rx integrates considerable storage resources on-chip to decouple CPU execution from the external bus. The 80960Rx includes a 4 Kbyte instruction cache, a 2 Kbyte data cache and 1 Kbyte data RAM.

1.3.6 Local Register Cache

The 80960Rx rapidly allocates and deallocates local register sets during context switches. The processor needs to flush a register set to the stack only when it saves more than seven sets to its local register cache.

1.3.7 Test Features

The 80960Rx incorporates features that enhance the user’s ability to test both the processor and the system to which it is attached. These features include ONCE (On-Circuit Emulation) mode and IEEE Std. 1149.1 Boundary Scan (JTAG). See CHAPTER 23, TEST FEATURES.

One of the boundary scan instructions, **HIGHZ**, forces the processor to float all its output pins (ONCE mode). ONCE mode can also be initiated at reset without using the boundary scan mechanism.

ONCE mode is useful for board-level testing. This feature allows a mounted 80960Rx to electrically "remove" itself from a circuit board. This mode allows system-level testing where a remote tester, such as an In-Circuit Emulator (ICE) system, can exercise the processor system. The test logic does not interfere with component or system behavior and ensures that components function correctly, and also the connections between various components are correct.

The JTAG Boundary Scan feature is an alternative to conventional "bed-of-nails" testing. It can examine connections that might otherwise be inaccessible to a test system.

1.3.8 Memory-Mapped Control Registers

The 80960Rx is compliant with 80960 family architecture and has the added advantage of memory-mapped, internal control registers not found on the 80960Kx, Sx or Cx processors. This feature provides software an interface to easily read and modify internal control registers.

Each memory-mapped, 32-bit register is accessed via regular memory-format instructions. The processor ensures that these accesses do not generate external bus cycles. See CHAPTER 14, MEMORY CONTROLLER.

1.3.9 Instructions, Data Types and Memory Addressing Modes

As with all 80960 family processors, the 80960Rx instruction set supports several different data types and formats:

- Bit
- Bit fields
- Integer (8-, 16-, 32-, 64-bit)
- Ordinal (8-, 16-, 32-, 64-bit unsigned integers)
- Triple word (96 bits)
- Quad word (128 bits)

Several chapters describe the i960 Rx I/O Processor instruction set, including:

- CHAPTER 3, PROGRAMMING ENVIRONMENT
- CHAPTER 5, INSTRUCTION SET OVERVIEW
- CHAPTER 6, INSTRUCTION SET REFERENCE

1.4 ABOUT THIS DOCUMENT

The i960 Rx I/O Processor incorporates Peripheral Component Interconnect (PCI) functionality with the i960 JF processor. As such, it is assumed that the reader has a working understanding of the Peripheral Component Interconnect (PCI), *PCI Local Bus Specification*, revision 2.1, and the i960 core processor.

1.4.1 Terminology

In this document, the following terms are used:

- *80960Rx* refers generically to the i960 Rx I/O Processor family. As of this printing, the family includes the 80960RP 33/5.0, 80960RP 33/3.3, 80960RD 66/3.3.
- *80960 local bus* refers to the i960 Rx I/O Processor's internal local bus, not the PCI local bus.
- *Primary and Secondary PCI buses* are the i960 Rx I/O Processor's internal PCI buses that conform to PCI SIG specifications.
- *i960 core processor* refers to the i960 JF processor that is integrated into the 80960Rx.
- *DWORD* is a 32-bit data word.
- *80960 Local memory* is a memory subsystem on the 80960 processor local bus.

The following terms are used primarily in CHAPTER 15, PCI-TO-PCI BRIDGE UNIT:

- *Downstream* — at or toward a PCI bus with a higher number (after configuration).
- *Host processor* — Processor located upstream from the i960 Rx I/O Processor.
- *Local processor* — i960 core processor within the i960 Rx I/O Processor.
- *Upstream* — At or toward a PCI bus with a lower number (after configuration).

1.4.2 Representing Numbers

Assume that all numbers are base 10 unless designated otherwise. In text, numbers in base 16 are represented as “nnnH”, where the “H” signifies hexadecimal. In pseudocode descriptions, hexadecimal numbers are represented in the form 0x1234ABCD. Binary numbers are not explicitly identified and are assumed when bit operations or bit ranges are used.

1.4.3 Fields

A *preserved* field in a data structure is one that the processor does not use. Preserved fields can be used by software; the processor does not modify such fields.

A *reserved* field is a field that may be used by an implementation. When the initial value of a reserved field is supplied by software, this value must be zero. Software should not modify reserved fields or depend on any values in reserved fields.

A *read only* field can be read to return the current value. Writes to *read only* fields are treated as no-op operations and do not change the current value or result in an error condition.

A *read/clear* field can also be read to return the current value. A write to a *read/clear* field with the data value of 0 causes no change to the field. A write to a *read/clear* field with a data value of 1 causes the field to be cleared (reset to the value of 0). For example, when a *read/clear* field has a value of F0H, and a data value of 55H is written, the resultant field is A0H.

A *read/set* field can also be read to return the current value. A write to a *read/set* field with the data value of 0 causes no change to the field. A write to a *read/set* field with a data value of 1 causes the field to be set (set to the value of 1). For example, when a *read/set* field has a value of F0H, and a data value of 55H is written, the resultant field is F5H.

1.4.4 Specifying Bit and Signal Values

The terms *set* and *clear* in this specification refer to bit values in register and data structures. When a bit is set, its value is 1; when the bit is clear, its value is 0. Likewise, *setting* a bit means giving it a value of 1 and *clearing* a bit means giving it a value of 0.

The terms *assert* and *deassert* refer to the logically active or inactive value of a signal or bit, respectively.

1.4.5 Signal Name Conventions

All signal names use the PCI signal name convention of using the “#” symbol at the end of a signal name to indicate that the signal’s active state occurs when it is at a low voltage. This includes 80960 processor-related signal names that normally use an $\overline{\text{overline}}$. The absence of the “#” symbol indicates that the signal’s active state occurs when it is at a high voltage.

1.4.6 *Solutions960®* Program

Intel’s *Solutions960*® program features a wide variety of development tools that support the i960 processor family. Many of these tools are developed by partner companies; some are developed by Intel, such as profile-driven optimizing compilers. For more information on these products, contact your local Intel representative.

1.4.7 Additional Information Sources

Intel documentation is available from your Intel Sales Representative or Intel Literature Sales.

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect IL 60056-7641
1-800-548-4725

Document Title	Order / Contact
i960® Rx I/O Processor Specification Update	Intel Order # 272918
i960® RP I/O Processor (33 MHz, 5.0 Volt version) data sheet	Intel Order # 272737
i960® RP/RD I/O Processor at 3.3 Volts data sheet	Intel Order # 273001
i960 RP Processor: A Single-Chip Intelligent I/O Subsystem Technical Brief	Intel Order # 272738
i960® Jx Microprocessor User's Guide	Intel Order # 272483
MultiProcessor Specification	Intel Order # 242016
PCI Local Bus Specification, revision 2.1	PCI Special Interest Group 1-800-433-5177
PCI-to-PCI Bridge Architecture Specification Revision 1.0	PCI Special Interest Group 1-800-433-5177
PCI System Design Guide, Revision 1.0	PCI Special Interest Group 1-800-433-5177
I^{2}C Peripherals for Microcontrollers	Philips Semiconductor
I^{2}C *Bus and How to Use It* (Including Specifications)	Philips Semiconductor
I^{2}C *Peripherals for Microcontrollers* (Including Fast Mode)	Signetics

1.4.8 Electronic Information

Intel's World-Wide Web Home Page	http://www.intel.com/
Wind River System's IxWorks	http://www.wrs.com/
I$_2$0 Special Interest Group Web Site	http://www.i2osig.org/

1.5 STEPPING DIFFERENCES SUMMARY

Table 1-1 identifies the sections in this document which contain information that is specific to an individual stepping.

Table 1-1. Stepping Differences Summary

Section	Description	Page
8.3	Interrupt Controller Connections for 80960RP 33/5.0 Volt	8-23
8.3	Interrupt Controller Connections for 80960Rx 33/3.3 Volt	8-24
8.3.2	PCI Interrupt Routing Summary for 80960RP 33/5.0 Volt	8-26
8.3.2	PCI Interrupt Routing Summary for 80960RP 33/3.3 Volt	8-26
8.4.1	PCI Interrupt Routing Select Register – PIRSR (80960RP 33/5.0 Volt)	8-32
8.4.1	PCI Interrupt Routing Select Register – PIRSR (80960Rx 33/3.3 Volt)	8-32
15.4.1	Private Configuration Commands (Type 0) on the Secondary Interface	15-7
15.4.2	Private PCI Memory IDSEL Select Configurations	15-8
15.13.25	Secondary IDSEL Select Register - SISR	15-66
16.7.12	Determining Block Sizes for Base Address Registers	16-37

2

DATA TYPES AND MEMORY ADDRESSING MODES

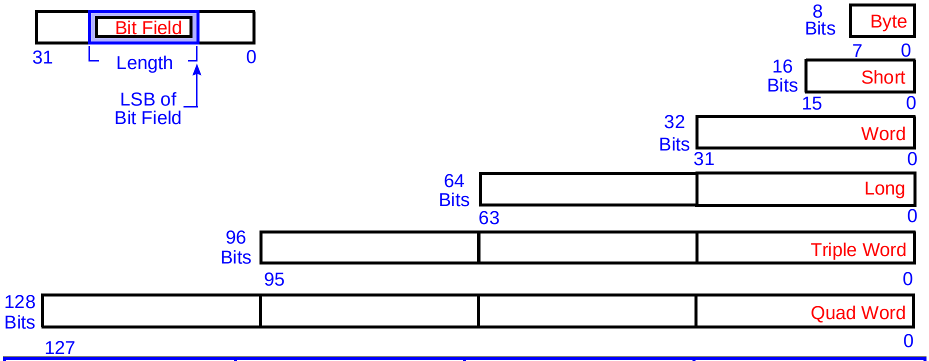
CHAPTER 2
DATA TYPES AND MEMORY ADDRESSING MODES

2.1 DATA TYPES

The instruction set references or produces several data lengths and formats. The i960® Rx I/O processor supports the following data types:

- Integer (signed 8, 16 and 32 bits)
- Long Word (64 bits)
- Quad Word (128 bits)
- Bit
- Ordinal (unsigned integer 8, 16, and 32 bits)
- Triple Word (96 bits)
- Bit Field

Figure 2-1 illustrates the class, data type and length of each type supported by i960 processors.



Class	Data Type	Length	Range
Numeric (Integer)	Byte Integer Short Integer Integer	8 Bits 16 Bits 32 Bits	-2^{7} to 2^{7} -1 -2^{15} to 2^{15} -1 -2^{31} to 2^{31} -1
Numeric (Ordinal)	Byte Ordinal Short Ordinal Ordinal Long Ordinal	8 Bits 16 Bits 32 Bits 64 Bits	0 to 2^{8} -1 0 to 2^{16} -1 0 to 2^{32} -1 0 to 2^{64} - 1
Non-Numeric	Bit Bit Field Long Word Triple Word Quad Word	1 Bit 1-32 Bits 64 Bits 96 Bits 128 Bits	N/A

Figure 2-1. Data Types and Ranges

2.1.1 Word/Dword Notation

Data lengths, as described in the PCI Local Bus Specification, Revision 2.1, differs from the conventions used for the 80960 architecture. See also Table 2-1:

- In the PCI specification the term *word* refers to a 16-bit block of data.
- In this manual and other documentation relating to the i960 Rx I/O processor, the term *word* refers to a 32-bit block of data.

Table 2-1. 80960 and PCI Architecture Data Word Notation Differences

No. of Bits	PCI Architecture	80960 Architecture
16	**word**	short word or half word
32	doubleword or dword	**word**

2.1.2 Integers

Integers are signed whole numbers that are stored and operated on in two’s complement format by the integer instructions. Most integer instructions operate on 32-bit integers. Byte and short integers are referenced by the byte and short classes of the load, store and compare instructions only.

Integer load or store size (byte, short or word) determines how sign extension or data truncation is performed when data is moved between registers and memory.

For instructions **ldib** (load integer byte) and **ldis** (load integer short), a byte or short word in memory is considered a two’s complement value. The value is sign-extended and placed in the 32-bit register that is the destination for the load.

```
    ldib
            7AH is loaded into a register as 0000 007AH
            FAH is loaded into a register as FFFF FFFAH
    ldis
            05A5H is loaded into a register as 0000 05A5H
            85A5H is loaded into a register as FFFF 85A5H
```

Example 2-1. Sign Extensions on Load Byte and Load Short

For instructions **stib** (store integer byte) and **stis** (store integer short), a 32-bit two's complement number in a register is stored to memory as a byte or short word. When register data is too large to be stored as a byte or short word, the value is truncated and the integer overflow condition is signalled. When an overflow occurs, either an AC register flag is set or the ARITHMETIC.INTEGER_OVERFLOW fault is generated, depending on the Integer Overflow Mask bit (AC.om) in the AC register. CHAPTER 9, FAULTS describes the integer overflow fault.

For instructions **ld** (load word) and **st** (store word), data is moved directly between memory and a register with no sign extension or data truncation.

2.1.3 Ordinals

Ordinals or unsigned integer data types are stored and treated as positive binary values. Figure 2-1 shows the supported ordinal sizes.

The large number of instructions that perform logical, bit manipulation and unsigned arithmetic operations reference 32-bit ordinal operands. When ordinals are used to represent Boolean values, 1 = TRUE and 0 = FALSE. Most extended arithmetic instructions reference the long ordinal data type. Only load (**ldob** and **ldos**), store (**stob** and **stos**), and compare ordinal instructions reference the byte and short ordinal data types.

Sign and sign extension are not considered when ordinal loads and stores are performed; however, the values may be zero-extended or truncated. A short word or byte load to a register causes the value loaded to be zero-extended to 32 bits. A short word or byte store to memory truncates an ordinal value in a register to fit the destination memory. No overflow condition is signalled in this case.

2.1.4 Bits and Bit Fields

The processor provides several instructions that perform operations on individual bits or bit fields within register operands. An individual bit is specified for a bit operation by giving its bit number and register. Internal registers always follow little endian byte order; the least significant bit is bit 0 and the most significant bit is bit 31.

A bit field is any contiguous group of bits (up to 32 bits long) in a 32-bit register. Bit fields do not span register boundaries. A bit field is defined by giving its length in bits (1-32) and the bit number of its lowest numbered bit (0-31).

Loading and storing bit and bit-field data is normally performed using the ordinal load (**ldo**) and store (**sto**) instructions. When an **ldi** instruction loads a bit or bit field value into a 32-bit register, the processor appends sign extension bits. A byte or short store can signal an integer overflow condition.

2.1.5 Triple and Quad Words

Triple and quad words refer to consecutive words in memory or in registers. Triple- and quad-word load, store and move instructions use these data types to accomplish block movements. No data manipulation (sign extension, zero extension or truncation) is performed in these instructions.

Triple- and quad-word data types can be considered a superset of the other data types described. Data in each word subset of a quad word is likely to be the operand or result of an ordinal, integer, bit or bit field instruction.

2.1.6 Register Data Alignment

Several instructions operate on multiple-word operands. For example, the load-long instruction (**ldl**) loads two words from memory into two consecutive registers. Here the register number for the least significant word is automatically loaded into the next higher-numbered register.

In cases where an instruction specifies a register number, and multiple, consecutive registers are implied, the register number must be even if two registers are accessed (e.g., g0, g2) and an integral multiple of four if three or four registers are accessed (e.g., g0, g4). When a register reference for a source value is not properly aligned, the registers that the processor writes to are undefined.

The i960 Rx I/O processor does not require data alignment in external memory; the processor hardware handles unaligned memory accesses automatically. Optionally, user software can configure the processor to generate a fault on unaligned memory accesses.

2.1.7 Literals

The architecture defines a set of 32 literals that can be used as operands in many instructions. These literals are ordinal (unsigned) values that range from 0 to 31 (5 bits). When a literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction requires an operand larger than 32 bits, the processor zero-extends the value to the operand size. If a literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

2.2 BIT AND BYTE ORDERING IN MEMORY

All occurrences of numeric and non-numeric data types, except bits and bit fields, must start on a byte boundary. Any data item occupying multiple bytes is stored as little endian.

2.3 MEMORY ADDRESSING MODES

Nine modes are available for addressing operands in memory. Each addressing mode is used to reference a byte location in the processor's address space. Table 2-2 shows the memory addressing modes and a brief description of each mode's address elements and assembly code syntax.

Table 2-2. Memory Addressing Modes

Mode	Description	Assembler Syntax	Inst. Type
Absolute *offset*	offset (smaller than 4096)	exp	MEMA
displacement	displacement (larger than 4095)	exp	MEMB
Register Indirect	abase	(reg)	MEMB
with offset	abase + offset	exp (reg)	MEMA
with displacement	abase + displacement	exp (reg)	MEMB
with index	abase + (index*scale)	(reg) [reg*scale]	MEMB
with index and displacement	abase + (index*scale) + displacement	exp (reg) [reg*scale]	MEMB
Index with displacement	(index*scale) + displacement	exp [reg*scale]	MEMB
instruction pointer (IP) with displacement	IP + displacement + 8	exp (IP)	MEMB

NOTE: *reg* is register, *exp* is an expression or symbolic label, and IP is the Instruction Pointer.

See Table B-9., MEM Format Instruction Encodings (pg. B-9) for more on addressing modes. For purposes of this memory addressing modes description, MEMA format instructions require one word of memory and MEMB usually require two words and therefore consume twice the bus bandwidth to read. Otherwise, both formats perform the same functions.

2.3.1 Absolute

Absolute addressing modes allow a memory location to be referenced directly as an offset from address 0H. At the instruction encoding level, two absolute addressing modes are provided: absolute offset and absolute displacement, depending on offset size.

- For the absolute offset addressing mode, the offset is an ordinal number ranging from 0 to 4095. The absolute offset addressing mode is encoded in the MEMA machine instruction format.
- For the absolute displacement addressing mode, the offset value ranges from 0 to 2^{32}-1. The absolute displacement addressing mode is encoded in the MEMB format.

Addressing modes and encoding instruction formats are described in CHAPTER 6, INSTRUCTION SET REFERENCE.

At the assembly language level, the two absolute addressing modes use the same syntax. Typically, development tools allow absolute addresses to be specified through arithmetic expressions (e.g., x + 44) or symbolic labels. After evaluating an address specified with the absolute addressing mode, the assembler converts the address into an offset or displacement and selects the appropriate instruction encoding format and addressing mode.

2.3.2 Register Indirect

Register indirect addressing modes use a register's 32-bit value as a base for address calculation. The register value is referred to as the address base (designated "abase" in Table 2-2). Depending on the addressing mode, an optional scaled index and offset can be added to this address base.

Register indirect addressing modes are useful for addressing elements of an array or record structure. When addressing array elements, the abase value provides the address of the first array element. An offset (or displacement) selects a particular array element.

In register-indirect-with-index addressing mode, the index is specified using a value contained in a register. This index value is multiplied by a scale factor. Allowable factors are 1, 2, 4, 8 and 16. The register-indirect-with-index addressing mode is encoded in the MEMA format.

The two versions of register-indirect-with-offset addressing mode at the instruction encoding level are register-indirect-with-offset and register-indirect-with-displacement. As with absolute addressing modes, the mode selected depends on the size of the offset from the base address.

At the assembly language level, the assembler allows the offset to be specified with an expression or symbolic label, then evaluates the address to determine whether to use register-indirect-with-offset (MEMA format) or register-indirect-with-displacement (MEMB format) addressing mode.

Register-indirect-with-index-and-displacement addressing mode adds both a scaled index and a displacement to the address base. There is only one version of this addressing mode at the instruction encoding level, and it is encoded in the MEMB instruction format.

2.3.3 Index with Displacement

A scaled index can also be used with a displacement alone. Again, the index is contained in a register and multiplied by a scaling constant before displacement is added. This mode uses MEMB format.

2.3.4 IP with Displacement

This addressing mode is used with load and store instructions to make them instruction pointer (IP) relative. IP-with-displacement addressing mode references the next instruction's address plus the displacement plus a constant of 8. The constant is added because, in a typical processor implementation, the address has incremented beyond the next instruction address at the time of address calculation. The constant simplifies IP-with-displacement addressing mode implementation. This mode uses MEMB format.

2.3.5 Addressing Mode Examples

The following examples show how i960 processor addressing modes are encoded in assembly language. Example 2-2 shows addressing mode mnemonics. Example 2-3 illustrates the usefulness of scaled index and scaled index plus displacement addressing modes. In this example, a procedure named array_op uses these addressing modes to fill two contiguous memory blocks separated by a constant offset. A pointer to the top of the block is passed to the procedure in g0, the block size is passed in g1 and the fill data in g2. Refer to APPENDIX A, MACHINE-LEVEL INSTRUCTION FORMATS.

```
st   g4,xyz           # Absolute; word from g4 stored at memory
                      # location designated with label xyz.
ldob (r3),r4          # Register indirect; ordinal byte from
                      # memory location given in r3 loaded
                      # into register r4 and zero extended.
stl  g6,xyz(g5)       # Register indirect with displacement;
                      # double word from g6,g7 stored at memory
                      # location xyz + g5.
ldq  (r8)[r9*4],r4    # Register indirect with index; quad-word
                      # beginning at memory location r8 + (r9
                      # scaled by 4) loaded into r4 through r7.
st   g3,xyz(g4)[g5*2]# Register indirect with index and
                      # displacement; word in g3 stored to mem
                      # location g4 + xyz + (g5 scaled by 2).
ldis xyz[r12*2],r13   # Index with displacement; load short
                      # integer at memory location xyz + r12
                      # into r13 and sign extended.
st   r4,xyz(IP)       # IP with displacement; store word in r4
                      # at memory location IP + xyz + 8.
```

Example 2-2. Addressing Mode Mnemonics

```
array_op:
      mov   g0,r4             # Pointer to array is copied to r4.
      subi  1,g1,r3           # Calculate index for the last array
      b     .I33              # element to be filled
.I34:
      st    g2,(r4)[r3*4]     # Fill element at index
      st    g2,0x30(r4)[r3*4] # Fill element at index+constant offset
      subi  1,r3,r3           # Decrement index
.I33:
      cmpible  0,r3,.I34      # Store next array elements if
      ret                     # index is not 0
```

Example 2-3. Scaled Index and Scaled Index Plus Displacement Addressing Modes

3

PROGRAMMING ENVIRONMENT

CHAPTER 3
PROGRAMMING ENVIRONMENT

This chapter describes the i960® Rx I/O processor's programming environment including global and local registers, control registers, literals, processor-state registers and address space.

3.1 OVERVIEW

The i960 architecture defines a programming environment for program execution, data storage and data manipulation. Figure 3-1 shows the programming environment elements that include a 4 Gbyte (2^{32} byte) flat address space, an instruction cache, a data cache, global and local general-purpose registers, a register cache, a set of literals, control registers and a set of processor state registers.

The processor includes several architecturally-defined data structures located in memory as part of the programming environment. These data structures handle procedure calls, interrupts and faults and provide configuration information at initialization. These data structures are:

- interrupt stack
- local stack
- supervisor stack
- control table
- fault table
- interrupt table
- system procedure table
- process control block
- initialization boot record

3.2 REGISTERS AND LITERALS AS INSTRUCTION OPERANDS

With the exception of a few special instructions, the i960 Rx I/O processor uses only simple load and store instructions to access memory. All operations take place at the register level. The processor uses 16 global registers, 16 local registers and 32 literals (constants 0-31) as instruction operands.

The global register numbers are g0 through g15; local register numbers are r0 through r15. Several of these registers are used for dedicated functions. For example, register r0 is the previous frame pointer, often referred to as *pfp*. i960 processor compilers and assemblers recognize only the instruction operands listed in Table 3-1. Throughout this manual, the registers' descriptive names, numbers, operands and acronyms are used interchangeably, as dictated by context.

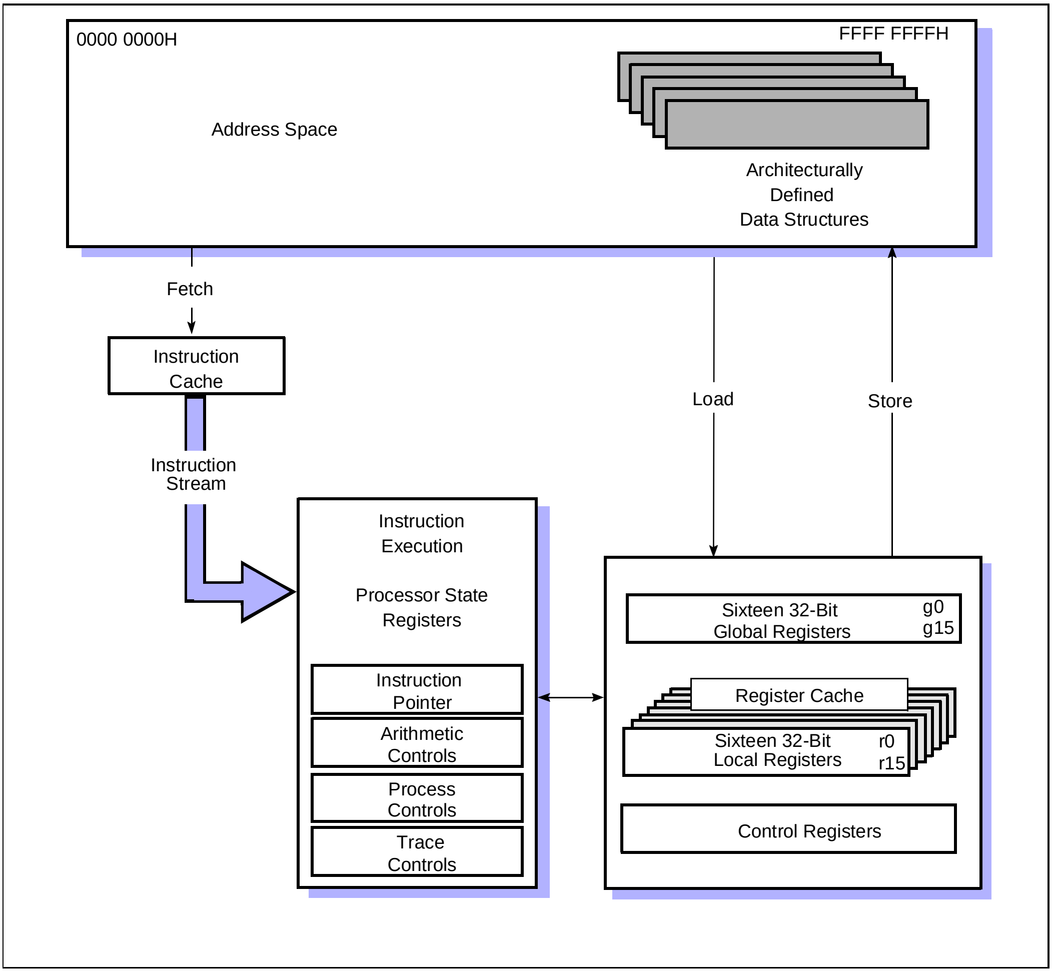


Figure 3-1. i960® Rx I/O Processor Programming Environment

3.2.1 Global Registers

Global registers are general-purpose 32-bit data registers that provide temporary storage for a program's computational operands. These registers retain their contents across procedure boundaries. As such, they provide a fast and efficient means of passing parameters between procedures.

Table 3-1. Registers and Literals Used as Instruction Operands

Instruction Operand	Register Name (number)	Function	Acronym
g0 - g14	global (g0-g14)	general purpose	
fp	global (g15)	frame pointer	FP
pfp	local (r0)	previous frame pointer	PFP
sp	local (r1)	stack pointer	SP
rip	local (r2)	return instruction pointer	RIP
r3 - r15	local (r3-r15)	general purpose	
0-31		literals	

The i960 architecture supplies 16 global registers, designated g0 through g15. Register g15 is reserved for the current Frame Pointer (FP), which contains the address of the first byte in the current (topmost) stack frame in internal memory. See section 7.1, CALL AND RETURN MECHANISM (pg. 7-2) for a description of the FP and procedure stack.

After the processor is reset, register g0 contains the i960 core processor device identification and stepping information. g0 retains this information until it is written over by the user program. The i960 core processor device identification and stepping information is also stored in the memory-mapped DEVICEID register located at FF00 8710H. In addition, the i960 Rx I/O processor device identification and stepping information is stored in the memory-mapped register located at 0000 17C0H.

3.2.2 Local Registers

The i960 architecture provides a separate set of 32-bit local data registers (r0 through r15) for each active procedure. These registers provide storage for variables that are local to a procedure. Each time a procedure is called, the processor allocates a new set of local registers and saves the calling procedure's local registers. When the application returns from the procedure, the local registers are released for the next procedure call. The processor performs local register management; a program need not explicitly save and restore these registers.

r3 through r15 are general purpose registers; r0 through r2 are reserved for special functions; r0 contains the Previous Frame Pointer (PFP); r1 contains the Stack Pointer (SP); r2 contains the Return Instruction Pointer (RIP). These are discussed in CHAPTER 7, PROCEDURE CALLS.

The processor does not always clear or initialize the set of local registers assigned to a new procedure. Also, the processor does not initialize the local register save area in the newly created stack frame for the procedure. User software should not rely on the initial values of local registers.

3.2.3 Register Scoreboarding

Register scoreboarding maintains register coherency by preventing parallel execution units from accessing registers for which there is an outstanding operation. When an instruction that targets a destination register or group of registers executes, the processor sets a register-scoreboard bit to indicate that this register or group of registers is being used in an operation. If the instructions that follow do not require data from registers already in use, the processor can execute those instructions before the prior instruction execution completes.

Software can use this feature to execute one or more single-cycle instructions concurrently with a multi-cycle instruction (e.g., multiply or divide). Example 3-1 shows a case where register scoreboarding prevents a subsequent instruction from executing. It also illustrates overlapping instructions that do not have register dependencies.

Example 3-1. Register Scoreboarding

```
muli r4,r5,r6   # r6 is scoreboarded
addi r6,r7,r8   # addi must wait for the previous multiply
     .          # to complete
     .
     .
muli r4,r5,r10  # r10 is scoreboarded
and  r6,r7,r8   # and instruction is executed concurrently with
```

3.2.4 Literals

The architecture defines a set of 32 literals that can be used as operands in many instructions. These literals are ordinal (unsigned) values that range from 0 to 31 (5 bits). When a literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction requires an operand larger than 32 bits, the processor zero-extends the value to the operand size. If a literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

3.2.5 Register and Literal Addressing and Alignment

Several instructions operate on multiple-word operands. For example, the load long instruction (**ldl**) loads two words from memory into two consecutive registers. The register for the less significant word is specified in the instruction. The more significant word is automatically loaded into the next higher-numbered register.

In cases where an instruction specifies a register number and multiple consecutive registers are implied, the register number must be even if two registers are accessed (e.g., g0, g2) and an integral multiple of 4 if three or four registers are accessed (e.g., g0, g4). If a register reference for a source value is not properly aligned, the source value is undefined and an OPERATION.INVALID_OPERAND fault is generated. If a register reference for a destination value is not properly aligned, the registers to which the processor writes and the values written are undefined. The processor then generates an OPERATION.INVALID_OPERAND fault. The assembly language code in Example 3-2 shows an example of correct and incorrect register alignment.

Example 3-2. Register Alignment

```
movl  g3,g8      # Incorrect alignment - resulting value
     .           # in registers g8 and g9 is
     .           # unpredictable (non-aligned source)
     .
movl  g4,g8      # Correct alignment
```

Global registers, local registers and literals are used directly as instruction operands. Table 3-2 lists instruction operands for each machine-level instruction format and the positions that can be filled by each register or literal.

Table 3-2. Allowable Register Operands

		Operand[1]		
Instruction Encoding	**Operand Field**	**Local Register**	**Global Register**	**Literal**
REG	*src1*	X	X	X
	src2	X	X	X
	src/dst (as *src*)	X	X	X
	src/dst (as *dst*)	X	X	
	src/dst (as both)	X	X	
MEM	*src/dst*	X	X	
	abase	X	X	
	index	X	X	
COBR	*src1*	X	X	
	src2	X	X	X
	dst	X[2]	X[2]	X[2]

NOTES:

1. "X" denotes the register can be used as an operand in a particular instruction field.
2. The **COBR** destination operands apply only to **TEST** instructions.

3.3 MEMORY-MAPPED CONTROL REGISTERS (MMRs)

The i960 Rx I/O processor gives software the interface to easily read and modify internal control registers. Each of these registers is accessed as a memory-mapped register with a unique memory address. There are two distinct sets of memory-mapped registers on the 80960Rx. The first set exists in the FF00 0000H through FFFF FFFFH address range and is used to control the i960 core processor functions. The second set exists in the 0000 1000H through 0000 17FFH address range and is used to control the 80960Rx integrated peripherals. The processor ensures that accesses to MMRs do not generate external bus cycles.

3.3.1 i960® Core Processor Function Memory-Mapped Registers

Portions of the i960 Rx I/O processor address space (addresses FF00 0000H through FFFF FFFFH) are reserved for memory-mapped registers. These memory-mapped registers are accessed through word-operand memory instructions (**atmod**, **atadd**, **sysctl**, **ld** and **st** instructions) only. Accesses to this address space do not generate external bus cycles. The latency in accessing each of these registers is one cycle.

Each register has an associated access mode (user and supervisor modes) and access type (read and write accesses). Table C-2 and Table C-3 show all the memory-mapped registers and the application modes of access.

The registers are partitioned into user and supervisor spaces based on their addresses. Addresses FF00 0000H through FF00 7FFFH are allocated to user space memory-mapped registers; Addresses FF00 8000H to FFFF FFFFH are allocated to supervisor space registers.

3.3.1.1 Restrictions on Instructions that Access the i960® Core Processor Memory-Mapped Registers

The majority of memory-mapped registers can be accessed by both load (**ld**) and store (**st**) instructions. However some registers have restrictions on the types of accesses they allow. To ensure correct operation, the access type restrictions for each register should be followed. The access type columns of Table C-2 and Table C-3 indicate the allowed access types for each register.

Unless otherwise indicated by its access type, the modification of a memory-mapped register by a **st** instruction takes effect completely before the next instruction starts execution.

Some operations require an atomic-read-modify-write sequence to a register, most notably IPND and IMSK. The **atmod** and **atadd** instructions provide a special mechanism to quickly modify the IPND and IMSK registers in an atomic manner on the i960 Rx I/O processor. Do not use this instruction on any other memory-mapped registers.

The **sysctl** instruction can also modify the contents of a memory-mapped register atomically; in addition, **sysctl** is the only method to read the breakpoint registers on the i960 Rx I/O processor; the breakpoints cannot be read using a **ld** instruction.

At initialization the control table is automatically loaded into the on-chip control registers. This action simplifies the user's start-up code by providing a transparent setup of the processor's peripherals. See CHAPTER 11, INITIALIZATION AND SYSTEM REQUIREMENTS.

3.3.1.2 Access Faults for i960® Core Processor MMRs

Memory-mapped registers are meant to be accessed only as aligned, word-size registers with adherence to the appropriate access mode. Accessing these registers in any other way results in faults or undefined operation. An access is performed using the following fault model:

1. The access must be a word-sized, word-aligned access; otherwise, the processor generates an OPERATION.UNIMPLEMENTED fault.

2. If the access is a store in user mode to an implemented supervisor location, a TYPE.MISMATCH fault occurs. It is unpredictable whether a store to an unimplemented supervisor location causes a fault.

3. If the access is neither of the above, the access is attempted. Note that an MMR may generate faults based on conditions specific to that MMR. (Example: trying to write the timer registers in user mode when they have been allocated to supervisor mode only.)

4. When a store access to an MMR faults, the processor ensures that the store does not take effect.

5. A load access of a reserved location returns an unpredictable value.

6. Avoid any store accesses to reserved locations. Such a store can result in undefined operation of the processor if the location is in supervisor space.

Instruction fetches from the memory-mapped register space are not allowed and result in an OPERATION.UNIMPLEMENTED fault.

3.3.2 i960® Rx I/O Processor Peripheral Memory-Mapped Registers

The Peripheral Memory-Mapped Register (PMMR) interface gives software the ability to read and modify internal control registers. Each of these 32-bit registers is accessed as a memory-mapped register with a unique memory address, using regular memory-format instructions from the i960 core processor. See APPENDIX C, MEMORY-MAPPED REGISTERS.

The memory-mapped registers discussed in this chapter are specific to the i960 Rx I/O processor only. They support the DMA controller, memory controller, PCI and peripheral interrupt controller, messaging unit, local bus arbitration unit, PCI to PCI bridge unit, and PCI address translation unit, I^2C bus interface unit, and the APIC bus interface unit. This manual provides chapters that fully describe each of these peripherals.

The PMMR interface (addresses 0000 1000H through 0000 17FFH) provides full accessibility from the primary ATU, secondary ATU, and the i960 core processor.

3.3.2.1 Accessing The Peripheral Memory-Mapped Registers

The PMMR interface is a slave device connected to the 80960 local bus. This interface accepts data transactions that appear on the 80960 local bus from the Primary ATU, Secondary ATU, and the i960 core processor. The PMMR interface allows these devices to perform read, write, or read-modify-write transactions.

The PMMR interface does not support multi-word burst accesses from any bus master. The PMMR interface supports 32-bit bus width transactions only. Because of this, PMCON0:1 must be configured as a 32-bit memory region for accesses that originate from the i960 core processor.

The PMMR interface is byte addressable. For PMMR reads, all accesses are promoted to word accesses and all data bytes are returned. The byte enables generated by the bus masters when performing PMMR write cycles indicate which data bytes are valid on the internal 80960 local bus. However, there may be requirements from the individual units that interface to the PMMR. For example, when configuring the DMA channel’s control register, a full 32-bit write must be performed to configure and restart the DMA channel. These restrictions are highlighted in the chapters describing the integrated peripheral units.

The PMMR interface supports the 80960 local bus atomic operations from the i960 core processor. The i960 core processor provides **atmod** (atomic modify) and **atadd** (atomic add) instructions for atomic accesses to memory. When the 80960 processor executes an **atmod** or **atadd** instruction, the LOCK# signal is asserted. The 80960 local bus is not granted to any other bus master until the LOCK# signal is deasserted. This prevents other bus masters from accessing the PMMR interface during a locked operation.

All PMMR transactions are allowed from i960 core processor operating in either user mode or supervisor mode. In addition, the PMMR does not provide any access fault to the i960 core processor.

The following PMMR registers have read/write access from the 80960 local bus (for both the PCI Bridge and ATU):

- Vendor ID register
- Device ID register

- Revision ID register
- Class Code register
- Header Type register
- Bridge Subsystem ID register
- Bridge Subsystem Vendor ID register

For accesses through PCI configuration cycles, access is specified in the register definition located in the appropriate chapter.

For PCI configuration read transactions, the PMMR returns a zero value for reserved registers. For PCI configuration write transactions, the PMMR discards the data. For all other accesses, reading or writing a reserved register is undefined. See Table C-2 and Table C-3 for register memory locations.

3.4 ARCHITECTURALLY DEFINED DATA STRUCTURES

The architecture defines a set of data structures including stacks, interfaces to system procedures, interrupt handling procedures and fault handling procedures. Table 3-3 defines the data structures and references other sections of this manual where detailed information can be found.

The i960 Rx I/O processor defines two initialization data structures: the Initialization Boot Record (IBR) and the Process Control Block (PRCB). These structures provide initialization data and pointers to other data structures in memory. When the processor is initialized, these pointers are read from the initialization data structures and cached for internal use.

Pointers to the system procedure table, interrupt table, interrupt stack, fault table and control table are specified in the processor control block. Supervisor stack location is specified in the system procedure table. User stack location is specified in the user’s startup code. Of these structures, only the system procedure table, fault table, control table and initialization data structures may be in ROM; the interrupt table and stacks must be in RAM. The interrupt table must be located in RAM to allow posting of software interrupts.

Table 3-3. Data Structure Descriptions

Structure (see section)	Description
User and Supervisor Stacks 7.6, USER AND SUPERVISOR STACKS (pg. 7-18)	The processor uses these stacks when executing application code.
Interrupt Stack 8.1.5, Interrupt Stack And Interrupt Record (pg. 8-6)	A separate interrupt stack is provided to ensure that interrupt handling does not interfere with application programs.
System Procedure Table 3.7, USER-SUPERVISOR PROTECTION MODEL (pg. 3-21) 7.5, SYSTEM CALLS (pg. 7-15)	Contains pointers to system procedures. Application code uses the system call instruction (**calls**) to access system procedures through this table. A system supervisor call switches execution mode from user mode to supervisor mode. When the processor switches modes, it also switches to the supervisor stack.
Interrupt Table 8.1.4, Interrupt Table (pg. 8-4)	The interrupt table contains vectors (pointers) to interrupt handling procedures. When an interrupt is serviced, a particular interrupt table entry is specified.
Fault Table 9.3, FAULT TABLE (pg. 9-4)	Contains pointers to fault handling procedures. When the processor detects a fault, it selects a particular entry in the fault table. The architecture does not require a separate fault handling stack. Instead, a fault handling procedure uses the supervisor stack, user stack or interrupt stack, depending on the processor execution mode in which the fault occurred and the type of call made to the fault handling procedure.
Control Table 11.4.4, Control Table (pg. 11-21)	Contains on-chip control register values. Control table values are moved to on-chip registers at initialization or with **sysctl**.

3.5 MEMORY ADDRESS SPACE

The i960 Rx I/O processor’s local address space is byte-addressable with addresses running contiguously from 0 to 2^{32}-1. Some memory space is reserved or assigned special functions as shown in Figure 3-2.

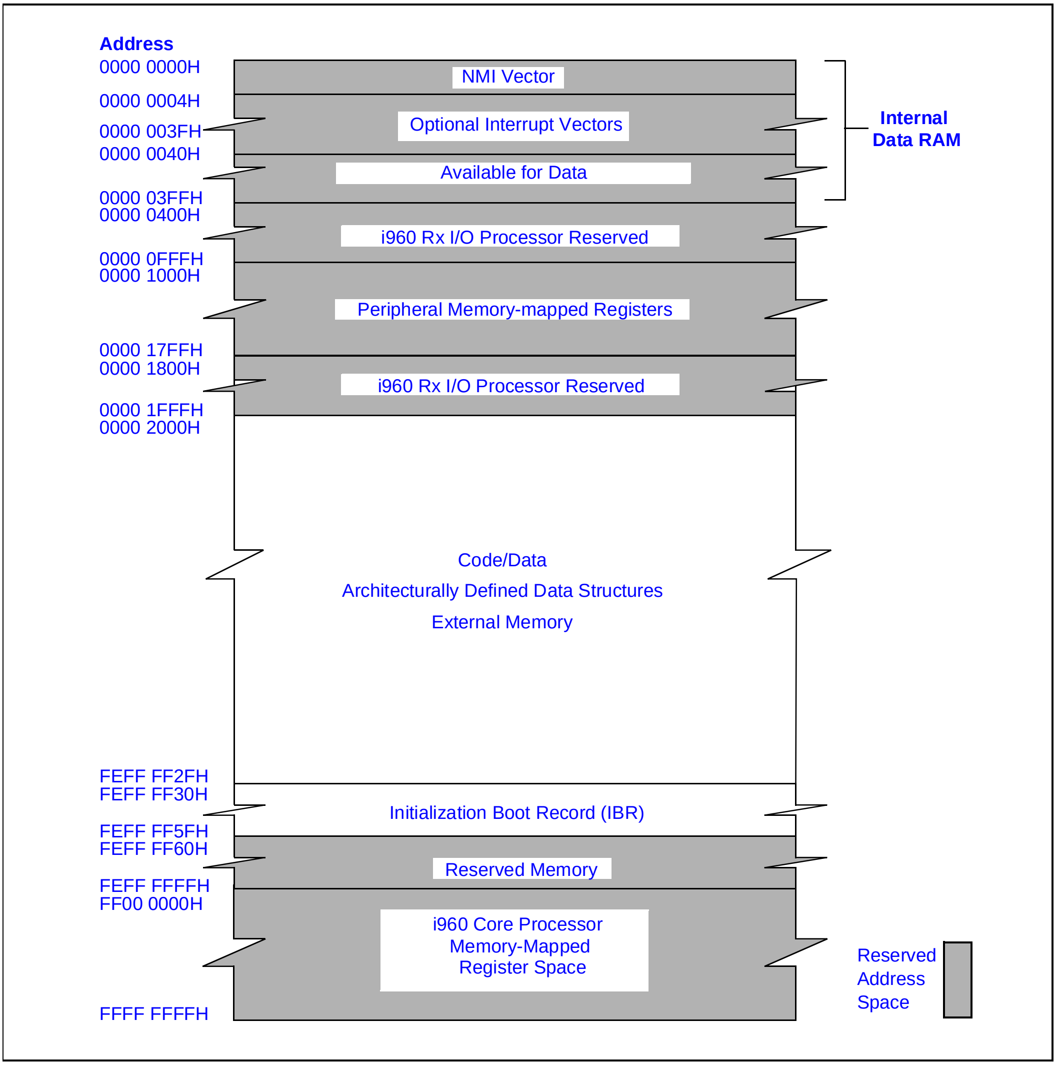


Figure 3-2. Local Memory Address Space

Physical addresses can be mapped to read-write memory, read-only memory and memory-mapped I/O. The architecture does not define a dedicated, addressable I/O space. There are no subdivisions of the address space such as segments. For memory management, an external memory management unit (MMU) may subdivide memory into pages or restrict access to certain areas of memory to protect a kernel's code, data and stack. However, the processor views this address space as linear.

An address in memory is a 32-bit value in the range 0H to FFFF FFFFH. Depending on the instruction, an address can reference in memory a single byte, short word (2 bytes), word (4 bytes), double word (8 bytes), triple word (12 bytes) or quad word (16 bytes). Refer to load and store instruction descriptions in CHAPTER 6, INSTRUCTION SET REFERENCE for multiple-byte addressing information.

3.5.1 Memory Requirements

The architecture requires that external memory have the following properties:

- Memory must be byte-addressable.
- Physical memory must not be mapped to reserved addresses that are specifically used by the processor implementation.
- Memory must guarantee indivisible access (read or write) for addresses that fall within 16-byte boundaries.
- Memory must guarantee atomic access for addresses that fall within 16-byte boundaries.

The latter two capabilities, *indivisible* and *atomic* access, are required only when multiple processors or other external agents, such as DMA or graphics controllers, share a common memory.

indivisible access
Guarantees that a processor, reading or writing a set of memory locations, complete the operation before another processor or external agent can read or write the same location. The processor requires indivisible access within an aligned 16-byte block of memory.

atomic access
A read-modify-write operation. Here the external memory system must guarantee that once a processor begins a read-modify-write operation on an aligned, 16-byte block of memory it is allowed to complete the operation before another processor or external agent can access to the same location. An atomic memory system can be implemented by using the LOCK# signal to qualify hold requests from external bus agents. The processor asserts LOCK# for the duration of an atomic memory operation.

The upper 16 Mbytes of the address space (addresses FF00 0000H through FFFF FFFFH and 0000 1000H through 0000 017FFH) are reserved for implementation-specific functions. i960 Rx I/O processor programs cannot use this address space except for accesses to memory-mapped registers. The processor does not generate any external bus cycles to this memory. As shown in Figure 3-2, part of the initialization boot record is located just below the i960 Rx I/O processor’s reserved memory.

The i960 Rx I/O processor requires some special consideration when using the lower 1 Kbyte of address space (addresses 0000H 03FFH). Loads and stores directed to these addresses access internal memory; instruction fetches from these addresses are not allowed by the processor. See section 4.1, INTERNAL DATA RAM (pg. 4-1). No external bus cycles are generated to this address space.

3.5.2 Data and Instruction Alignment in the Address Space

Instructions, program data and architecturally defined data structures can be placed anywhere in non-reserved address space while adhering to these alignment requirements:

- Align instructions on word boundaries.
- Align all architecturally defined data structures on the boundaries specified in Table 3-4.
- Align instruction operands for the atomic instructions (**atadd**, **atmod**) to word boundaries in memory.

The i960 Rx I/O processor can perform unaligned load or store accesses. The processor handles a non-aligned load or store request by:

- Automatically servicing a non-aligned memory access with microcode assistance as described in section 12.4.2, Bus Transactions Across Region Boundaries (pg. 12-7).
- After the access completes, the processor can generate an OPERATION.UNALIGNED fault, if directed to do so.

The method of handling faults is selected at initialization based on the value of the Fault Configuration Word in the Process Control Block. See section 11.4.2, Process Control Block – PRCB (pg. 11-17).

Table 3-4. Alignment of Data Structures in the Address Space

Data Structure	Alignment Boundary
System Procedure Table	4 byte
Interrupt Table	4 byte
Fault Table	4 byte
Control Table	16 byte
User Stack	16 byte
Supervisor Stack	16 byte
Interrupt Stack	16 byte
Process Control Block	16 byte
Initialization Boot Record	Fixed at FEFF FF30H

3.5.3 Byte, Word and Bit Addressing

The processor provides instructions for moving data blocks of various lengths from memory to registers (**ld**) and from registers to memory (**st**). Supported sizes for blocks are bytes, short words (2 bytes), words (4 bytes), double words, triple words and quad words. For example, **stl** (store long) stores an 8-byte (double word) data block in memory.

The most efficient way to move data blocks longer than 16 bytes is to move them in quad-word increments, using quad-word instructions **ldq** and **stq**.

When a data block is stored in memory, the block's least significant byte is stored at a base memory address and the more significant bytes are stored at successively higher byte addresses. This method of ordering bytes in memory is referred to as “little endian” ordering.

When loading a byte, short word or word from memory to a register, the block's least significant bit is always loaded in register bit 0. When loading double words, triple words and quad words, the least significant word is stored in the base register. The more significant words are then stored at successively higher-numbered registers. Individual bits can be addressed only in data that resides in a register: bit 0 in a register is the least significant bit, bit 31 is the most significant bit.

3.5.4 Internal Data RAM

The i960 Rx I/O processor has 1 Kbyte of on-chip data RAM. Only data accesses are allowed in this region. Portions of the data RAM can also be reserved for functions such as caching interrupt vectors. The internal RAM is fully described in CHAPTER 4, CACHE AND ON-CHIP DATA RAM.

3.5.5 Instruction Cache

The instruction cache enhances performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of cached code and loops of code in the cache and also provides more bus bandwidth for data operations in external memory. The i960 Rx I/O processor instruction cache is a 4 Kbyte, two-way set associative cache, organized in two sets of four-word lines.

3.5.6 Data Cache

The data cache on the i960 Rx I/O processor is a write-through 2-Kbyte direct-mapped cache. For more information, see CHAPTER 4, CACHE AND ON-CHIP DATA RAM.

3.6 PROCESSOR-STATE REGISTERS

The architecture defines four 32-bit registers that contain status and control information:

- Instruction Pointer (IP) register
- Process Controls (PC) register
- Arithmetic Controls (AC) register
- Trace Controls (TC) register

3.6.1 Instruction Pointer (IP) Register

The IP register contains the address of the instruction currently being executed. This address is 32 bits long; however, since instructions are required to be aligned on word boundaries in memory, the IP's two least-significant bits are always 0 (zero).

All i960 processor instructions are either one or two words long. The IP gives the address of the lowest-order byte of the first word of the instruction.

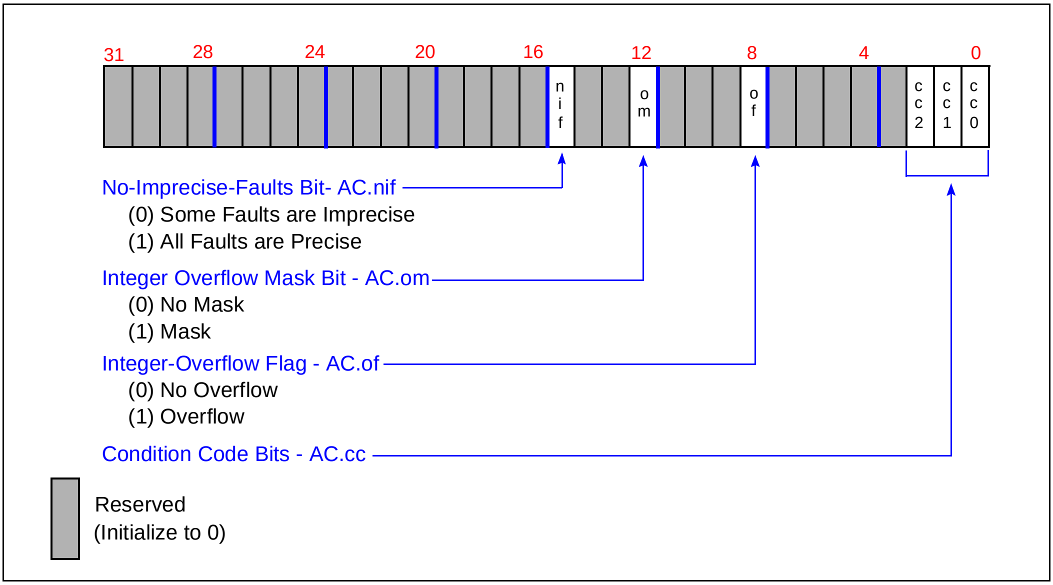
The IP register cannot be read directly. However, the IP-with-displacement addressing mode lets software use the IP as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current IP value.

When a break occurs in the instruction stream due to an interrupt, procedure call or fault, the processor stores the IP of the next instruction to be executed in local register r2, which is usually referred to as the return IP or RIP register. Refer to CHAPTER 7, PROCEDURE CALLS for further discussion.

3.6.2 Arithmetic Controls Register – AC

The AC register (Table 3-5) contains condition code flags, integer overflow flag, mask bit and a bit that controls faulting on imprecise faults. Unused AC register bits are reserved.

Table 3-5. Arithmetic Controls Register – AC


31
28
24
20
16
12
8
4
0
nif
om
of
cc2
cc1
cc0
No-Imprecise-Faults Bit- AC.nif
(0) Some Faults are Imprecise
(1) All Faults are Precise
Integer Overflow Mask Bit - AC.om
(0) No Mask
(1) Mask
Integer-Overflow Flag - AC.of
(0) No Overflow
(1) Overflow
Condition Code Bits - AC.cc
Reserved
(Initialize to 0)

3.6.2.1 Initializing and Modifying the AC Register

At initialization, the AC register is loaded from the Initial AC image field in the Process Control Block. Set reserved bits to 0 in the AC Register Initial Image. Refer to CHAPTER 11, INITIALIZATION AND SYSTEM REQUIREMENTS.

After initialization, software must not modify or depend on the AC register's initial image in the PRCB. Software can use the modify arithmetic controls (**modac**) instruction to examine and/or modify any of the register bits. This instruction provides a mask operand that lets user software limit access to the register's specific bits or groups of bits, such as the reserved bits.

The processor automatically saves and restores the AC register when it services an interrupt or handles a fault. The processor saves the current AC register state in an interrupt record or fault record, then restores the register upon returning from the interrupt or fault handler.

3.6.2.2 Condition Code (AC.cc)

The processor sets the AC register's *condition code flags* (bits 0-2) to indicate the results of certain instructions, such as compare instructions. Other instructions, such as conditional branch instructions, examine these flags and perform functions as dictated by the state of the condition code flags. Once the processor sets the condition code flags, the flags remain unchanged until another instruction executes that modifies the field.

Condition code flags show true/false conditions, inequalities (greater than, equal or less than conditions) or carry and overflow conditions for the extended arithmetic instructions. To show true or false conditions, the processor sets the flags as shown in Table 3-6. To show equality and inequalities, the processor sets the condition code flags as shown in Table 3-7.

Table 3-6. Condition Codes for True or False Conditions

Condition Code	Condition
010_2	true
000_2	false

Table 3-7. Condition Codes for Equality and Inequality Conditions

Condition Code	Condition
000_2	unordered
001_2	greater than
010_2	equal
100_2	less than

The term *unordered* is used when comparing floating point numbers. The i960 Rx I/O processor does not implement on-chip floating point processing.

To show carry out and overflow, the processor sets the condition code flags as shown in Table 3-8.

Table 3-8. Condition Codes for Carry Out and Overflow

Condition Code	Condition
$01X_2$	carry out
$0X1_2$	overflow

Certain instructions, such as the branch-if instructions, use a 3-bit mask to evaluate the condition code flags. For example, the branch-if-greater-or-equal instruction (**bge**) uses a mask of 011_2 to determine if the condition code is set to either greater-than or equal. Conditional instructions use similar masks for the remaining conditions such as: greater-or-equal (011_2), less-or-equal (110_2) and not-equal (101_2). The mask is part of the instruction opcode; the instruction performs a bitwise AND of the mask and condition code.

The AC register *integer overflow flag* (bit 8) and *integer overflow mask bit* (bit 12) are used in conjunction with the ARITHMETIC.INTEGER_OVERFLOW fault. The mask bit disables fault generation. When the fault is masked and integer overflow is encountered, the processor sets the integer overflow flag instead of generating a fault. If the fault is not masked, the fault is allowed to occur and the flag is not set.

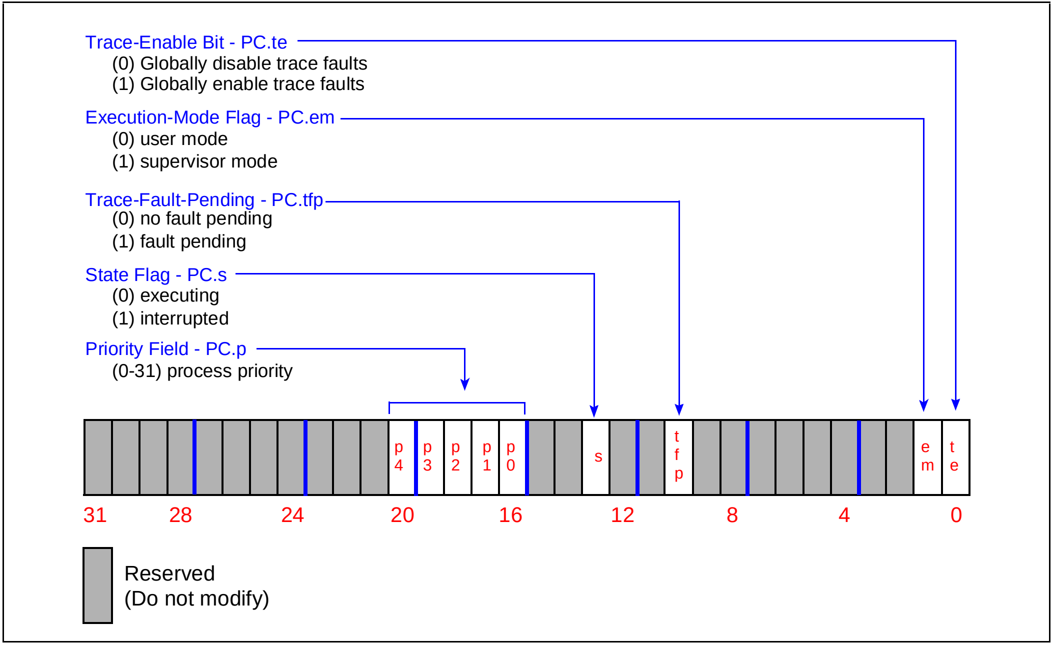
Once the processor sets this flag, the flag remains set until the application software clears it. Refer to the discussion of the ARITHMETIC.INTEGER_OVERFLOW fault in CHAPTER 9, FAULTS for more information about the integer overflow mask bit and flag.

The *no imprecise faults (AC.nif) bit* (bit 15) determines whether or not faults are allowed to be imprecise. If set, all faults are required to be precise; if clear, certain faults can be imprecise. See section 9.9, PRECISE AND IMPRECISE FAULTS (pg. 9-20) for more information.

3.6.3 Process Controls Register – PC

The PC register (Table 3-9) is used to control processor activity and show the processor's current state. The PC register *execution mode flag* (bit 1) indicates that the processor is operating in either user mode (0) or supervisor mode (1). The processor automatically sets this flag on a system call when a switch from user mode to supervisor mode occurs and it clears the flag on a return from supervisor mode. (User and supervisor modes are described in section 3.7, USER-SUPERVISOR PROTECTION MODEL (pg. 3-21).

Table 3-9. Process Controls Register – PC



PC register *state flag* (bit 13) indicates the processor state: executing (0) or interrupted (1). If the processor is servicing an interrupt, its state is interrupted. Otherwise, the processor's state is executing.

While in the interrupted state, the processor can receive and handle additional interrupts. When nested interrupts occur, the processor remains in the interrupted state until all interrupts are handled, then switches back to the executing state on the return from the initial interrupt procedure.

The PC register *priority field* (bits 16 through 20) indicates the processor's current executing or interrupted priority. The architecture defines a mechanism for prioritizing execution of code, servicing interrupts and servicing other implementation-dependent tasks or events. This mechanism defines 32 priority levels, ranging from 0 (the lowest priority level) to 31 (the highest). The priority field always reflects the current priority of the processor. Software can change this priority by use of the **modpc** instruction.

The processor uses the priority field to determine whether to service an interrupt immediately or to post the interrupt. The processor compares the priority of a requested interrupt with the current process priority. When the interrupt priority is greater than the current process priority or equal to 31, the interrupt is serviced; otherwise it is posted. When an interrupt is serviced, the process priority field is automatically changed to reflect interrupt priority. See CHAPTER 8, INTERRUPTS.

The PC register *trace enable bit* (bit 0) and *trace fault pending flag* (bit 10) control the tracing function. The trace enable bit determines whether trace faults are globally enabled (1) or globally disabled (0). The trace fault pending flag indicates that a trace event has been detected (1) or not detected (0). The tracing functions are further described in CHAPTER 10, TRACING AND DEBUGGING.

3.6.3.1 Initializing and Modifying the PC Register

Any of the following three methods can be used to change bits in the PC register:

- Modify process controls instruction (**modpc**)
- Alter the saved process controls prior to a return from an interrupt handler or fault handler

The **modpc** instruction reads and modifies the PC register directly. A TYPE.MISMATCH fault results if software executes **modpc** in user mode with a non-zero mask. As with **modac**, **modpc** provides a mask operand that can be used to limit access to specific bits or groups of bits in the register. In user mode, software can use **modpc** to read the current PC register.

In the latter two methods, the interrupt or fault handler changes process controls in the interrupt or fault record that is saved on the stack. Upon return from the interrupt or fault handler, the modified process controls are copied into the PC register. The processor must be in supervisor mode prior to return for modified process controls to be copied into the PC register.

When process controls are changed as described above, the processor recognizes the changes immediately except for one situation: if **modpc** is used to change the trace enable bit, the processor may not recognize the change before the next four non-branch instructions are executed.

After initialization (hardware reset), the process controls reflect the following conditions:

- priority = 31
- trace enable = disabled
- no trace fault pending
- execution mode = supervisor
- state = interrupted

When the processor is reinitialized with a **sysctl** reinitialize message, the PC register is not changed.

Software should not use **modpc** to modify execution mode or trace fault state flags except under special circumstances, such as in initialization code. Normally, execution mode is changed through the call and return mechanism. See section 6.2.43, modpc (pg. 6-78) for more details.

3.6.4 Trace Controls (TC) Register

The TC register, in conjunction with the PC register, controls processor tracing facilities. It contains trace mode enable bits and trace event flags that are used to enable specific tracing modes and record trace events, respectively. Trace controls are described in CHAPTER 10, TRACING AND DEBUGGING.

3.7 USER-SUPERVISOR PROTECTION MODEL

The processor can be in either of two execution modes: user or supervisor. The capability of a separate user and supervisor execution mode creates a code and data protection mechanism referred to as the user-supervisor protection model. This mechanism allows code, data and stack for a kernel (or system executive) to reside in the same address space as code, data and stack for the application. The mechanism restricts access to all or parts of the kernel by the application code. This protection mechanism prevents application software from inadvertently altering the kernel.

3.7.1 Supervisor Mode Resources

Supervisor mode is a privileged mode that provides several additional capabilities over user mode.

- When the processor switches to supervisor mode, it also switches to the supervisor stack. Switching to the supervisor stack helps maintain a kernel’s integrity. For example, it allows access to system debugging software or a system monitor, even if an application’s program destroys its own stack.

- In supervisor mode, the processor is allowed access to a set of supervisor-only functions and instructions. For example, the processor uses supervisor mode to handle interrupts and trace faults. Operations that can modify interrupt controller behavior or reconfigure bus controller characteristics can be performed only in supervisor mode. These functions include modification of control registers and internal data RAM that is dedicated to interrupt controllers. A fault is generated if supervisor-only operations are attempted while the processor is in user mode.

The PC register execution mode flag specifies processor execution mode. The processor automatically sets and clears this flag when it switches between the two execution modes.

- **dcctl** (data cache control)
- SFR as instruction operand
- **icctl** (instruction cache control)
- **intctl** (global interrupt enable and disable)
- **intdis** (global interrupt disable)
- **inten** (global interrupt enable)
- **modpc** (modify process controls w/ non-zero mask)
- **sysctl** (system control)
- Protected internal data RAM or Supervisor MMR space write
- Protected timer unit registers

Note that all of these instructions return a TYPE.MISMATCH fault if executed in user mode.

3.7.2 Using the User-Supervisor Protection Model

A program switches from user mode to supervisor mode by making a system-supervisor call (also referred to as a supervisor call). A system-supervisor call is a call executed with the call-system instruction (**calls**). With **calls**, the IP for the called procedure comes from the system procedure table. An entry in the system procedure table can specify an execution mode switch to supervisor mode when the called procedure is executed. **calls** and the system procedure table thus provide a tightly controlled interface to procedures that can execute in supervisor mode. Once the processor switches to supervisor mode, it remains in that mode until a return is performed to the procedure that caused the original mode switch.

Interrupts and faults can cause the processor to switch from user to supervisor mode. When the processor handles an interrupt, it automatically switches to supervisor mode. However, it does not switch to the supervisor stack. Instead, it switches to the interrupt stack. Fault table entries determine if a particular fault transitions the processor from user to supervisor mode.

If an application does not require a user-supervisor protection mechanism, the processor can always execute in supervisor mode. At initialization, the processor is placed in supervisor mode prior to executing the first instruction of the application code. The processor then remains in supervisor mode indefinitely, as long as no action is taken to change execution mode to user mode. The processor does not need a user stack in this case.

4

CACHE AND ON-CHIP DATA RAM

CHAPTER 4
CACHE AND ON-CHIP DATA RAM

This chapter describes the structure and user configuration of all forms of on-chip storage, including caches (data, local register and instruction) and data RAM.

4.1 INTERNAL DATA RAM

Internal data RAM is mapped to the lower 1 Kbyte (0 to 03FFH) of the address space. Loads and stores with target addresses in internal data RAM operate directly on the internal data RAM; no external bus activity is generated. Data RAM allows time-critical data storage and retrieval without dependence on external bus performance. Only data accesses are allowed to the internal data RAM; instructions cannot be fetched from the internal data RAM. Instruction fetches directed to the data RAM cause an OPERATION.UNIMPLEMENTED fault to occur.

Internal data RAM locations are never cached in the data cache. Logical Memory Template bits controlling caching are ignored for data RAM accesses.

Some internal data RAM locations are reserved for functions other than general data storage. The first 64 bytes of data RAM may be used to cache interrupt vectors, which reduces latency for these interrupts. The word at location 0000H is always reserved for the cached NMI vector. With the exception of the cached NMI vector, other reserved portions of the data RAM can be used for data storage when the alternate function is not used. All locations of the internal data RAM can be read in both supervisor and user mode.

The first 64 bytes (0000H to 003FH) of internal RAM are always user-mode write-protected. This portion of data RAM can be read while executing in user or supervisor mode; however, it can be only modified in supervisor mode. This area can also be write-protected from supervisor mode writes by setting the BCON.sirp bit. See section 12.3.1, Bus Control Register – BCON (pg. 12-6). Protecting this portion of the data RAM from user and supervisor rights preserves the interrupt vectors that may be cached there. See section 8.5.2.1, Vector Caching Option (pg. 8-46).

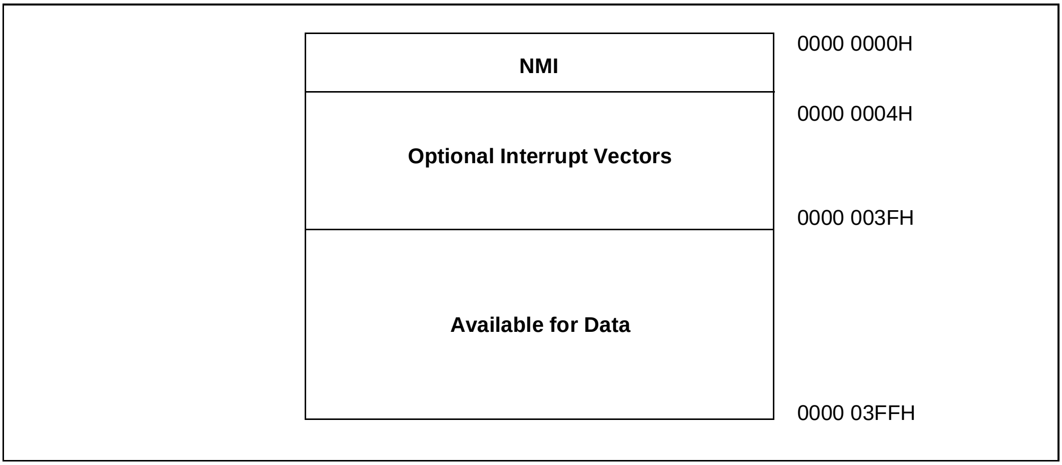


Figure 4-1. Internal Data RAM and Register Cache

The remainder of the internal data RAM can always be written from supervisor mode. User mode write protection is optionally selected for the rest of the data RAM (40H to 3FFH) by setting the Bus Control Register RAM protection bit (BCON.irp). Writes to internal data RAM locations while they are protected generate a TYPE.MISMATCH fault. See section 12.3.1, Bus Control Register – BCON (pg. 12-6) for the format of the BCON register.

New versions of i960 processor compilers take advantage of internal data RAM. Profiling compilers, such as those offered by Intel, can allocate the most frequently used variables into this RAM.

4.2 LOCAL REGISTER CACHE

The i960® Rx I/O processor provides fast storage of local registers for call and return operations by using an internal local register cache (also known as a *stack frame cache*). Up to eight local register sets can be contained in the cache before sets must be saved in external memory. The register set is all the local registers (i.e., r0 through r15). The processor uses a 128-bit wide bus to store local register sets quickly to the register cache. An integrated procedure call mechanism saves the current local register set when a call is executed. A local register set is saved into a frame in the local register cache, one frame per register set. When the eighth frame is saved, the oldest set of local registers is flushed to the procedure stack in external memory, which frees one frame.

Section 7.1.4, Caching Local Register Sets (pg. 7-7) and section 7.1.5, Mapping Local Registers to the Procedure Stack (pg. 7-11) further discuss the relationship between the internal register cache and the external procedure stack.

The branch-and-link (**bal** and **balx**) instructions do not cause the local registers to be stored.

The entire internal register cache contents can be copied to the external procedure stack through the flushreg instruction. Section 6.2.30, flushreg (pg. 6-54) explains the instruction itself and section 7.2, MODIFYING THE PFP REGISTER (pg. 7-11) offers a practical example when flushreg must be used.

To decrease interrupt latency, software can reserve a number of frames in the local register cache solely for high priority interrupts (interrupted state and process priority greater than or equal to 28). The remaining frames in the cache can be used by all code, including high-priority interrupts. When a frame is reserved for high-priority interrupts, the local registers of the code interrupted by a high-priority interrupt can be saved to the local register cache without causing a frame flush to memory, providing the local register cache is not already full. Thus, the register allocation for the implicit interrupt call does not incur the latency of a frame flush.

Software can reserve frames for high-priority interrupt code by writing bits 10 through 8 of the register cache configuration word in the PRCB. This value indicates the number of free frames within the register cache that can be used by high-priority interrupts only. Any attempt by non-critical code to reduce the number of free frames below this value will result in a frame flush to external memory. The free frame check is performed only when a frame is pushed, which occurs only for an implicit or explicit call. The following pseudo-code illustrates the operation of the register cache when a frame is pushed.

Example 4-1. Register Cache Operation

```
frames_for_non_critical = 7- RCW[11:8];
if (interrupt_request)
      set_interrupt_handler_PC;
push_frame;
number_of_frames = number_of_frames + 1;
if (number_of_frames = 8) {
            flush_register_frame(oldest_frame);
            number_of_frames = number_of_frames - 1; }
else if ( number_of_frames = (frames_for_non_critical + 1) &&
       (PC.priority < 28 || PC.state != interrupted) ) {
             flush_register_frame(oldest_frame);
             number_of_frames = number_of_frames - 1; }
```

The valid range for the number of reserved free frames is 0 to 7. Setting the value to 0 reserves no frames for exclusive use by high-priority interrupts. Setting the value to 1 reserves 1 frame for high-priority interrupts and 6 frames to be shared by all code. Setting the value to 7 causes the register cache to become disabled for non-critical code. If the number of reserved high-priority frames exceeds the allocated size of the register cache, the entire cache is reserved for high-priority interrupts. In that case, all low-priority interrupts and procedure calls cause frame spills to external memory.

4.3 INSTRUCTION CACHE

The i960 Rx I/O processor features a 4-Kbyte, 2-way set-associative instruction cache (I-cache) organized in lines of four 32-bit words. The cache provides fast execution of cached code and loops of code and provides more bus bandwidth for data operations in external memory. To optimize cache updates when branches or interrupts are executed, each word in the line has a separate valid bit. When requested instructions are found in the cache, the instruction fetch time is one cycle for up to four words. A mechanism to load and lock critical code within a way of the cache is provided along with a mechanism to disable the cache. The cache is managed through the **icctl** or **sysctl** instruction. The **sysctl** instruction supports the instruction cache to maintain compatibility with other i960 processor software. Using **icctl** is the preferred and more versatile method for controlling the instruction cache on the i960 Rx I/O processor.

Cache misses cause the processor to issue a double-word or a quad-word fetch, based on the location of the Instruction Pointer:

- If the IP is at word 0 or word 1 of a 16-byte block, a four-word fetch is initiated.
- If the IP is at word 2 or word 3 of a 16-byte block, a two-word fetch is initiated.

4.3.1 Enabling and Disabling the Instruction Cache

Enabling the instruction cache is controlled on reset or initialization by the instruction cache configuration word in the Process Control Block (PRCB); see Table 11-8. When bit 16 in the instruction cache configuration word is set, the instruction cache is disabled and all instruction fetches are directed to external memory. Disabling the instruction cache is useful for tracing execution in a software debug environment.

The instruction cache remains disabled until one of three operations is performed:

- **icctl** is issued with the enable instruction cache operation (preferred method)
- **sysctl** is issued with the configure-instruction-cache message type and cache configuration mode other than disable cache (provides compatibility with other i960 processors; not the preferred method for i960 Rx I/O processor).
- The processor is reinitialized with a new value in the instruction cache configuration word

4.3.2 Operation While the Instruction Cache Is Disabled

Disabling the instruction cache *does not* disable instruction buffering that may occur in the instruction fetch unit. A four-word instruction buffer is always enabled, even when the cache is disabled.

There is one tag and four word-valid bits associated with the buffer. Because there is only one tag for the buffer, any “miss” within the buffer causes the following:

- All four words of the buffer are invalidated.
- A new tag value for the required instruction is loaded.
- The required instruction(s) are fetched from external memory.

Depending on the alignment of the “missed” instruction, either two or four words of instructions are fetched and only the valid bits corresponding to the fetched words are set in the buffer. No external instruction fetches are generated until there is a “miss” within the buffer, even in the presence of forward and backward branches.

4.3.3 Loading and Locking Instructions in the Instruction Cache

The processor can be directed to load a block of instructions into the cache and then lock out all normal updates to the cache. This cache load-and-lock mechanism is provided to minimize latency on program control transfers to key operations such as interrupt service routines. The block size that can be loaded and locked on the i960 Rx I/O processor is one way of the cache.

An **icctl** or **sysctl** instruction is issued with a configure-instruction-cache message type to select the load-and-lock mechanism. When the lock option is selected, the processor loads the cache starting at an address specified as an operand to the instruction.

4.3.4 Instruction Cache Visibility

Instruction cache status can be determined by issuing **icctl** with an instruction-cache status message. To facilitate debugging, the instruction cache contents, instructions, tags and valid bits can be written to memory. This is done by issuing **icctl** with the store cache operation.

4.3.5 Instruction Cache Coherency

The i960 Rx I/O processor does not snoop the bus to prevent instruction cache incoherency. The cache does not detect modification to program memory by loads, stores or actions of other bus masters. Several situations may require program memory modification, such as uploading code at initialization or loading from a backplane bus or a disk drive.

The application program is responsible for synchronizing its own code modification and cache invalidation. In general, a program must ensure that modified code space is not accessed until modification and cache-invalidate are completed. To achieve cache coherency, instruction cache contents should be invalidated after code modification is complete. **icctl** invalidates the instruction cache for the i960 Rx I/O processor. Alternately, i960 processor legacy software can use **sysctl**.

4.4 DATA CACHE

The i960 Rx I/O processor features a 2-Kbyte, direct-mapped cache that enhances performance by reducing the number of data load and store accesses to external memory. The cache is write-through and write-allocate. It has a line size of 4 words and each line in the cache has a valid bit. To reduce fetch latency on cache misses, each word within a line also has a valid bit. Caches are managed through the **dcctl** instruction.

User settings in the memory region configuration registers LMCON0-1 and DLMCON determine the data accesses that are cacheable or non-cacheable based on memory region.

4.4.1 Enabling and Disabling the Data Cache

To cache data, two conditions must be met:

1. The data cache must be enabled. A **dcctl** instruction issued with an enable data cache message enables the cache. On reset or initialization, the data cache is always disabled and all valid bits are set to zero.
2. Data caching for a location must be enabled by the corresponding logical memory template, or by the default logical memory template if no other template applies. See section 12.2, PROGRAMMING THE PHYSICAL MEMORY ATTRIBUTES (PMCON REGISTERS) (pg. 12-3) for more details on logical memory templates.

When the data cache is disabled, all data fetches are directed to external memory. Disabling the data cache is useful for debugging or monitoring a system. To disable the data cache, issue a **dcctl** with a disable data cache message. The enable and disable status of the data cache and various attributes of the cache can be determined by a **dcctl** issued with a data-cache status message.

4.4.2 Multi-Word Data Accesses that Partially Hit the Data Cache

The following applies only when data caching is enabled for an access.

For a multi-word load access (**ldl**, **ldt**, **ldq**) in which none of the requested words hit the data cache, an external bus transaction is started to acquire all the words of the access.

For a multi-word load access that partially hits the data cache, the processor may either:

- Load or reload all words of the access (even those that hit) from the external bus.
- Load only missing words from the external bus and interleave them with words found in the data cache.

The multi-word alignment determines which of the above methods is used:

- Naturally aligned multi-word accesses cause all words to be reloaded.
- An unaligned multi-word access causes only missing words to be loaded.

When any words accessed with **ldl**, **ldt**, or **ldq** miss the data cache, every word accessed by that load instruction is updated in the cache.

Load Instruction	Number of Updated Words
ldq	4 words
ldt	3 words
ldl	2 words

In each case, the external bus accesses used to acquire the data may consist of none, one, or several burst accesses based on the alignment of the data and the bus-width of the memory region that contains the data. See Chapter 13, LOCAL BUS for more details.

A multi-word load access that completely hits in the data cache does not cause external bus accesses.

For a multi-word store access (**stl**, **stt**, **stq**) an external bus transaction is started to write all words of the access regardless if any or all words of the access hit the data cache. External bus accesses used to write the data may consist of either one or several burst accesses based on data alignment and the bus-width of the memory region that receives the data. The cache is also updated accordingly as described earlier in this chapter.

4.4.3 Data Cache Fill Policy

The i960 Rx I/O processor always uses a “natural” fill policy for cacheable loads. The processor fetches only the amount of data that is requested by a load (i.e., a word, long word, etc.) on a data cache miss. Exceptions are byte and short-word accesses, which are always promoted to words. This allows a complete word to be brought into the cache and marked valid. When the data cache is disabled and loads are done from a cacheable region, promotions from bytes and short words still take place.

4.4.4 Data Cache Write Policy

The write policy determines what happens on cacheable writes (stores). The i960 Rx I/O processor always uses a write-through policy. Stores are always seen on the external bus, thus maintaining coherency between the data cache and external memory.

The i960 Rx I/O processor always uses a write-allocate policy for data. For a cacheable location, data is always written to the data cache regardless of whether the access is a hit or miss. The following cases are relevant to consider:

1. In the case of a hit for a word or multi-word store, the appropriate line and word(s) are updated with the data.
2. In the case of a miss for a word or multi-word store, a tag and cache line are allocated, if needed, and the appropriate valid bits, line, and word(s) are updated.
3. In the case of byte or short-word data that hits a valid word in the cache, both the word in cache and external memory are updated with the data; the cache word remains valid.
4. In the case of byte or short-word data that falls within a valid line but misses because the appropriate word is invalid, both the word and external memory are updated with the data; however, the cache word remains invalid.
5. In the case of byte or short-word data that does not fall within a valid line, the external memory is updated with the data. For data writes less than a word, the data cache is not updated; the tags and valid bits are not changed.

A byte or short word is always invalid in the data cache since valid bits only apply to words.

For cacheable stores that are equal to or greater than a word in length, cache tags and appropriate valid bits are updated whenever data is written into the cache. Consider a word store that misses as an example. The tag is always updated and its valid bit is set. The appropriate valid bit for that word is always set and the other three valid bits are always cleared. If the word store hits the cache, the tag bits remain unchanged. The valid bit for the stored word is set; all other valid bits are unchanged.

Cacheable stores that are less than a word in length are handled differently. Byte and short-word stores that hit the cache (i.e., are contained in valid words within valid cache lines) do not change the tag and valid bits. The processor writes the data into the cache and external memory as usual. A byte or short-word store to an invalid word within a valid cache line leaves the word valid bit cleared because the rest of the word is still invalid. In these two cases the processor simultaneously writes the data into the cache and the external memory.

4.4.5 Data Cache Coherency and Non-Cacheable Accesses

The i960 Rx I/O processor ensures that the data cache is always kept coherent with accesses that it initiates and performs. The most visible application of this requirement concerns non-cacheable accesses discussed below. However, the processor does not provide data cache coherency for accesses on the external bus that it did not initiate. Software is responsible for maintaining coherency in a multi-processor environment.

An access is defined as non-cacheable when any of the following is true:

1. The access falls into an address range mapped by an enabled LMCON or DLMCON and the data-caching enabled bit in the matching LMCON is clear.
2. The entire data cache is disabled.
3. The access is a read operation of the read-modify-write sequence performed by an **atmod** or **atadd** instruction.
4. The access is an implicit read access to the interrupt table to post or deliver a software interrupt.

If the memory location targeted by an **atmod** or **atadd** instruction is currently in the data cache, it is invalidated.

If the address for a non-cacheable store matches a tag (“tag hit”), the corresponding cache line is marked invalid. This is because the word is not actually updated with the value of the store. This behavior ensures that the data cache never contains stale data in a single-processor system. A simple case illustrates the necessity of this behavior: a read of data previously stored by a non-cacheable access must return the new value of the data, not the value in the cache. Because the processor invalidates the appropriate word in the cache line on a store hit when the cache is disabled, coherency can be maintained when the data cache is enabled and disabled dynamically.

Data loads or stores invalidate the corresponding lines of the cache even when data caching is disabled. This behavior further ensures that the cache does not contain stale data.

4.4.6 External I/O and Bus Masters and Cache Coherency

The i960 Rx I/O processor implements a single processor coherency mechanism. There is no hardware mechanism, such as bus snooping, to support multiprocessing. If another bus master can change shared memory, there is no guarantee that the data cache contains the most recent data. The user must manage such data coherency issues in software.

A suggested practice is to program the LMCON0-1 registers such that I/O regions are non-cacheable. Partitioning the system in this fashion eliminates I/O as a source of coherency problems. See section 12.2, PROGRAMMING THE PHYSICAL MEMORY ATTRIBUTES (PMCON REGISTERS) (pg. 12-3) for more information on this subject.

4.4.7 Data Cache Visibility

Data cache status can be determined by a **dcctl** instruction issued with a data-cache status message. Data cache contents, data, tags and valid bits can be written to memory as an aid for debugging. This operation is accomplished by a **dcctl** instruction issued with the dump cache operand. See section 6.2.23, dcctl (pg. 6-39) for more information.

5

INSTRUCTION SET OVERVIEW

CHAPTER 5
INSTRUCTION SET OVERVIEW

This chapter provides an overview of the i960® microprocessor family's instruction set and i960® Rx I/O processor-specific instruction set extensions. Also discussed are the assembly-language and instruction-encoding formats, various instruction groups and each group's instructions.

CHAPTER 6, INSTRUCTION SET REFERENCE describes each instruction, including assembly language syntax, and the action taken when the instruction executes and examples of how to use the instruction.

5.1 INSTRUCTION FORMATS

i960 Rx I/O processor instructions may be described in two formats: assembly language and instruction encoding. The following subsections briefly describe these formats.

5.1.1 Assembly Language Format

Throughout this manual, instructions are referred to by their assembly language mnemonics. For example, the add ordinal instruction is referred to as **addo**. Examples use Intel 80960 assembly language syntax which consists of the instruction mnemonic followed by zero to three operands, separated by commas. In the following assembly language statement example for **addo**, ordinal operands in global registers g5 and g9 are added together, and the result is stored in g7:

```
    addo g5, g9, g7 # g7 = g9 + g5
```

In the assembly language listings in this chapter, registers are denoted as:

g	global register	r	local register
#	pound sign precedes a comment		

All numbers used as literals or in address expressions are assumed to be decimal. Hexadecimal numbers are denoted with a "0x" prefix (e.g., 0xffff0012). Several assembly language instruction statement examples follow. Additional assembly language examples are given in section 2.3.5, Addressing Mode Examples (pg. 2-7).

```
subi r3, r5, r6         #r6 = r5 - r3
setbit 13, g4, g5       #g5 = g4 with bit 13 set
lda 0xfab3, r12         #r12 = 0xfab3
ld (r4), g3             #g3 = memory location that r4 points to
st g10, (r6)[r7*2]      #g10 = memory location that r6+2*r7 points to
```

5.1.2 Instruction Encoding Formats

All instructions are encoded in one 32-bit machine language instruction — an *opword* — which must be word aligned in memory. An opword's most significant eight bits contain the opcode field. The opcode field determines the instruction to be performed and how the remainder of the machine language instruction is interpreted. Instructions are encoded in opwords in one of four formats (see Figure 5-1). For more information on instruction formats, see APPENDIX A, MACHINE-LEVEL INSTRUCTION FORMATS.

Table 5-1. Instruction Encoding Formats (REG, COBR, CRTL, MEM)

Instruction Type	Format	Description
register	REG	Most instructions are encoded in this format. Used primarily for instructions which perform register-to-register operations.
compare and branch	COBR	An encoding optimization which combines compare and branch operations into one opword. Other compare and branch operations are also provided as REG and CTRL format instructions.
control	CTRL	For branches and calls that do not depend on registers for address calculation.
memory	MEM	Used for referencing an operand which is a memory address. Load and store instructions — and some branch and call instructions — use this format. MEM format has two encodings: MEMA or MEMB. Usage depends upon the addressing mode selected. MEMB-formatted addressing modes use the word in memory immediately following the instruction opword as a 32-bit constant. MEMA format uses one word and MEMB uses two words.

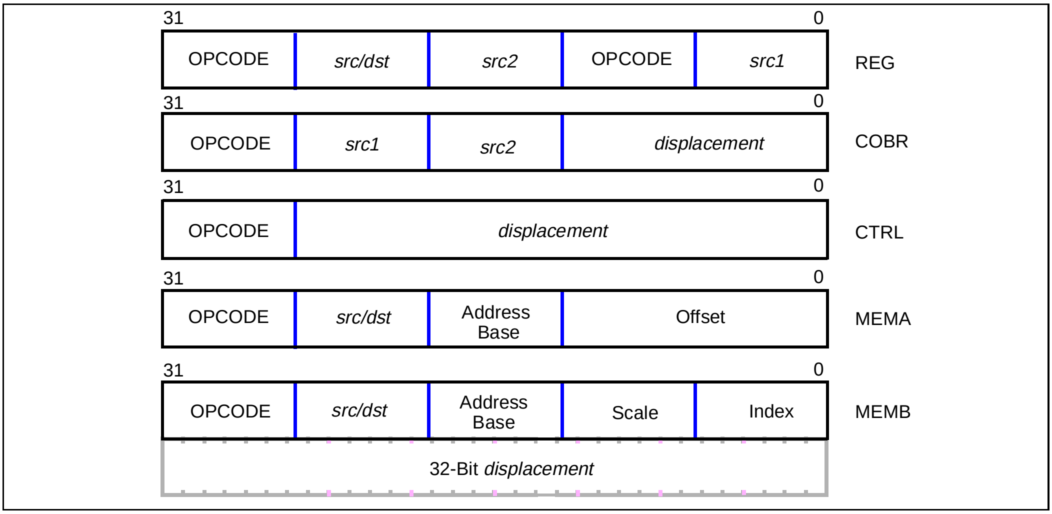


Figure 5-1. Machine-Level Instruction Formats

5.1.3 Instruction Operands

This section identifies and describes operands that can be used with the instruction formats.

Format	Operand(s)	Description
REG	*src1*, *src2*, *src/dst*	*src1* and *src2* can be global registers, local registers or literals. *src/dst* is either a global or a local register.
CTRL	*displacement*	CTRL format is used for branch and call instructions. *displacement* value indicates the target instruction of the branch or call.
COBR	*src1*, *src2*, *displacement*	*src1*, *src2* indicate values to be compared; *displacement* indicates branch target. *src1* can specify a global register, local register or a literal. *src2* can specify a global or local register.
MEM	*src/dst*, *efa*	Specifies source or destination register and an effective address (*efa*) formed by using the processor's addressing modes as described in section 2.3, MEMORY ADDRESSING MODES (pg. 2-5). Registers specified in a MEM format instruction must be either a global or local register.

5.2 INSTRUCTION GROUPS

The i960 processor instruction set can be categorized into the following functional groups shown in Table 5-2. The actual number of instructions is greater than those shown in this list because, for some operations, several unique instructions are provided to handle various operand sizes, data types or branch conditions. The following sections provide an overview of the instructions in each group. For detailed information about each instruction, refer to CHAPTER 6, INSTRUCTION SET REFERENCE.

Table 5-2. 80960Rx Instruction Set

Data Movement	Arithmetic	Logical	Bit, Bit Field and Byte
Load Store Move *Conditional Select Load Address	Add Subtract Multiply Divide Remainder Modulo Shift Extended Shift Extended Multiply Extended Divide Add with Carry Subtract with Carry *Conditional Add *Conditional Subtract Rotate	And Not And And Not Or Exclusive Or Not Or Or Not Nor Exclusive Nor Not Nand	Set Bit Clear Bit Not Bit Alter Bit Scan For Bit Span Over Bit Extract Modify Scan Byte for Equal *Byte Swap
Comparison	**Branch**	**Call/Return**	**Fault**
Compare Conditional Compare Compare and Increment Compare and Decrement Test Condition Code Check Bit	Unconditional Branch Conditional Branch Compare and Branch	Call Call Extended Call System Return Branch and Link	Conditional Fault Synchronize Faults
Debug	**Processor Management**	**Atomic**	
Modify Trace Controls Mark Force Mark	Flush Local Registers Modify Arithmetic Controls Modify Process Controls *Halt System Control *Cache Control *Interrupt Control	Atomic Add Atomic Modify	

* Denotes newer instructions that are NOT available on 80960CA/CF, 80960KA/KB and 80960SA/SB implementations.

5.2.1 Data Movement

These instructions are used to move data from memory to global and local registers, from global and local registers to memory, and between local and global registers.

Rules for register alignment must be followed when using load, store and move instructions that move 8, 12 or 16 bytes at a time. See section 3.5, MEMORY ADDRESS SPACE (pg. 3-10) for alignment requirements for code portability across implementations.

5.2.1.1 Load and Store Instructions

Load instructions copy bytes or words from memory to local or global registers or to a group of registers. Each load instruction has a corresponding store instruction to memory bytes or words to copy from a selected local or global register or group of registers. All load and store instructions use the MEM format.

ld	load word	**st**	store word
ldob	load ordinal byte	**stob**	store ordinal byte
ldos	load ordinal short	**stos**	store ordinal short
ldib	load integer byte	**stib**	store integer byte
ldis	load integer short	**stis**	store integer short
ldl	load long	**stl**	store long
ldt	load triple	**stt**	store triple
ldq	load quad	**stq**	store quad

ld copies 4 bytes from memory into a register; **ldl** copies 8 bytes; **ldt** copies 12 bytes into successive registers; **ldq** copies 16 bytes into successive registers.

st copies 4 bytes from a register into memory; **stl** copies 8 bytes; **stt** copies 12 bytes from successive registers; **stq** copies 16 bytes from successive registers.

For **ld**, **ldob**, **ldos**, **ldib** and **ldis**, the instruction specifies a memory address and register; the memory address value is copied into the register. The processor automatically extends byte and short (half-word) operands to 32 bits according to data type. Ordinals are zero-extended; integers are sign-extended.

For **st**, **stob**, **stos**, **stib** and **stis**, the instruction specifies a memory address and register; the register value is copied into memory. For byte and short instructions, the processor automatically reformats the source register’s 32-bit value for the shorter memory location. For **stib** and **stis**, this reformatting can cause integer overflow when the register value is too large for the shorter memory location. When integer overflow occurs, either an integer-overflow fault is generated or the integer-overflow flag in the AC register is set, depending on the integer-overflow mask bit setting in the AC register.

For **stob** and **stos**, the processor truncates the register value and does not create a fault when truncation resulted in the loss of significant bits.

5.2.1.2 Move

Move instructions copy data from a local or global register or group of registers to another register or group of registers. These instructions use the REG format.

mov	move word
movl	move long word
movt	move triple word
movq	move quad word

5.2.1.3 Load Address

The Load Address instruction (**lda**) computes an effective address in the address space from an operand presented in one of the addressing modes. **lda** is commonly used to load a constant into a register. This instruction uses the MEM format and can operate upon local or global registers.

On the i960 Rx I/O processor, **lda** is useful for performing simple arithmetic operations. The processor's parallelism allows **lda** to execute in the same clock as another arithmetic or logical operation.

5.2.2 Select Conditional

Given the proper condition code bit settings in the Arithmetic Controls register, these instructions move one of two pieces of data from its source to the specified destination.

selno	Select Based on Unordered
selg	Select Based on Greater
sele	Select Based on Equal
selge	Select Based on Greater or Equal
sell	Select Based on Less
selne	Select Based on Not Equal
selle	Select Based on Less or Equal
selo	Select Based on Ordered

5.2.3 Arithmetic

Table 5-3 lists arithmetic operations and data types for which the i960 Rx I/O processor provides instructions. "X" in this table indicates that the microprocessor provides an instruction for the specified operation and data type. All arithmetic operations are carried out on operands in registers or literals. Refer to section 5.2.11, Atomic Instructions (pg. 5-17) for instructions which handle specific requirements for in-place memory operations.

All arithmetic instructions use the REG format and can operate on local or global registers. The following subsections describe arithmetic instructions for ordinal and integer data types.

Table 5-3. Arithmetic Operations

Arithmetic Operations	Data Types	
	Integer	Ordinal
Add	X	X
Add with Carry	X	X
Conditional Add	X	X
Subtract	X	X
Subtract with Carry	X	X
Conditional Subtract	X	X
Multiply	X	X
Extended Multiply		X
Divide	X	X
Extended Divide		X
Remainder	X	X
Modulo	X	
Shift Left	X	X
Shift Right	X	X
Extended Shift Right		X
Shift Right Dividing Integer	X	

NOTE: “X” indicates that an instruction is available for the specified operation and data type.

5.2.3.1 Add, Subtract, Multiply, Divide, Conditional Add, Conditional Subtract

These instructions perform add, subtract, multiply or divide operations on integers and ordinals:

addi	Add Integer
addo	Add Ordinal
subi	Subtract Integer
subo	Subtract Ordinal
SUB<cc>	Conditional Subtract
muli	Multiply Integer
mulo	Multiply Ordinal
divi	Divide Integer
divo	Divide Ordinal

addi, **ADDI<cc>**, **subi**, **SUBI<cc>**, **muli** and **divi** generate an integer-overflow fault when the result is too large to fit in the 32-bit destination. **divi** and **divo** generate a zero-divide fault when the divisor is zero.

5.2.3.2 Remainder and Modulo

These instructions divide one operand by another and retain the remainder of the operation:

remi	remainder integer
remo	remainder ordinal
modi	modulo integer

The difference between the remainder and modulo instructions lies in the sign of the result. For **remi** and **remo**, the result has the same sign as the dividend; for **modi**, the result has the same sign as the divisor.

5.2.3.3 Shift, Rotate and Extended Shift

These shift instructions shift an operand a specified number of bits left or right:

shlo	shift left ordinal
shro	shift right ordinal
shli	shift left integer
shri	shift right integer
shrdi	shift right dividing integer
rotate	rotate left
eshro	extended shift right ordinal

Except for **rotate**, these instructions discard bits shifted beyond the register boundary.

shlo shifts zeros in from the least significant bit; **shro** shifts zeros in from the most significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

shli shifts zeros in from the least significant bit. When the shift operation results in an overflow, an integer-overflow fault is generated (when enabled). The destination register is written with the source shifted as much as possible without overflow and an integer-overflow fault is signaled.

shri performs a conventional arithmetic shift right operation by extending the sign bit. However, when this instruction is used to divide a negative integer operand by the power of 2, it may produce an incorrect quotient. (Discarding the bits shifted out has the effect of rounding the result toward negative.)

shrdi is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result when the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands. **shli** and **shrdi** are equivalent to **muli** and **divi** by the power of 2, respectively, except in cases where an overflow error occurs.

rotate rotates operand bits to the left (toward higher significance) by a specified number of bits. Bits shifted beyond the register's left boundary (bit 31) appear at the right boundary (bit 0).

The **eshro** instruction performs an ordinal right shift of a source register pair (64 bits) by as much as 32 bits and stores the result in a single (32-bit) register. This instruction is equivalent to an extended divide by a power of 2, which produces no remainder. The instruction is also the equivalent of a 64-bit extract of 32 bits.

5.2.3.4 Extended Arithmetic

These instructions support extended-precision arithmetic; i.e., arithmetic operations on operands greater than one word in length:

addc	add ordinal with carry
subc	subtract ordinal with carry
emul	extended multiply
ediv	extended divide

addc adds two word operands (literals or contained in registers) plus the AC Register condition code bit 1 (used here as a carry bit). When the result has a carry, bit 1 of the condition code is set; otherwise, it is cleared. This instruction's description in CHAPTER 6, INSTRUCTION SET REFERENCE gives an example of how this instruction can be used to add two long-word (64-bit) operands together.

subc is similar to **addc**, except it is used to subtract extended-precision values. Although **addc** and **subc** treat their operands as ordinals, the instructions also set bit 0 of the condition codes when the operation would have resulted in an integer overflow condition. This facilitates a software implementation of extended integer arithmetic.

emul multiplies two ordinals (each contained in a register), producing a long ordinal result (stored in two registers). **ediv** divides a long ordinal by an ordinal, producing an ordinal quotient and an ordinal remainder (stored in two adjacent registers).

5.2.4 Logical

These instructions perform bitwise Boolean operations on the specified operands:

and	*src2* AND *src1*
notand	(NOT *src2*) AND *src1*
andnot	*src2* AND (NOT *src1*)
xor	*src2* XOR *src1*
or	*src2* OR *src1*
nor	NOT (*src2* OR *src1*)
xnor	*src2* XNOR *src1*
not	NOT *src1*
notor	(NOT *src2*) or *src1*
ornot	*src2* **or** (NOT *src1*)
nand	NOT (*src2* AND *src1*)

All logical instructions use the REG format and can operate on literals or local or global registers.

5.2.5 Bit, Bit Field and Byte Operations

These perform operations on a specified bit or bit field in an ordinal operand. All Bit, Bit Field and Byte instructions use the REG format and can operate on literals or local or global registers.

5.2.5.1 Bit Operations

These instructions operate on a specified bit:

setbit	set bit
clrbit	clear bit
notbit	invert bit
alterbit	alter bit
scanbit	scan for bit
spanbit	span over bit

setbit, **clrbit** and **notbit** set, clear or complement (toggle) a specified bit in an ordinal.

alterbit alters the state of a specified bit in an ordinal according to the condition code. When the condition code is 010_2, the bit is set; when the condition code is 000_2, the bit is cleared.

chkbit, described in section 5.2.6, Comparison (pg. 5-11), can be used to check the value of an individual bit in an ordinal.

scanbit and **spanbit** find the most significant set bit or clear bit, respectively, in an ordinal.

5.2.5.2 Bit Field Operations

The two bit field instructions are **extract** and **modify**.

extract converts a specified bit field, taken from an ordinal value, into an ordinal value. In essence, this instruction shifts right a bit field in a register and fills in the bits to the left of the bit field with zeros. (**eshro** also provides the equivalent of a 64-bit extract of 32 bits).

modify copies bits from one register into another register. Only masked bits in the destination register are modified. **modify** is equivalent to a bit field move.

5.2.5.3 Byte Operations

scanbyte performs a byte-by-byte comparison of two ordinals to determine when any two corresponding bytes are equal. The condition code is set based on the results of the comparison. **scanbyte** uses the REG format and can specify literals or local or global registers as arguments.

bswap alters the order of bytes in a word, reversing its "endianess."

5.2.6 Comparison

The processor provides several types of instructions for comparing two operands, as described in the following subsections.

5.2.6.1 Compare and Conditional Compare

These instructions compare two operands then set the condition code bits in the AC register according to the results of the comparison:

cmpi	Compare Integer
cmpib	Compare Integer Byte
cmpis	Compare Integer Short
cmpo	Compare Ordinal
concmpi	Conditional Compare Integer
concmpo	Conditional Compare Ordinal
chkbit	Check Bit

These all use the REG format and can specify literals or local or global registers. The condition code bits are set to indicate whether one operand is less than, equal to, or greater than the other operand. See section 3.6.2, Arithmetic Controls Register – AC (pg. 3-16) for a description of the condition codes for conditional operations.

cmpi and **cmpo** simply compare the two operands and set the condition code bits accordingly. **concmpi** and **concmpo** first check the status of condition code bit 2:

- When not set, the operands are compared as with **cmpi** and **cmpo**.
- When set, no comparison is performed and the condition code flags are not changed.

The conditional-compare instructions are provided specifically to optimize two-sided range comparisons to check for the condition when A is between B and C (B ≤ A ≤ C). Here, a compare instruction (**cmpi** or **cmpo**) checks one side of the range (A ≥ B) and a conditional compare instruction (**concmpi** or **concmpo**) checks the other side (A ≤ C) according to the result of the first comparison. The condition codes following the conditional comparison directly reflect the results of both comparison operations. Therefore, only one conditional branch instruction is required to act upon the range check; otherwise, two branches would be needed.

chkbit checks a specified bit in a register and sets the condition code flags according to the bit state. The condition code is set to 010_2 when the bit is set and 000_2 otherwise.

5.2.6.2 Compare and Increment or Decrement

These instructions compare two operands, set the condition code bits according to the compare results, then increment or decrement one of the operands:

cmpinci	compare and increment integer
cmpinco	compare and increment ordinal
cmpdeci	compare and decrement integer
cmpdeco	compare and decrement ordinal

These all use the REG format and can specify literals or local or global registers. They are an architectural performance optimization which allows two register operations (e.g., compare and add) to execute in a single cycle. The intended use of these instructions is at the end of iterative loops.

5.2.6.3 Test Condition Codes

These test instructions allow the state of the condition code flags to be tested:

teste	test for equal
testne	test for not equal
testl	test for less
testle	test for less or equal
testg	test for greater
testge	test for greater or equal
testo	test for ordered
testno	test for unordered

When the condition code matches the instruction-specified condition, a TRUE (0000 0001H) is stored in a destination register; otherwise, a FALSE (0000 0000H) is stored. All use the COBR format and can operate on local and global registers.

5.2.7 Branch

Branch instructions allow program flow direction to be changed by explicitly modifying the IP. The processor provides three branch instruction types:

- unconditional branch
- conditional branch
- compare and branch

Most branch instructions specify the target IP by specifying a signed *displacement* to be added to the current IP. Other branch instructions specify the target IP's memory address, using one of the processor's addressing modes. This latter group of instructions is called extended addressing instructions (e.g., branch extended, branch-and-link extended).

5.2.7.1 Unconditional Branch

These instructions are used for unconditional branching:

b	Branch
bx	Branch Extended
bal	Branch and Link
balx	Branch and Link Extended

b and **bal** use the CTRL format. **bx** and **balx** use the MEM format and can specify local or global registers as operands. **b** and **bx** cause program execution to jump to the specified target IP. These two instructions perform the same function; however, their determination of the target IP differs. The target IP of a **b** instruction is specified at link time as a relative *displacement* from the current IP. The target IP of the **bx** instruction is the absolute address resulting from the instruction's use of a memory-addressing mode during execution.

bal and **balx** store the next instruction's address in a specified register, then jump to the specified target IP. (For **bal**, the RIP is automatically stored in register g14; for **balx**, the RIP location is specified with an instruction operand.) As described in section 7.9, BRANCH-AND-LINK (pg. 7-21), branch and link instructions provide a method of performing procedure calls that do not use the processor's integrated call/return mechanism. Here, the saved instruction address is used as a return IP. Branch and link is generally used to call leaf procedures (that is, procedures that do not call other procedures).

bx and **balx** can make use of any memory-addressing mode.

5.2.7.2 Conditional Branch

With conditional branch (**BRANCH IF**) instructions, the processor checks the AC register condition code flags. When these flags match the value specified with the instruction, the processor jumps to the target IP. These instructions use the *displacement-plus-ip* method of specifying the target IP:

be	branch if equal/true
bne	branch if not equal
bl	branch if less
ble	branch if less or equal
bg	branch if greater
bge	branch if greater or equal
bo	branch if ordered
bno	branch if unordered/false

All use the CTRL format. **bo** and **bno** are used with real numbers. **bno** can also be used with the result of a **chkbit** or **scanbit** instruction. Refer to section 3.6.2.2, Condition Code (AC.cc) (pg. 3-17) for a discussion of the condition code for conditional operations.

5.2.7.3 Compare and Branch

These instructions compare two operands then branch according to the comparison result. Three instruction subtypes are compare integer, compare ordinal and branch on bit:

cmpibe	compare integer and branch if equal
cmpibne	compare integer and branch if not equal
cmpibl	compare integer and branch if less
cmpible	compare integer and branch if less or equal
cmpibg	compare integer and branch if greater
cmpibge	compare integer and branch if greater or equal
cmpibo	compare integer and branch if ordered
cmpibno	compare integer and branch if unordered
cmpobe	compare ordinal and branch if equal
cmpobne	compare ordinal and branch if not equal
cmpobl	compare ordinal and branch if less
cmpoble	compare ordinal and branch if less or equal
cmpobg	compare ordinal and branch if greater
cmpobge	compare ordinal and branch if greater or equal
bbs	check bit and branch if set
bbc	check bit and branch if clear

All use the COBR machine instruction format and can specify literals, local or global registers as operands. With compare ordinal and branch (**compob***) and compare integer and branch (**compib***) instructions, two operands are compared and the condition code bits are set as described in section 5.2.6, Comparison (pg. 5-11). A conditional branch is then executed as with the conditional branch (**BRANCH IF**) instructions.

With check bit and branch instructions (**bbs**, **bbc**), one operand specifies a bit to be checked in the second operand. The condition code flags are set according to the state of the specified bit: 010_2 (true) when the bit is set and 000_2 (false) when the bit is clear. A conditional branch is then executed according to condition code bit settings.

These instructions can be used to optimize execution performance time. When it is not possible to separate adjacent compare and branch instructions from other unrelated instructions, replacing two instructions with a single compare and branch instruction increases performance.

5.2.8 Call/Return

The i960 Rx I/O processor offers an on-chip call/return mechanism for making procedure calls. Refer to section 7.1, CALL AND RETURN MECHANISM (pg. 7-2). The following instructions support this mechanism:

call	call
callx	call extended
calls	call system
ret	return

call and **ret** use the CTRL machine-instruction format. **callx** uses the MEM format and can specify local or global registers. **calls** uses the REG format and can specify local or global registers.

call and **callx** make local calls to procedures. A local call is a call that does not require a switch to another stack. **call** and **callx** differ only in the method of specifying the target procedure's address. The target procedure of a call is determined at link time and is encoded in the opword as a signed *displacement* relative to the call IP. **callx** specifies the target procedure as an absolute 32-bit address calculated at run time using any one of the addressing modes. For both instructions, a new set of local registers and a new stack frame are allocated for the called procedure.

calls is used to make calls to system procedures — procedures that provide a kernel or system-executive service. This instruction operates similarly to **call** and **callx**, except that it gets its target-procedure address from the system procedure table. An index number included as an operand in the instruction provides an entry point into the procedure table.

Depending on the type of entry being pointed to in the system procedure table, **calls** can cause either a system-supervisor call or a system-local call to be executed. A system-supervisor call is a call to a system procedure that switches the processor to supervisor mode and switches to the supervisor stack. A system-local call is a call to a system procedure that does not cause an execution mode or stack change. Supervisor mode is described throughout CHAPTER 7, PROCEDURE CALLS.

ret performs a return from a called procedure to the calling procedure (the procedure that made the call). **ret** obtains its target IP (return IP) from linkage information that was saved for the calling procedure. **ret** is used to return from all calls — including local and supervisor calls — and from implicit calls to interrupt and fault handlers.

5.2.9 Faults

Generally, the processor generates faults automatically as the result of certain operations. Fault handling procedures are then invoked to handle various fault types without explicit intervention by the currently running program. These conditional fault instructions permit a program to explicitly generate a fault according to the state of the condition code flags. All use the CTRL format.

faulte	fault if equal
faultne	fault if not equal
faultl	fault if less
faultle	fault if less or equal
faultg	fault if greater
faultge	fault if greater or equal
faulto	fault if ordered
faultno	fault if unordered

syncf ensures that any faults that occur during the execution of prior instructions occur before the instruction that follows the **syncf**. **syncf** uses the REG format and requires no operands.

5.2.10 Debug

The processor supports debugging and monitoring of program activity through the use of trace events. The following instructions support these debugging and monitoring tools:

modpc	modify process controls
modtc	modify trace controls
mark	mark
fmark	force mark

These all use the REG format. Trace functions are controlled with bits in the Trace Control (TC) register which enable or disable various types of tracing. Other TC register flags indicate when an enabled trace event is detected. Refer to CHAPTER 10, TRACING AND DEBUGGING.

modtc permits trace controls to be modified. **mark** causes a breakpoint trace event to be generated when breakpoint trace mode is enabled. **fmark** generates a breakpoint trace independent of the state of the breakpoint trace mode bits.

Other instructions that are helpful in debugging include **modpc** and **sysctl**. **modpc** can enable/disable trace fault generation. The **sysctl** instruction also provides control over breakpoint trace event generation. This instruction is used, in part, to load and control the i960 Rx I/O processor's breakpoint registers.

5.2.11 Atomic Instructions

Atomic instructions perform an atomic read-modify-write operation on operands in memory. An atomic operation is one in which other memory operations are forced to occur before or after, but not during, the accesses that comprise the atomic operation. These instructions are required to enable synchronization between interrupt handlers and background tasks in any system. They are also particularly useful in systems where several agents — processors, coprocessors or external logic — have access to the same system memory for communication.

The atomic instructions are atomic add (**atadd**) and atomic modify (**atmod**). **atadd** causes an operand to be added to the value in the specified memory location. **atmod** causes bits in the specified memory location to be modified under control of a mask. Both instructions use the REG format and can specify literals or local or global registers as operands.

5.2.12 Processor Management

These instructions control processor-related functions:

modpc	Modify the Process Controls register
flushreg	Flush cached local register sets to memory
modac	Modify the Arithmetic Controls register

All use the REG format and can specify literals or local or global registers.

modpc provides a method of reading and modifying PC register contents. Only programs operating in supervisor mode may modify the PC register; however, any program may read it.

The processor provides a flush local registers instruction (**flushreg**) to save the contents of the cached local registers to the stack. The flush local registers instruction automatically stores the contents of all the local register sets — except the current set — in the register save area of their associated stack frames.

The modify arithmetic controls instruction (**modac**) allows the AC register contents to be copied to a register and/or modified under the control of a mask. The AC register cannot be explicitly addressed with any other instruction; however, it is implicitly accessed by instructions that use the condition codes or set the integer overflow flag.

sysctl is used to configure the interrupt controller, breakpoint registers and instruction cache. It also permits software to signal an interrupt or cause a processor reset and reinitialization. **sysctl** may be executed only by programs operating in supervisor mode.

intctl, **inten** and **intdis** are used to enable and disable interrupts and to determine current interrupt enable status.

5.3 PERFORMANCE OPTIMIZATION

Performance optimization is categorized into two sections: instructions optimizations and miscellaneous optimizations.

5.3.1 Instruction Optimizations

Instruction optimizations are broken down by the instruction classification.

5.3.1.1 Load / Store Execution Model

Because the i960 Rx I/O processor has a 32-bit external data bus, multiple word accesses require multiple cycles. The processor uses microcode to sequence the multi-word accesses. Because the microcode can ensure that aligned multi-words are bursted together on the external bus, software should not substitute multiple single-word instructions for one multi-word instruction for data that is not likely to be in cache; i.e., one **ldq** provides better bus performance than four **ld** instructions.

Once a load is issued, the processor attempts to execute other instructions while the load is outstanding. It is important to note that when the load misses the data cache, the processor does not stall the issuing of subsequent instructions (other than stores) that do not depend on the load.

Software should avoid following a load with an instruction that depends on the result of the load. For a load that hits the data cache, a one-cycle stall occurs when the instruction immediately after the load requires the data. When the load fails to hit the data cache, the instruction depending on the load is stalled until the outstanding load request is resolved.

Multiple, back-to-back load instructions do not stall the processor until the bus queue becomes full.

The processor delays issuing a store instruction until all previously-issued load instructions complete. This happens regardless of whether the store is dependent on the load. This ordering between loads and stores ensures that the return data from a previous cache-read miss does not overwrite the cache line updated by a subsequent store.

5.3.1.2 Compare Operations

Byte and short word data is more efficiently compared using the new byte and short compare instructions (**cmpob, cmpib, cmpos, cmpis**), rather than shifting the data and using a word compare instruction.

5.3.1.3 Microcoded Instructions

While the majority of instructions on the i960 Rx I/O processor are single cycle and are executed directly by processor hardware, some require microcode emulation. Entry into a microcode routine requires two cycles. Exit from microcode typically requires two cycles. For some routines, one cycle of the exit process can execute in parallel with another instruction, thus saving one cycle of execution time.

5.3.1.4 Multiply-Divide Unit Instructions

The Multiply-Divide Unit (MDU) performs a number of multi-cycle arithmetic operations. These can range from 2 cycles for a 16-bitx32-bit **mulo**, 4 cycles for a 32-bitx32-bit **mulo**, to 30+ cycles for an **ediv**.

Once issued, these MDU instructions are executed in parallel with other non-MDU instructions that do not depend on the result of the MDU operation. Attempting to issue another MDU instruction while a current MDU instruction is executing, stalls the processor until the first one completes.

5.3.1.5 Multi-Cycle Register Operations

A few register operations can also take multiple cycles. The following instructions are performed in microcode:

- **bswap**
- **extract**
- **eshro**
- **modify**
- **movl**
- **movt**
- **movq**
- **shrdi**
- **scanbit**
- **spanbit**
- **testno**
- **testo**
- **testl**
- **testle**
- **teste**
- **testne**
- **testg**
- **testge**

On the i960 Rx I/O processor, **test<cc>** *dst* is microcoded and takes many more cycles than **SEL<cc>** 0,1,*dst***,** which is executed in one cycle directly by processor hardware.

Multi-register move operation execution time can be decreased at the expense of cache utilization and code density by using **mov** the appropriate number of times instead of **movl**, **movt** and **movq**.

5.3.1.6 Simple Control Transfer

There is no branch look-ahead or branch prediction mechanism on the i960 Rx I/O processor. Simple branch instructions take one cycle to execute, and one more cycle is needed to fetch the target instruction if the branch is actually taken.

b, bal, bno, bo, bl, ble, be, bne, bg, bge

One mode of the **bx** (branch-extended) instruction, **bx** (base), is also a simple branch and takes one cycle to execute and one cycle to fetch the target.

As a result, a **bal(g14)** or **bx (g14)** sequence provides a two-cycle call and return mechanism for efficient leaf procedure implementation.

Compare-and-branch instructions have been optimized on the i960 Rx I/O processor. They require two cycles to execute, and one more cycle to fetch the target instruction if the branch is actually taken. The instructions are:

- **cmpobno**
- **cmpobo**
- **cmpobl**
- **cmpoble**
- **cmpobe**
- **cmpobne**
- **cmpobg**
- **cmpobge**
- **cmpibno**
- **cmpibo**
- **cmpibl**
- **cmpible**
- **cmpibe**
- **cmpibg**
- **cmpibne**
- **cmpibge**
- **bbc**
- **bbs**

5.3.1.7 Memory Instructions

The i960 Rx I/O processor provides efficient support for naturally aligned byte, short, and word accesses that use one of six optimized addressing modes. These accesses require only one to two cycles to execute; additional cycles are needed for a load to return its data.

The byte, short and word memory instructions are:

ldob, ldib, ldos, ldis, ld, lda stob, stib, stos, stis, st

The remainder of accesses require multiple cycles to execute. These include:

- Unaligned short, and word accesses
- Byte, short, and word accesses that do not use one of the 6 optimized addressing modes
- Multi-word accesses

The multi-word accesses are:

ldl, ldt, ldq, stl, stt, stq

5.3.1.8 Unaligned Memory Accesses

Unaligned memory accesses are performed by microcode. Microcode sequences the access into smaller aligned pieces and does any merging of data that is needed. As a result, these accesses are not as efficient as aligned accesses. In addition, no bursting on the external bus is performed for these accesses. Whenever possible, unaligned accesses should be avoided.

5.3.2 Miscellaneous Optimizations

5.3.2.1 Masking of Integer Overflow

The i960 core architecture inserts an implicit **syncf** before performing a call operation or delivering an interrupt so that a fault handler can be dispatched first, when necessary. **syncf** can require a number of cycles to complete when a multi-cycle integer-multiply (**muli**) or integer-divide (**divi**) instruction is issued previously and integer-overflow faults are unmasked (allowed to occur). Call performance and interrupt latency can be improved by masking integer-overflow faults (AC.om = 1), which allows the implicit **syncf** to complete more quickly.

5.3.2.2 Avoid Using PFP, SP, R3 As Destinations for MDU Instructions

When performing a call operation or delivering an interrupt, the processor typically attempts to push the first four local registers (pfp, sp, rip, and r3) onto the local register cache as early as possible. Because of register-interlock, this operation is stalled until previous instructions return their results to these registers. In most cases, this is not a problem; however, in the case of multi-cycle instructions (**divo, divi, ediv, modi, remo, and remi**), the processor could be stalled for many cycles waiting for the result and unable to proceed to the next step of call processing or interrupt delivery.

Call performance and interrupt latency can be improved by avoiding the first four registers as the destination for a MDU instruction. Generally, registers pfp, sp, and rip should be avoided; they are used for procedure linking.

5.3.2.3 Use Global Registers (g0 - g14) As Destinations for MDU Instructions

Using the same rationale as in the previous item, call processing and interrupt performance are improved even further by using global registers (g0-g14) as the destination for multi-cycle MDU instructions. This is because there is no dependency between g0-g14 and implicit or explicit call operations (i.e., global registers are not pushed onto the local register cache).

5.3.2.4 Execute in Imprecise Fault Mode

Significant performance improvement is possible by allowing imprecise faults (AC.nif = 0). In precise fault mode (AC.nif = 1), the processor does not issue a new instruction until the previous one completes. This ensures that a fault from the previous instruction is delivered before the next instruction can begin execution. Imprecise fault mode allows new instructions to be issued before previous ones complete, thus increasing the instruction issue rate. Many applications can tolerate the imprecise fault reporting for the performance gain. A **syncf** can be used in imprecise fault mode to isolate faults at desired points of execution when necessary.

5.3.3 Cache Control

The following instructions provide instruction and data cache control functions.

icctl	Instruction cache control
dcctl	Data cache control

icctl and **dcctl** provide cache control functions including: enabling, disabling, loading and locking (instruction cache only), invalidating, getting status and storing cache information out to memory.

6

INSTRUCTION SET REFERENCE

CHAPTER 6
INSTRUCTION SET REFERENCE

This chapter provides detailed information about each instruction available to the i960® Rx I/O processor. Instructions are listed alphabetically by assembly language mnemonic. Format and notation used in this chapter are defined in section 6.1, NOTATION (pg. 6-1).

Information in this chapter is oriented toward programmers who write assembly language code for the i960 Rx I/O processor. Information provided for each instruction includes:

- Alphabetic listing of all instructions
- Assembly language mnemonic, name and format
- Description of the instruction’s operation
- Opcode and instruction encoding format
- Faults that can occur during execution
- Action (or algorithm) and other side effects of executing an instruction
- Assembly language example
- Related instructions

Additional information about the instruction set can be found in the following chapters and appendices in this manual:

- CHAPTER 5, INSTRUCTION SET OVERVIEW - Summarizes the instruction set by group and describes the assembly language instruction format.
- APPENDIX B, OPCODES AND EXECUTION TIMES - A quick-reference listing of instruction encodings assists debugging with a logic analyzer.
- APPENDIX A, MACHINE-LEVEL INSTRUCTION FORMATS - Describes instruction set opword encodings.

6.1 NOTATION

In general, notation in this chapter is consistent with usage throughout the manual; however, there are a few exceptions. Read the following subsections to understand notations that are specific to this chapter.

6.1.1 Alphabetic Reference

Instructions are listed alphabetically by assembly language mnemonic. When several instructions are related and fall together alphabetically, they are described as a group on a single page.

The instruction's assembly language mnemonic is shown in bold at the top of the page (e.g., **subc**). Occasionally, it is not practical to list all mnemonics at the page top. In these cases, the name of the instruction group is shown in capital letters (e.g., **BRANCH<cc>** or **FAULT<cc>**).

The i960 Rx I/O processor-specific extensions to the i960 microprocessor instruction set are indicated in the header text for each such instruction. This type of notation is also used to indicate new core architecture instructions. Sections describing new core instructions provide notes as to which i960-series processors do not implement these instructions.

Generally, instruction set extensions are not portable to other i960 processor implementations. Further, new core instructions are not typically portable to earlier i960 processor family implementations such as the i960 Kx microprocessors.

6.1.2 Mnemonic

The *Mnemonic* section gives the mnemonic (in boldface type) and instruction name for each instruction covered on the page, for example:

subi Subtract Integer

This name is the actual assembly language instruction name recognized by assemblers.

6.1.3 Format

The *Format* section gives the instruction's assembly language format and allowable operand types. Format is given in two or three lines. The following is a two-line format example:

sub*	*src1*	*src2*	*dst*
	reg/lit	reg/lit	reg

The first line gives the assembly language mnemonic (boldface type) and operands (italics). When the format is used for two or more instructions, an abbreviated form of the mnemonic is used. An * (asterisk) at the end of the mnemonic indicates a variable: in the above example, **sub*** is either **subi** or **subo**. Capital letters indicate an instruction class. For example, **ADD<cc>** refers to the class of conditional add instructions (e.g., **addio**, **addig**, **addoo**, **addog**).

Operand names are designed to describe operand function (e.g., *src*, *len*, *mask*).

The second line shows allowable entries for each operand. Notation is as follows:

reg	Global (g0 ... g15) or local (r0 ... r15) register
lit	Literal of the range 0 ... 31
disp	Signed displacement of range (-2^{22} ... $2^{22} - 1$)
mem	Address defined with the full range of addressing modes

In some cases, a third line is added to show register or memory location contents. For example, it may be useful to know that a register is to contain an address. The notation used in this line is as follows:

addr	Address
efa	Effective Address

6.1.4 Description

The *Description* section is a narrative description of the instruction's function and operands. It also gives programming hints when appropriate.

6.1.5 Action

The *Action* section gives an algorithm written in a "C-like" pseudo-code that describes direct effects and possible side effects of executing an instruction. Algorithms document the instruction's net effect on the programming environment; they do not necessarily describe how the processor actually implements the instruction. The following is an example of the action algorithm for the **alterbit** instruction:

```
if ((AC.cc & $010_2$)==0)
    dst = src2 & ~(2**(src1%32));
else
    dst = src2 | 2**(src1%32);
```

Table 6-1 defines each abbreviation used in the instruction reference pseudo-code. The pseudo-code has been written to comply as closely as possible with standard C programming language notation. Table 6-1 lists the pseudocode symbol definitions.

Table 6-1. Pseudo-Code Symbol Definitions (Sheet 1 of 2)

=	Assignment
==, !=	Comparison: equal, not equal
<, >	less than, greater than
<=, >=	less than or equal to, greater than or equal to
<<, >>	Logical Shift
**	Exponentiation
&, &&	Bitwise AND, logical AND
|, ||	Bitwise OR, logical OR
^	Bitwise XOR
~	One's Complement
%	Modulo
+, -	Addition, Subtraction

Table 6-1. Pseudo-Code Symbol Definitions (Sheet 2 of 2)

=	Assignment
*	Multiplication (Integer or Ordinal)
/	Division (Integer or Ordinal)
#	Comment delimiter

Table 6-2. Faults Applicable to All Instructions

Fault Type	Subtype	Description
OPERATION	UNIMPLEMENTED	An attempt to execute any instruction fetched from internal data RAM or a memory-mapped region causes an operation unimplemented fault.
TRACE	MARK	A Mark Trace Event is signaled after completion of an instruction for which there is a hardware breakpoint condition match. A Trace fault is generated when PC.mk is set.
	INSTRUCTION	An Instruction Trace Event is signaled after instruction completion. A Trace fault is generated when both PC.te and TC.i=1.

Table 6-3. Common Faulting Conditions

Fault Type	Subtype	Description
OPERATION	UNALIGNED	Any instruction that causes an unaligned memory access causes an operation aligned fault when unaligned faults are not masked in the fault configuration word in the Processor Control Block (PRCB).
	INVALID_OPCODE	This fault is generated when the processor attempts to execute an instruction containing an undefined opcode or addressing mode.
	INVALID_OPERAN D	This fault is caused by a non-defined operand in a supervisor mode only instruction or by an operand reference to an unaligned long-, triple- or quad-register group.
	UNIMPLEMENTED	This fault can occur due to an attempt to perform a non-word or unaligned access to a memory-mapped region or when attempting to fetch instructions from MMR space or internal data RAM.
Type	MISMATCH	Any instruction that attempts to write to supervisor protected internal data RAM or a memory-mapped register in supervisor space while not in supervisor mode causes a TYPE.MISMATCH fault. This fault is also generated for any non-supervisor mode reference to an SFR.

6.1.6 Faults

The *Faults* section lists faults that can be signaled as a direct result of instruction execution. Table 6-2 shows the possible faulting conditions that are common to the entire instruction set and could directly result from any instruction. These fault types are not included in the instruction reference. Table 6-3 shows the possible faulting conditions that are common to large subsets of the instruction set. When an instruction can generate a fault, it is noted in that instruction's *Faults* section. In these sections, "Standard" refers to the faults shown in Table 6-2 and Table 6-3.

6.1.7 Example

The *Example* section gives an assembly language example of an application of the instruction.

6.1.8 Opcode and Instruction Format

The *Opcode and Instruction Format* section gives the opcode and instruction format for each instruction, for example:

subi 593H REG

The opcode is given in hexadecimal format. The format is one of four possible formats: REG, COBR, CTRL and MEM. Refer to APPENDIX A, MACHINE-LEVEL INSTRUCTION FORMATS for more information on the formats.

6.1.9 See Also

The *See Also* section gives the mnemonics of related instructions which are also alphabetically listed in this chapter.

6.1.10 Side Effects

This section indicates whether the instruction causes changes to the condition code bits in the Arithmetic Controls.

6.1.11 Notes

This section provides additional information about an instruction such as whether it is implemented in other i960 processor families.

6.2 INSTRUCTIONS

The processor's instructions are arranged alphabetically by instruction or instruction group.

6.2.1 ADD<cc>

Mnemonic:		
	addono	Add Ordinal if Unordered
	addog	Add Ordinal if Greater
	addoe	Add Ordinal if Equal
	addoge	Add Ordinal if Greater or Equal
	addol	Add Ordinal if Less
	addone	Add Ordinal if Not Equal
	addole	Add Ordinal if Less or Equal
	addoo	Add Ordinal if Ordered
	addino	Add Integer if Unordered
	addig	Add Integer if Greater
	addie	Add Integer if Equal
	addige	Add Integer if Greater or Equal
	addil	Add Integer if Less
	addine	Add Integer if Not Equal
	addile	Add Integer if Less or Equal
	addio	Add Integer if Ordered

Format:	**add***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Conditionally adds *src2* and *src1* values and stores the result in *dst* based on the AC register condition code. If for Unordered the condition code is 0, or if for all other cases the logical AND of the condition code and the mask part of the opcode is not 0, then the values are added and placed in the destination. Otherwise the destination is left unchanged. Table 6-4 shows the condition code mask for each instruction. The mask is in opcode bits 4-6.

Table 6-4. Condition Code Mask Descriptions (Sheet 1 of 2)

Instruction	Mask	Condition
addono	000_2	Unordered
addino		
addog	001_2	Greater
addig		
addoe	010_2	Equal
addie		
addoge	011_2	Greater or equal
addige		
addol	100_2	Less
addil		
addone	101_2	Not equal
addine		

Table 6-4. Condition Code Mask Descriptions (Sheet 2 of 2)

Instruction	Mask	Condition
addole	110_2	Less or equal
addile		
addoo	111_2	Ordered
addio		

Action:

addo<cc>:

```
if((mask & AC.cc) || (mask == AC.cc))
    dst = (src1 + src2)[31:0];
```

addi<cc>:

```
if((mask & AC.cc) || (mask == AC.cc))
{
    {   true_result = (src1 + src2);
        dst = true_result[31:0];
    }
    if((true_result > (2**31) - 1) || (true_result < -2**31))
                                                # Check for overflow
    {   if(AC.om == 1)
            AC.of = 1;
        else
            generate_fault(ARITHMETIC.OVERFLOW);
    }
}
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
ARITHMETIC.OVERFLOW	Occurs only with **addi<cc>**.

Example:

```
# Assume (AC.cc AND 001_2) ≠ 0.
addig r4, r8, r10      # r10 = r8 + r4

# Assume (AC.cc AND 101_2) = 0.
addone r4, r8, r10     # r10 is not changed.
```

Opcode:	**addono**	780H	REG
	addog	790H	REG
	addoe	7A0H	REG
	addoge	7B0H	REG
	addol	7C0H	REG
	addone	7D0H	REG
	addole	7E0H	REG
	addoo	7F0H	REG
	addino	781H	REG
	addig	791H	REG
	addie	7A1H	REG
	addige	7B1H	REG
	addil	7C1H	REG
	addine	7D1H	REG
	addile	7E1H	REG
	addio	7F1H	REG

See Also: **addc, SUB<cc>, addi, addo**

Notes: This class of core instructions is not implemented on 80960Cx, Kx and Sx processors.

6.2.2 addc

Mnemonic: **addc** Add Ordinal With Carry

Format: **addc** *src1*, *src2*, *dst*
reg/lit reg/lit reg

Description: Adds *src2* and *src1* values and condition code bit 1 (used here as a carry-in) and stores the result in *dst*. If ordinal addition results in a carry out, condition code bit 1 is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, condition code bit 0 is set; otherwise, bit 0 is cleared. Regardless of addition results, condition code bit 2 is always set to 0.

addc can be used for ordinal or integer arithmetic. **addc** does not distinguish between ordinal and integer source operands. Instead, the processor evaluates the result for both data types and sets condition code bits 0 and 1 accordingly.

An integer overflow fault is never signaled with this instruction.

Action:

```
dst = (src1 + src2 + AC.cc[1])[31:0];
AC.cc[2:0] = $000_2$;
if((src2[31] == src1[31]) && (src2[31] != dst[31]))
    AC.cc[0] = 1;                                   # Set overflow bit.
AC.cc[1] = (src2 + src1 + AC.cc[1])[32];            # Carry out.
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Example of double-precision arithmetic.
# Assume 64-bit source operands
# in g0,g1 and g2,g3
cmpo 1, 0           # Clears Bit 1 (carry bit) of
                    # the AC.cc.
addc g0, g2, g0     # Add low-order 32 bits:
                    # g0 = g2 + g0 + carry bit
addc g1, g3, g1     # Add high-order 32 bits:
                    # g1 = g3 + g1 + carry bit
                    # 64-bit result is in g0, g1.
```

Opcode: **addc** 5B0H REG

See Also: **ADD<cc>, SUB<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.3 addi, addo

Mnemonic:		
	addo	Add Ordinal
	addi	Add Integer

Format:				
	add*	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Adds *src2* and *src1* values and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that **addi** can signal an integer overflow.

Action:

addo:

```
dst = (src2 +src1)[31:0];
```

addi:

```
true_result = (src1 + src2);
dst = true_result[31:0];
if((true_result > (2**31) - 1) || (true_result < -2**31)) # Check for overflow
{    if(AC.om == 1)
          AC.of = 1;
     else
          generate_fault(ARITHMETIC.OVERFLOW);
}
```

Faults:		
	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	ARITHMETIC.OVERFLOW	Occurs only with **addi**.

Example:

```
addi r4, g5, r9     # r9 = g5 + r4
```

Opcode:			
	addo	590H	REG
	addi	591H	REG

See Also: **addc, subi, subo, subc, ADD<cc>**

6.2.4 alterbit

Mnemonic: **alterbit** Alter Bit

Format:	**alterbit**	*bitpos*,	*src*,	*dst*
		reg/lit	reg/lit	reg

Description: Copies *src* value to *dst* with one bit altered. *bitpos* operand specifies bit to be changed; condition code determines the value to which the bit is set. If condition code is $X1X_2$, bit 1 = 1, the selected bit is set; otherwise, it is cleared. Typically this instruction is used to set the *bitpos* bit in the *targ* register if the result of a compare instruction is the equal condition code (010_2).

Action:

```
if((AC.cc & $010_2$)==0)
    dst = src & ~(2**(bitpos%32));
else
    dst = src | 2**(bitpos%32);
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Assume AC.cc = 010$_2$.
alterbit 24, g4,g9 # g9 = g4, with bit 24 set.
```

Opcode: **alterbit** 58FH REG

See Also: **chkbit, clrbit, notbit, setbit**

6.2.5 and, andnot

Mnemonic:

Mnemonic:	**and**	And
	andnot	And Not

Format:

and	*src1,*	*src2,*	*dst*
	reg/lit	reg/lit	reg
andnot	*src1,*	*src2,*	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise AND (**and**) or AND NOT (**andnot**) operation on *src2* and *src1* values and stores result in *dst*. Note in the action expressions below, *src2* operand comes first, so that with **andnot** the expression is evaluated as:

{*src2* and not (*src1*)}

rather than

{*src1* and not (*src2*)}.

Action:

and:

```
dst = src2 & src1;
```

andnot:

```
dst = src2 & ~src1;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
and 0x7, g8, g2      # Put lower 3 bits of g8 in g2.
andnot 0x7, r12, r9  # Copy r12 to r9 with lower
                     # three bits cleared.
```

Opcode:

and	581H	REG
andnot	582H	REG

See Also: **nand, nor, not, notand, notor, or, ornot, xnor, xor**

6.2.6 atadd

Mnemonic: **atadd** Atomic Add

Format:	**atadd**	*addr*,	*src*,	*dst*
		reg	reg/lit	reg

Description: Adds *src* value (full word) to value in the memory location specified with *addr* operand. This read-modify-write operation is performed on the actual data in memory and never on a cached value on chip. Initial value from memory is stored in *dst*.

Memory read and write are done atomically (i.e., other bus masters must be prevented from accessing the word of memory containing the word specified by *src/dst* operand until operation completes). See 3.5.1, Memory Requirements (pg. 3-12) or more information on atomic accesses.

Memory location in *addr* is the word's first byte (LSB) address. Address is automatically aligned to a word boundary. (Note that *addr* operand maps to *src1* operand of the REG format.)

Action:

```
implicit_syncf();
tempa = addr & 0xFFFFFFFC;
temp = atomic_read(tempa);
atomic_write(tempa, temp+src);
dst = temp;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
atadd r8, r3, r11   # r8 contains the address of
                    # memory location.
                    # r11 = (r8)
                    # (r8) = r11 + r3.
```

Opcode: **atadd** 612H REG

See Also: **atmod**

6.2.7 atmod

Mnemonic:	**atmod**	Atomic Modify		
Format:	**atmod**	*addr,* reg	*mask,* reg/lit	*src/dst* reg

Description: Copies the selected bits of *src/dst* value into memory location specified in *addr*. The read-modify-write operation is performed on the actual data in memory and never on a cached value on chip. Bits set in *mask* operand select bits to be modified in memory. Initial value from memory is stored in *src/dst*. See 3.5.1, Memory Requirements (pg. 3-12) for information on atomic accesses.

Memory read and write are done atomically (i.e., other bus masters must be prevented from accessing the word of memory containing the word specified with the *src/dst* operand until operation completes).

Memory location in *addr* is the modified word's first byte (LSB) address. Address is automatically aligned to a word boundary.

Action:

```
implicit_syncf();
tempa = addr & 0xFFFFFFFC;
tempb = atomic_read(tempa);
temp = (tempb &~ mask) | (src_dst & mask);
atomic_write(tempa, temp);
src_dst = tempb;
```

Faults: STANDARD   Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
atmod g5, g7, g10   # tempa = (g5)
                    # temp = (tempa andnot g7) or
                    # (g10 and g7)
                    # (g5) = temp
                    # g10 = tempa
```

Opcode: **atmod**   610H   REG

See Also: **atadd**

6.2.8 b, bx

Mnemonic:	**b**	Branch
	bx	Branch Extended
Format:	**b**	*targ* disp
	bx	*targ* mem

Description: Branches to the specified target.

With the **b** instruction, IP specified with *targ* operand can be no farther than -2^{23} to (2^{23}- 4) bytes from current IP. When using the Intel i960 processor assembler, *targ* operand must be a label which specifies target instruction's IP.

bx performs the same operation as **b** except the target instruction can be farther than -2^{23} to (2^{23}- 4) bytes from current IP. Here, the target operand is an effective address, which allows the full range of addressing modes to be used to specify target instruction's IP. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing target address in a register then using a register-indirect addressing mode.

Refer to 2.3, MEMORY ADDRESSING MODES (pg. 2-5) for information on this subject.

Action: **b, bx:**

```
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
b xyz                  # IP = xyz;
bx 1332 (ip)           # IP = IP + 8 + 1332;
# this example uses IP-relative addressing
```

Opcode:	**b**	08H	CTRL
	bx	84H	MEM

See Also: **bal, balx, BRANCH<cc>, COMPARE AND BRANCH<cc>, bbc, bbs**

6.2.9 bal, balx

Mnemonic:	**bal**	Branch and Link	
	balx	Branch and Link Extended	
Format:	**bal**	*targ* disp	
	balx	*targ,* mem	*dst* reg

Description: Stores address of instruction following **bal** or **balx** in a register then branches to the instruction specified with the *targ* operand.

The **bal** and **balx** instructions are used to call leaf procedures (procedures that do not call other procedures). The IP saved in the register provides a return IP that the leaf procedure can branch to (using a **b** or **bx** instruction) to perform a return from the procedure. Note that these instructions do not use the processor's call-and-return mechanism, so the calling procedure shares its local-register set with the called (leaf) procedure.

With **bal**, address of next instruction is stored in register g14. *targ* operand value can be no farther than -2^{23} to $(2^{23}- 4)$ bytes from current IP. When using the Intel i960 processor assembler, *targ* must be a label which specifies the target instruction's IP.

balx performs same operation as **bal** except next instruction address is stored in *dst* (allowing the return IP to be stored in any available register). With **balx**, the full address space can be accessed. Here, the target operand is an effective address, which allows full range of addressing modes to be used to specify target IP. "IP + displacement" addressing mode allows instruction to be IP-relative. Indirect branching can be performed by placing target address in a register and then using a register-indirect addressing mode.

See 2.3, MEMORY ADDRESSING MODES (pg. 2-5) for a complete discussion of addressing modes available with memory-type operands.

Action:

bal:

```
g14 = IP + 4;
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
```

balx:

```
dst = IP + instruction_length;
# Instruction_length = 4 or 8 depending on the addressing mode used.
IP[31:2] = effective_address(targ[31:2]);     # Resume execution at new IP.
IP[1:0] = 0;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
bal xyz              # g14 = IP + 4
                     # IP = xyz
balx (g2), g4        # g4 = IP + 4
                     # IP = (g2)
```

Opcode:

bal	0BH	CTRL
balx	85H	MEM

See Also: **b, bx, BRANCH<cc>, COMPARE AND BRANCH<cc>, bbc, bbs**

6.2.10 bbc, bbs

Mnemonic:	**bbc**	Check Bit and Branch If Clear	
	bbs	Check Bit and Branch If Set	
Format:	**bb***	*bitpos*, reg/lit	*src*, reg	*targ* disp

Description: Checks bit (designated by *bitpos*) in *src* and sets AC register condition code according to *src* value. The processor then performs conditional branch to instruction specified with *targ*, based on condition code state.

For **bbc**, if selected bit in *src* is clear, the processor sets condition code to 000_2 and branches to instruction specified by *targ*; otherwise, it sets condition code to 010_2 and goes to next instruction.

For **bbs**, if selected bit is set, the processor sets condition code to 010_2 and branches to *targ*; otherwise, it sets condition code to 000_2 and goes to next instruction.

targ can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from current IP. When using the Intel i960 processor assembler, *targ* must be a label which specifies target instruction's IP.

Action:

bbs:

```
if((src & 2**(bitpos%32)) == 1)
{     AC.cc = 010₂;
      temp[31:2] = sign_extension(targ[12:2]);
      IP[31:2] = IP[31:2] + temp[31:2];
      IP[1:0] = 0;
}
else
      AC.cc = 000₂;
```

bbc:

```
if((src & 2**(bitpos%32)) == 0)
{     AC.cc = 000₂;
      temp[31:2] = sign_extension(targ[12:2]);
      IP[31:2] = IP[31:2] + temp[31:2];
      IP[1:0] = 0;
}
else
      AC.cc = 010₂;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Assume bit 10 of r6 is clear.
bbc 10, r6, xyz     # Bit 10 of r6 is checked
                    # and found clear:
                    # AC.cc = 000
                    # IP = xyz;
```

Opcode:

bbc	30H	COBR
bbs	37H	COBR

See Also: **chkbit, COMPARE AND BRANCH<cc>, BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.11 BRANCH<cc>

Mnemonic:	**be**	Branch If Equal
	bne	Branch If Not Equal
	bl	Branch If Less
	ble	Branch If Less Or Equal
	bg	Branch If Greater
	bge	Branch If Greater Or Equal
	bo	Branch If Ordered
	bno	Branch If Unordered

Format: **b*** *targ*
disp

Description: Branches to instruction specified with *targ* operand according to AC register condition code state.

For all branch<cc> instructions except **bno**, the processor branches to instruction specified with *targ*, if the logical AND of condition code and mask part of opcode is not zero. Otherwise, it goes to next instruction.

For **bno**, the processor branches to instruction specified with *targ* if the condition code is zero. Otherwise, it goes to next instruction.

For instance, **bno** (unordered) can be used as a branch if false instruction when coupled with **chkbit**. For **bno**, branch is taken if condition code equals 000_2. **be** can be used as branch-if true instruction.

The *targ* operand value can be no farther than -2^{23} to $(2^{23}- 4)$ bytes from current IP.

The following table shows condition code mask for each instruction. The mask is in opcode bits 0-2.

Instruction	Mask	Condition
bno	000_2	Unordered
bg	001_2	Greater
be	010_2	Equal
bge	011_2	Greater or equal
bl	100_2	Less
bne	101_2	Not equal
ble	110_2	Less or equal
bo	111_2	Ordered

Action:

```
if((mask & AC.cc) || (mask == AC.cc))
{     temp[31:2] = sign_extension(targ[23:2]);
      IP[31:2] = IP[31:2] + temp[31:2];
      IP[1:0] = 0;
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Assume (AC.cc AND 100₂) ≠ 0
bl xyz                  # IP = xyz;
```

Opcode:

be	12H	CTRL
bne	15H	CTRL
bl	14H	CTRL
ble	16H	CTRL
bg	11H	CTRL
bge	13H	CTRL
bo	17H	CTRL
bno	10H	CTRL

See Also: **b, bx, bbc, bbs, COMPARE AND BRANCH<cc>, bal, balx, BRANCH<cc>**

6.2.12 bswap

Mnemonic:	**bswap**	Byte Swap	
Format:	**bswap**	*src1:src,* reg/lit	*src2:dst* reg

Description: Alters the order of bytes in a word, reversing its "endianess."

Copies bytes 3:0 of *src1* to *src2* reversing order of the bytes. Byte 0 of *src1* becomes byte 3 of *src2*, byte 1 of *src1* becomes byte 2 of *src2*, etc.

Action:

```
dst = (rotate_left(src 8) & 0x00FF00FF)
      +(rotate_left(src 24) & 0xFF00FF00);
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
                     # g8 = 0x89ABCDEF
bswap g8, g10        # Reverse byte order.
                     # g10 now 0xEFCDAB89
```

Opcode:	**bswap**	5ADH	REG

See Also: **scanbyte, rotate**

Notes: This core instruction is not implemented on Cx, Kx and Sx 80960 processors.

6.2.13 call

Mnemonic: **call** Call

Format: **call** *targ*
disp

Description: Calls a new procedure. *targ* operand specifies the IP of called procedure's first instruction. When using the Intel i960 processor assembler, *targ* must be a label.

In executing this instruction, the processor performs a local call operation as described in 7.1.3.1, Call Operation (pg. 7-6). As part of this operation, the processor saves the set of local registers associated with the calling procedure and allocates a new set of local registers and a new stack frame for the called procedure. Processor then goes to the instruction specified with *targ* and begins execution.

targ can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP.

Action:

```
# Wait for any uncompleted instructions to finish.
implicit_syncf();
temp =  (SP + (SALIGN*16 - 1)) & ~(SALIGN*16 - 1)
    # Round stack pointer to next boundary.
    # SALIGN=1 on 80960Rx.
RIP = IP;
if (register_set_available)
    allocate_new_frame( );
else
    {   save_register_set( );       # Save register set in memory at its FP.
        allocate_new_frame( );
    }
    #   Local register references now refer to new frame.
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
PFP = FP;
FP = temp;
SP = temp + 64;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
call xyz            # IP = xyz
```

Opcode: **call** 09H CTRL

See Also: **bal, calls, callx**

6.2.14 calls

Mnemonic: **calls** Call System

Format: **calls** *targ*
reg/lit

Description: Calls a system procedure. The *targ* operand gives the number of the procedure being called. For **calls**, the processor performs system call operation described in 7.5, SYSTEM CALLS (pg. 7-15). *targ* provides an index to a system procedure table entry from which the processor gets the called procedure's IP.

The called procedure can be a local or supervisor procedure, depending on system procedure table entry type. If it is a supervisor procedure, the processor switches to supervisor mode (if not already in this mode).

As part of this operation, processor also allocates a new set of local registers and a new stack frame for called procedure. If the processor switches to supervisor mode, the new stack frame is created on the supervisor stack.

Action:

```
# Wait for any uncompleted instructions to finish.
implicit_syncf();
If (targ > 259)
    generate_fault(PROTECTION.LENGTH);
temp = get_sys_proc_entry(sptbase + 48 + 4*targ);
           # sptbase is address of supervisor procedure table.

 if (register_set_available)
    allocate_new_frame( );
    else
    {     save_register_set( );    # Save a frame in memory at its FP.
          allocate_new_frame( );
          # Local register references now refer to new frame.
    }
RIP = IP;
IP[31:2] = effective_address(temp[31:2]);
IP[1:0] = 0;
if ((temp.type == local) ||  (PC.em == supervisor))
    {                # Local call or supervisor call from supervisor mode.
          tempa =  (SP + (SALIGN*16 - 1)) & ~(SALIGN*16 - 1)
          # Round stack pointer to next boundary.
          # SALIGN=1 on 80960Rx.
          temp.RRR = 000_2;
    }
    else             # Supervisor call from user mode.
    {     tempa = SSP;                  # Get Supervisor Stack pointer.
```

```
          temp.RRR = $010_2$ | PC.te;
          PC.em = supervisor;
          PC.te = temp.te;
     }
PFP = FP;
PFP.rrr = temp.RRR;
FP = tempa;
SP = tempa + 64;
```

Faults:	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	PROTECTION.LENGTH	Specifies a procedure number greater than 259.

Example:

```
calls r12          # IP = value obtained from
                   # procedure table for procedure
                   # number given in r12.
calls 3            # Call procedure 3.
```

Opcode:	**calls**	660H	REG

See Also: **bal, call, callx, ret**

6.2.15 callx

Mnemonic: **callx** Call Extended

Format: **callx** *targ*
mem

Description: Calls new procedure. *targ* specifies IP of called procedure's first instruction.

In executing **callx**, the processor performs a local call as described in 7.1.3.1, Call Operation (pg. 7-6). As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. Processor then goes to the instruction specified with *targ* and begins execution of new procedure.

callx performs the same operation as call except the target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP.

The *tar*g operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect calls can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 2, DATA TYPES AND MEMORY ADDRESSING MODES for more information.

Action:

```
#     Wait for any uncompleted instructions to finish;
implicit_syncf();
      temp =  (SP + (SALIGN*16 - 1)) & ~(SALIGN*16 - 1)
            # Round stack pointer to next boundary.
            # SALIGN=1 on 80960Rx.
RIP = IP;
if (register_set_available)
      allocate_new_frame( );
else
      {     save_register_set( );         # Save register set in memory at its FP;
            allocate_new_frame( );
      }
      #     Local register references now refer to new frame.
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
PFP = FP;
FP = temp;
SP = temp + 64;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
callx (g5)    # IP = (g5), where the address in g5
              # is the address of the new procedure.
```

Opcode:	**callx**	86H	MEM

See Also: **bal, call, calls, ret**

6.2.16 chkbit

Mnemonic:	**chkbit**	Check Bit	
Format:	**chkbit**	*bitpos*, reg/lit	*src2* reg/lit

Description: Checks bit in *src2* designated by *bitpos* and sets condition code according to value found. If bit is set, condition code is set to 010_2; if bit is clear, condition code is set to 000_2.

Action:

```
if (((src2 & 2**(bitpos % 32)) == 0)
    AC.cc = 000_2;
else
    AC.cc = 010_2;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
chkbit 13, g8      # Checks bit 13 in g8 and sets
                   # AC.cc according to the result.
```

Opcode: **chkbit** 5AEH REG

See Also: **alterbit, clrbit, notbit, setbit, cmpi, cmpo**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.17 clrbit

Mnemonic:	**clrbit**	Clear Bit		
Format:	**clrbit**	*bitpos*, reg/lit	*src*, reg/lit	*dst* reg
Description:	Copies *src* value to *dst* with one bit cleared. *bitpos* operand specifies bit to be cleared.			
Action:	dst = src & ~(2**(bitpos%32));			
Faults:	STANDARD		Refer to section 6.1.6, Faults (pg. 6-5).	
Example:	`clrbit 23, g3, g6   # g6 = g3 with bit 23 cleared.`			
Opcode:	**clrbit**	58CH	REG	
See Also:	**alterbit, chkbit, notbit, setbit**			

6.2.18 cmpdeci, cmpdeco

Mnemonic: **cmpdeci** Compare and Decrement Integer
cmpdeco Compare and Decrement Ordinal

Format:				
	cmpdec*	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Compares *src2* and *src1* values and sets the condition code according to comparison results. *src2* is then decremented by one and result is stored in *dst*. The following table shows condition code setting for the three possible results of the comparison.

Condition Code	Comparison
100_2	*src1* < *src2*
010_2	*src1* = *src2*
001_2	*src1* > *src2*

These instructions are intended for use in ending iterative loops. For **cmpdeci**, integer overflow is ignored to allow looping down through the minimum integer values.

Action:

```
if(src1 < src2)
    AC.cc = 100_2;
else if(src1 == src2)
    AC.cc = 010_2;
else
    AC.cc = 001_2;
dst = src2 -1;      # Overflow suppressed for cmpdeci.
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
cmpdeci 12, g7, g1  # Compares g7 with 12 and sets
                    # AC.cc to indicate the result
                    # g1 = g7 - 1.
```

Opcode:			
	cmpdeci	5A7H	REG
	cmpdeco	5A6H	REG

See Also: **cmpinco, cmpo, cmpi, cmpinci, COMPARE AND BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.19 cmpinci, cmpinco

Mnemonic:	**cmpinci** **cmpinco**	Compare and Increment Integer Compare and Increment Ordinal

Format:	**cmpinc***	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Compares *src2* and *src1* values and sets the condition code according to comparison results. *src2* is then incremented by one and result is stored in *dst*. The following table shows condition code settings for the three possible comparison results.

Condition Code	Comparison
100_2	*src1 < src2*
010_2	*src1 = src2*
001_2	*src1 > src2*

These instructions are intended for use in ending iterative loops. For **cmpinci**, integer overflow is ignored to allow looping up through the maximum integer values.

Action:

```
if (src1 < src2)
    AC.cc = 100₂;
else if (src1 == src2)
    AC.cc = 010₂;
else
    AC.cc = 001₂;

dst = src2 + 1;    # Overflow suppressed for cmpinci.
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
cmpinco r8, g2, g9  # Compares the values in g2
                    # and r8 and sets AC.cc to
                    # indicate the result:
                    # g9 = g2 + 1
```

Opcode:	**cmpinci**	5A5H	REG
	cmpinco	5A4H	REG

See Also: **cmpdeco, cmpo, cmpi, cmpdeci, COMPARE AND BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.20 COMPARE

Mnemonic:	**cmpi**	Compare Integer
	cmpib	Compare Integer Byte
	cmpis	Compare Integer Short
	cmpo	Compare Ordinal
	cmpob	Compare Ordinal Byte
	cmpos	Compare Ordinal Short

Format:	**cmp***	*src1*,	*src2*
		reg/lit	reg/lit

Description: Compares *src2* and *src1* values and sets condition code according to comparison results. The following table shows condition code settings for the three possible comparison results.

Condition Code	Comparison
100_2	*src1* < *src2*
010_2	*src1* = *src2*
001_2	*src1* > *src2*

cmpi* followed by a branch-if instruction is equivalent to a compare-integer-and-branch instruction. The latter method of comparing and branching produces more compact code; however, the former method can execute byte and short compares without masking. The same is true for **cmpo*** and the compare-ordinal-and-branch instructions.

Action:

```
# For cmpo, cmpi, N = 31.
# For cmpos, cmpis, N = 15.
# For cmpob, cmpib, N = 7.

if (src1[N:0] < src2[N:0])
    AC.cc = $100_2$;
else if (src1[N:0] == src2[N:0])
    AC.cc = $010_2$;
else if (src1[N:0] > src2[N:0])
    AC.cc = $001_2$;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
cmpo r9, 0x10     # Compares the value in r9 with 0x10
                  # and sets AC.cc to indicate the
                  # result.
bg xyz            # Branches to xyz if the value of r9
                  # was greater than 0x10.
```

Opcode:

cmpi	5A1H	REG
cmpib	595H	REG
cmpis	597H	REG
cmpo	5A0H	REG
cmpob	594H	REG
cmpos	596H	REG

See Also: **COMPARE AND BRANCH<cc>, cmpdeci, cmpdeco, cmpinci, cmpinco, concmpi, concmpo**

Side Effects: Sets the condition code in the arithmetic controls.

Notes: The core instructions **cmpib**, **cmpis**, **compob** and **compos** are not implemented on i960 Cx, Kx and Sx processors.

6.2.21 COMPARE AND BRANCH<cc>

Mnemonic:	**cmpibe**	Compare Integer and Branch If Equal
	cmpibne	Compare Integer and Branch If Not Equal
	cmpibl	Compare Integer and Branch If Less
	cmpible	Compare Integer and Branch If Less Or Equal
	cmpibg	Compare Integer and Branch If Greater
	cmpibge	Compare Integer and Branch If Greater Or Equal
	cmpibo	Compare Integer and Branch If Ordered
	cmpibno	Compare Integer and Branch If Not Ordered
	cmpobe	Compare Ordinal and Branch If Equal
	cmpobne	Compare Ordinal and Branch If Not Equal
	cmpobl	Compare Ordinal and Branch If Less
	cmpoble	Compare Ordinal and Branch If Less Or Equal
	cmpobg	Compare Ordinal and Branch If Greater
	cmpobge	Compare Ordinal and Branch If Greater Or Equal

Format:	**cmpib***	*src1*,	*src2*,	*targ*
		reg/lit	reg	disp
	cmpob*	*src1*,	*src2*,	*targ*
		reg/lit	reg	disp

Description: Compares *src2* and *src1* values and sets AC register condition code according to comparison results. If logical AND of condition code and mask part of opcode is not zero, the processor branches to instruction specified with *targ*; otherwise, the processor goes to next instruction.

targ can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from current IP. When using the Intel i960 processor assembler, *targ* must be a label that specifies target instruction's IP.

Functions these instructions perform can be duplicated with a **cmpi** or **cmpo** followed by a branch-if instruction, as described in section 6.2.20, COMPARE (pg. 6-32).

The following table shows the condition-code mask for each instruction. The mask is in bits 0-2 of the opcode.

Instruction	Mask	Branch Condition
cmpibno	000_2	No Condition
cmpibg	001_2	*src1 > src2*
cmpibe	010_2	*src1 = src2*
cmpibge	011_2	*src1* ≥ *src2*
cmpibl	100_2	*src1 < src2*
cmpibne	101_2	*src1* ≠ *src2*
cmpible	110_2	*src1* ≤ *src2*
cmpibo	111_2	Any Condition
cmpobg	001_2	*src1 > src2*
cmpobe	010_2	*src1 = src2*
cmpobge	011_2	*src1* ≥ *src2*
cmpobl	100_2	*src1 < src2*
cmpobne	101_2	*src1* ≠ *src2*
cmpoble	110_2	*src1* ≤ *src2*

cmpibo always branches; **cmpibno** never branches.

Action:

```
if(src1 < src2)
    AC.cc = $100_2$;
else if(src1 == src2)
    AC.cc = $010_2$;
else
    AC.cc = $001_2$;
if((mask && AC.cc) != $000_2$)
    IP[31:2] = efa[31:2];        # Resume execution at the new IP.
    IP[1:0] = 0;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Assume g3 < g9
cmpibl g3, g9, xyz  # g9 is compared with g3;
                    # IP = xyz.
# assume 19 ≥ r7
cmpobge 19, r7, xyz # 19 is compared with r7;
                    # IP = xyz.
```

Opcode:	**cmpibe**	3AH	COBR
	cmpibne	3DH	COBR
	cmpibl	3CH	COBR
	cmpible	3EH	COBR
	cmpibg	39H	COBR
	cmpibge	3BH	COBR
	cmpibo	3FH	COBR
	cmpibno	38H	COBR
	cmpobe	32H	COBR
	cmpobne	35H	COBR
	cmpobl	34H	COBR
	cmpoble	36H	COBR
	cmpobg	31H	COBR
	cmpobge	33H	COBR

See Also: **BRANCH<cc>, cmpi, cmpo, bal, balx**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.22 concmpi, concmpo

Mnemonic:	**concmpi**	Conditional Compare Integer
	concmpo	Conditional Compare Ordinal

Format:	**concmp***	*src1*, reg/lit	*src2* reg/lit

Description: Compares *src2* and *src1* values if condition code bit 2 is not set. If comparison is performed, condition code is set according to comparison results. Otherwise, condition codes are not altered.

These instructions are provided to facilitate bounds checking by means of two-sided range comparisons (e.g., is A between B and C?). They are generally used after a compare instruction to test whether a value is inclusively between two other values.

The example below illustrates this application by testing whether g3 value is between g5 and g6 values, where g5 is assumed to be less than g6. First a comparison (**cmpo**) of g3 and g6 is performed. If g3 is less than or equal to g6 (i.e., condition code is either 010_2 or 001_2), a conditional comparison (**concmpo**) of g3 and g5 is then performed. If g3 is greater than or equal to g5 (indicating that g3 is within the bounds of g5 and g6), condition code is set to 010_2; otherwise, it is set to 001_2.

Action:

```
if (AC.cc != 1XX_2)
{     if(src1 <= src2)
              AC.cc = 010_2;
      else
              AC.cc  = 001_2;
}
```

Faults: STANDARD   Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
cmpo g6, g3         # Compares g6 and g3
                    # and sets AC.cc.
concmpo g5, g3      # If AC.cc < 100_2 (g6 ≥ g3)
                    # g5 is compared with g3.
```

At this point, depending on the register ordering, the condition code is one of those listed on Table 6-5.

Table 6-5. concmpo Example: Register Ordering and CC

Order	CC
g5 < g6 < g3	100_2
g5 < g6 = g3	010_2
g5 < g3 < g6	010_2
g5 = g3 < g6	010_2
g3 < g5 < g6	001_2

Opcode:	**concmpi**	5A3H	REG
	concmpo	5A2H	REG

See Also: **cmpo, cmpi, cmpdeci, cmpdeco, cmpinci, cmpinco, COMPARE AND BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.23 dcctl

Mnemonic: **dcctl** Data-cache Control

Format: *src1,* reg/lit *src2,* reg/lit *src/dst* reg

Description: Performs management and control of the data cache including disabling, enabling, invalidating, ensuring coherency, getting status, and storing cache contents to memory. Operations are indicated by the value of *src1*. *src2* and *src/dst* are also used by some operations. When needed by the operation, the processor orders the effects of the operation with previous and subsequent operations to ensure correct behavior.

Table 6-6. dcctl Operand Fields

Function	*src1*	*src2*	*src/dst*
Disable D-cache	0	NA	NA
Enable D-cache	1	NA	NA
Global invalidate D-cache	2	NA	NA
Ensure cache coherency[1]	3	NA	NA
Get D-cache status	4	NA	*src:* NA *dst:* Receives D-cache status (see Figure 6-1).
Reserved	5	NA	NA
Store D-cache to memory	6	Destination address for cache sets	*src*: D-cache set #'s to be stored (see Figure 6-1).
Reserved	7	NA	NA
Quick invalidate	8	1	NA
Reserved	9	NA	NA

[1] Invalidates data cache on 80960Rx.

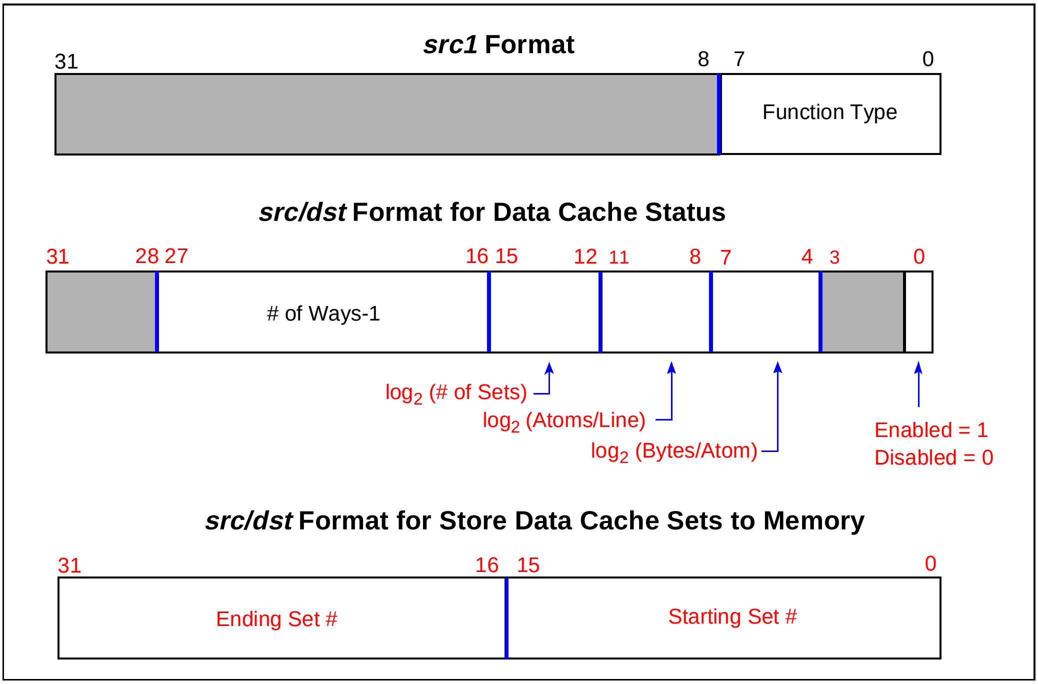
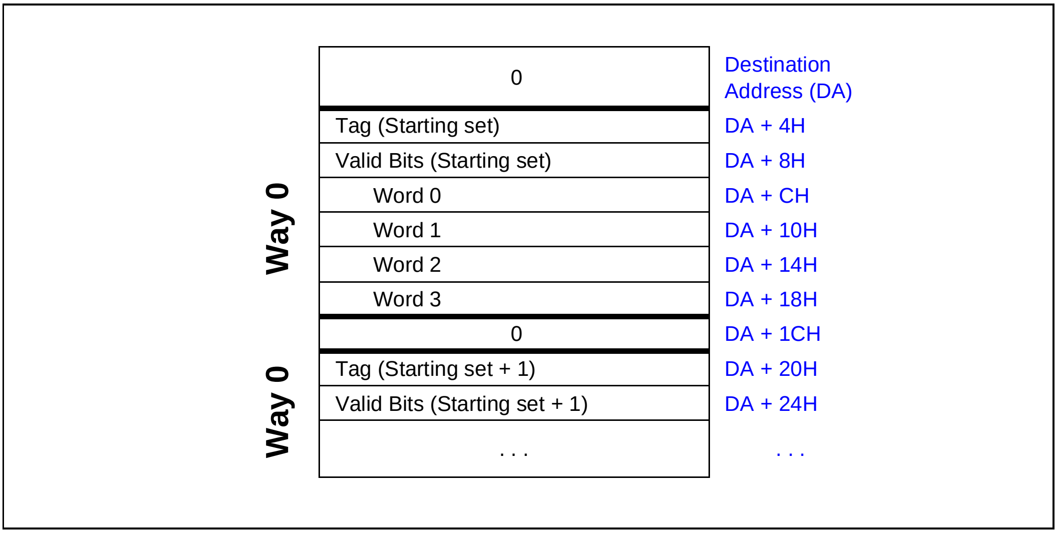


Figure 6-1. dcctl *src1* and *src/dst* Formats

Table 6-7. dcctl Status Values and D-Cache Parameters

Value	Value on 80960Rx
bytes per atom	4
atoms per line	4
number of sets	128 (full)
number of ways	1 (Direct)
cache size	2-Kbytes(full)
Status[0] (enable / disable)	0 or 1
Status[1:3] (reserved)	0
Status[7:4] ($\log_2$(bytes per atom))	2
Status[11:8] ($\log_2$(atoms per line))	2
Status[15:12] ($\log_2$(number of sets))	7 (full)
Status[27:16] (number of ways - 1)	0



	0	Destination Address (DA)
Way 0	Tag (Starting set)	DA + 4H
	Valid Bits (Starting set)	DA + 8H
	Word 0	DA + CH
	Word 1	DA + 10H
	Word 2	DA + 14H
	Word 3	DA + 18H
	0	DA + 1CH
Way 0	Tag (Starting set + 1)	DA + 20H
	Valid Bits (Starting set + 1)	DA + 24H
	. . .	. . .

Figure 6-2. Store Data Cache to Memory Output Format

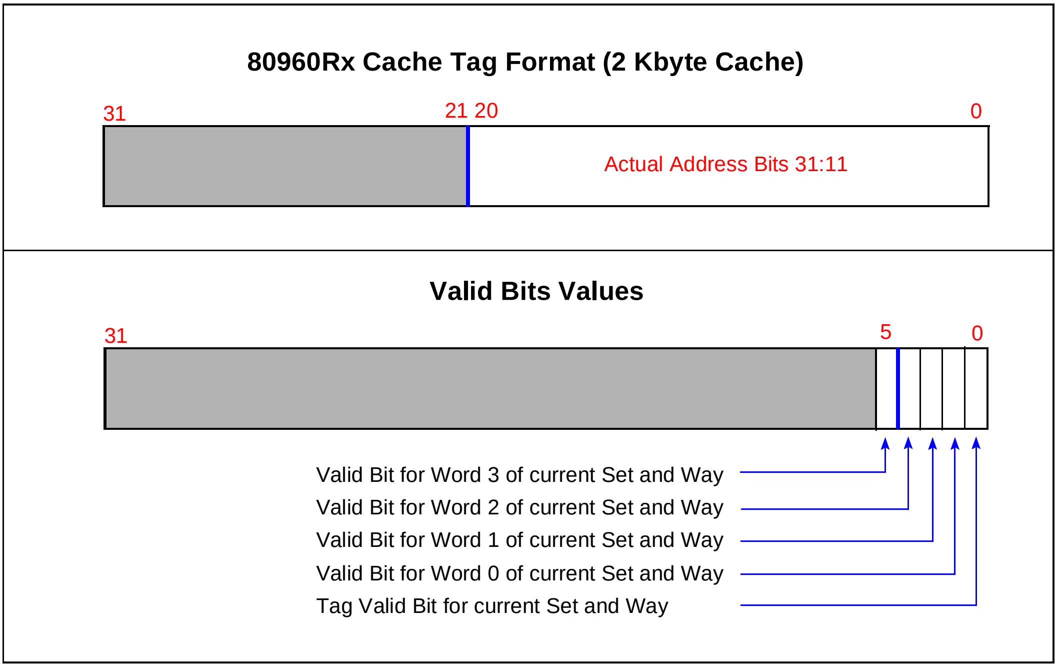

80960Rx Cache Tag Format (2 Kbyte Cache)
31
21 20
0
Actual Address Bits 31:11
Valid Bits Values
31
5
0
Valid Bit for Word 3 of current Set and Way
Valid Bit for Word 2 of current Set and Way
Valid Bit for Word 1 of current Set and Way
Valid Bit for Word 0 of current Set and Way
Tag Valid Bit for current Set and Way

Figure 6-3. D-Cache Tag and Valid Bit Formats

Action:

```
if (PC.em != supervisor)
    generate_fault(TYPE.MISMATCH);
order_wrt(previous_operations);
switch (src1[7:0]) {

    case 0:     # Disable data cache.
                disable_Dcache( );
                break;
    case 1:     # Enable data cache.
                enable_Dcache( );
                break;
    case 2:     # Global invalidate data cache.
                invalidate_Dcache( );
                break;
    case 3:     # Ensure coherency of data cache with memory.
                # Causes data cache to be invalidated on this processor.
                ensure_Dcache_coherency( );
                break;
    case 4:     # Get data cache status into src_dst.
                if (Dcache_enabled) src_dst[0] = 1;
                else src_dst[0] = 0;
                # Atom is 4 bytes.
                src_dst[7:4] = log2(bytes per atom);
                # 4 atoms per line.
                src_dst[11:8] = log2(atoms per line);
                src_dst[15:12] = log2(number of sets);
                src_dst[27:16] = number of ways-1; # in lines per set
                # cache size = ([27:16]+1) << ([7:4] + [11:8] + [15:12]).
                break;
case 6:      # Store data cache sets to memory pointed to by src2.
                start = src_dst[15:0]          #    Starting set number.
                end   = src_dst[31:16]         #    Ending set number.
                                               #    (zero-origin).
                if (end >= Dcache_max_sets) end = Dcache_max_sets - 1;
                if (start > end) generate_fault
                                (OPERATION.INVALID_OPERAND);
                memadr = src2;                 #    Must be word-aligned.
                if (0x3 & memadr! = 0)
                generate_fault(OPERATION.INVALID_OPERAND)
                for (set = start; set <= end; set++){
                    # Set_Data is described at end of this code flow.
                    memory[memadr] = Set_Data[set];
                    memadr += 4;
                    for (way = 0; way < numb_ways; way++)
                        {memory[memadr] = tags[set][way];
                         memadr += 4;
                         memory[memadr] = valid_bits[set][way];
                         memadr += 4;
                        for (word = 0; word < words_in_line; word++)
                            {memory[memadr] =
                                        Dcache_line[set][way][word];
                             memadr += 4;
                            }
                            }
                }
                break;
    default:    # Reserved.
                generate_fault(OPERATION.INVALID_OPERAND);
                break;
    }
order_wrt(subsequent_operations)
```

Faults:	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.
	OPERATION.INVALID_OPERAND	

Example:

```
                        # g0 = 6, g1 = 0x10000000,
                        # g2 = 0x001F0001
dcctl g0,g1,g2          # Store the status of D-cache
                        # sets 1-0x1F to memory starting
                        # at 0x10000000.
```

Opcode:	**dcctl**	65CH	REG

See Also: **sysctl**

Notes: DCCTL function 6 stores data-cache sets to a target range in external memory. For any memory location that is cached and also within the target range for function 6, the corresponding word-valid bit will be cleared after function 6 completes to ensure data-cache coherency. Thus, **dcctl** function 6 can alter the state of the cache after it completes, but only the word-valid bits. In all cases, even when the cache sets to store to external memory overlap the cache sets that map the target range in external memory, DCCTL function 6 always returns the state of the cache as it existed when the DCCTL was issued.

This instruction is implemented on the 80960Rx, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.24 divi, divo

Mnemonic:	**divi**	Divide Integer	
	divo	Divide Ordinal	
Format:	**div***	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Divides *src2* value by *src1* value and stores the result in *dst*. Remainder is discarded.

For **divi**, an integer-overflow fault can be signaled.

Action:

divo:

```
if (src1 == 0)
{       dst = undefined_value;
        generate_fault (ARITHMETIC.ZERO_DIVIDE);
else
        dst = src2/src1;
```

divi:

```
if (src1 == 0)
{       dst = undefined_value;
        generate_fault (ARITHMETIC.ZERO_DIVIDE);}
else if ((src2 == -2**31) && (src1 == -1))
        {               dst = -2**31

        if (AC.om == 1)
                        AC.of  = 1;
        else
                generate_fault (ARITHMETIC.OVERFLOW);
        }
else
        dst  = src2 / src1;
```

Faults:

STANDARD	Refer to Section 6.1.6 on page 6-5.
ARITHMETIC.ZERO_DIVIDE	The *src1* operand is 0.
ARITHMETIC.OVERFLOW	Result too large for destination register (**divi** only). If overflow occurs and AC.om=1, fault is suppressed and AC.of is set to 1. Result's least significant 32 bits are stored in *dst*.

Example:

```
divo r3, r8, r13    # r13 = r8/r3
```

Opcode:	**divi**	74BH	REG
	divo	70BH	REG

See Also: **ediv, mulo, muli, emul**

6.2.25 ediv

Mnemonic:	**ediv**	Extended Divide		
Format:	**ediv**	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description:

Divides *src2* by *src1* and stores result in *dst*. The *src2* value is a long ordinal (64 bits) contained in two adjacent registers. *src2* specifies the lower numbered register which contains operand's least significant bits. *src2* must be an even numbered register (i.e., g0, g2, ... or r4, r6, r8...). *src1* value is a normal ordinal (i.e., 32 bits).

The result consists of a one-word remainder and a one-word quotient. Remainder is stored in the register designated by *dst*; quotient is stored in the next highest numbered register. *dst* must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ...).

This instruction performs ordinal arithmetic.

If this operation overflows (quotient or remainder do not fit in 32 bits), no fault is raised and the result is undefined.

Action:

```
if((reg_number(src2)%2 != 0) || (reg_number(dst)%2 != 0))
{     dst[0] = undefined_value;
      dst[1] = undefined_value;
      generate_fault (OPERATION.INVALID_OPERAND);
}
else if(src1 == 0)
{     dst[0] = undefined_value;
      dst[1] = undefined_value;
      generate_fault(ARITHMETIC.DIVIDE_ZERO);
}
else        # Quotient
{     dst[1] = ((src2 + reg_value(src2[1]) * 2**32) / src1)[31:0];
      #Remainder
      dst[0] = (src2 + reg_value(src2[1]) * 2**32
            - ((src2 + reg_value(src2[1]) * 2**32 / src1) * src1);
}
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
ARITHMETIC.ZERO_DIVIDE	The *src1* operand is 0.

Example:

```
ediv g3, g4, g10    # g10 = remainder of g4,g5/g3
                    # g11 = quotient of g4,g5/g3
```

Opcode:	**ediv**	671H	REG
See Also:	**emul, divi, divo**		

6.2.26 emul

Mnemonic:	**emul**	Extended Multiply		
Format:	**emul**	*src1,* reg/lit	*src2,* reg/lit	*dst* reg

Description: Multiplies *src2* by *src1* and stores the result in *dst*. Result is a long ordinal (64 bits) stored in two adjacent registers. *dst* specifies lower numbered register, which receives the result's least significant bits. *dst* must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ...).

This instruction performs ordinal arithmetic.

Action:

```
if(reg_number(dst)%2 != 0)
{    dst[0] = undefined_value;
     dst[1] = undefined_value;
     generate_fault(OPERATION.INVALID_OPERAND);
}
else
{    dst[0] = (src1 * src2)[31:0];
     dst[1] = (src1 * src2)[63:32];
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
emul r4, r5, g2     # g2,g3 = r4 * r5.
```

Opcode: **emul** 670H REG

See Also: **ediv, muli, mulo**

6.2.27 eshro

Mnemonic:	**eshro**	Extended Shift Right Ordinal		
Format:	**eshro**	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Shifts *src2* right by (*src1* **mod** 32) places and stores the result in *dst*. Bits shifted beyond the least-significant bit are discarded.

src2 value is a long ordinal (i.e., 64 bits) contained in two adjacent registers. *src2* operand specifies the lower numbered register, which contains operand's least significant bits. *src2* operand must be an even numbered register (i.e., r4, r6, r8, ... or g0, g2).

src1 operand is a single 32-bit register or literal where the lower 5 bits specify the number of places that the *src2* operand is to be shifted.

The least significant 32 bits of the shift operation result are stored in *dst*.

Action:

```
if(reg_number(src2)%2 != 0)
{     dst[0] = undefined_value;
      dst[1] = undefined_value;
      generate_fault(OPERATION.INVALID_OPERAND);
}
else
      dst = shift_right((src2 + reg_value(src2[1]) * 2**32),(src1%32))[31:0];
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
eshro g3, g4, g11   # g11 = g4,5 shifted right by
                    # (g3 MOD 32).
```

Opcode:	**eshro**	5D8H	REG

See Also: **SHIFT, extract**

Notes: This core instruction is not implemented on the Kx and Sx 80960 processors.

6.2.28 extract

Mnemonic:	**extract**	Extract		
Format:	**extract**	*bitpos* reg/lit	*len* reg/lit	*src/dst* reg

Description: Shifts a specified bit field in *src/dst* right and zero fills bits to left of shifted bit field. *bitpos* value specifies the least significant bit of the bit field to be shifted; *len* value specifies bit field length.

Action:

```
src_dst = (src_dst >> min(bitpos, 32))
      & ~ (0xFFFFFFFF << len);
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
extract 5, 12, g4   # g4 = g4 with bits 5 through
                    # 16 shifted right.
```

Opcode: **extract** 651H REG

See Also: **modify**

6.2.29 FAULT<cc>

Mnemonic:

faulte	Fault If Equal
faultne	Fault If Not Equal
faultl	Fault If Less
faultle	Fault If Less Or Equal
faultg	Fault If Greater
faultge	Fault If Greater Or Equal
faulto	Fault If Ordered
faultno	Fault If Not Ordered

Format: **fault***

Description: Raises a constraint-range fault if the logical AND of the condition code and opcode's mask part is not zero. For **faultno** (unordered), fault is raised if condition code is equal to 000_2.

faulto and **faultno** are provided for use by implementations with a floating point coprocessor. They are used for compare and branch (or fault) operations involving real numbers.

The following table shows the condition-code mask for each instruction. The mask is opcode bits 0-2.

Instruction	Mask	Condition
faultno	000_2	Unordered
faultg	001_2	Greater
faulte	010_2	Equal
faultge	011_2	Greater or equal
faultl	100_2	Less
faultne	101_2	Not equal
faultle	110_2	Less or equal
faulto	111_2	Ordered

Action:

For all except faultno:

```
if(mask && AC.cc != 000₂)
    generate_fault(CONSTRAINT.RANGE);
```

faultno:

```
if(AC.cc == 000₂)
    generate_fault(CONSTRAINT.RANGE);
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
CONSTRAINT.RANGE	If condition being tested is true.

Example: # Assume (AC.cc AND 110_2)$\neq$$000_2$
faultle # Generate CONSTRAINT_RANGE fault

Opcode:

faulte	1AH	CTRL
faultne	1DH	CTRL
faultl	1CH	CTRL
faultle	1EH	CTRL
faultg	19H	CTRL
faultge	1BH	CTRL
faulto	1FH	CTRL
faultno	18H	CTRL

See Also: **BRANCH<cc>, TEST<cc>**

6.2.30 flushreg

Mnemonic:	**flushreg**	Flush Local Registers
Format:	**flushreg**	

Description: Copies the contents of every cached register set, except the current set, to its associated stack frame in memory. The entire register cache is then marked as purged (or invalid). On a return to a stack frame for which the local registers are not cached, the processor reloads the locals from memory.

flushreg is provided to allow a debugger or application program to circumvent the processor's normal call/return mechanism. For example, a debugger may need to go back several frames in the stack on the next return, rather than using the normal return mechanism that returns one frame at a time. Since the local registers of an unknown number of previous stack frames may be cached, a **flushreg** must be executed prior to modifying the PFP to return to a frame other than the one directly below the current frame.

To reduce interrupt latency, **flushreg** is abortable. If an interrupt of higher priority than the current process is detected while **flushreg** is executing, **flushreg** flushes at least one frame and aborts. After executing the interrupt handler, the processor returns to the **flushreg** instruction and re-executes it. **flushreg** does not reflush any frames that were flushed before the interrupt occurred. **flushreg** is not aborted by high priority interrupts if tracing is enabled in the PC or if any faults are pending at the time of the interrupt.

Action: Each local cached register set except the current one is flushed to its associated stack frame in memory and marked as purged, meaning that they are reloaded from memory if and when they become the current local register set.

Faults:	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
flushreg
```

Opcode:	**flushreg**	66DH	REG

6.2.31 fmark

Mnemonic: **fmark** Force Mark

Format: **fmark**

Description: Generates a mark trace event. Causes a mark trace event to be generated, regardless of mark trace mode flag setting, providing the trace enable bit, bit 0 in the Process Controls, is set.

For more information on trace fault generation, refer to CHAPTER 10, TRACING AND DEBUGGING.

Action: A mark trace event is generated, independent of the setting of the mark-trace-mode flag.

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
TRACE.MARK	A TRACE.MARK fault is generated if PC.te=1.

Example:

```
# Assume PC.te = 1
fmark
# Mark trace event is generated at this point in the
# instruction stream.
```

Opcode: **fmark** 66CH REG

See Also: **mark**

6.2.32 halt

Mnemonic: **halt** Halt CPU

Format: **halt** *src1*
reg/lit

Description: Causes the i960 core processor to enter HALT mode. Entry into Halt mode allows the interrupt enable state to be conditionally changed based on the value of *src1*.

src1	Operation
0	Disable interrupts and halt
1	Enable interrupts and halt
2	Use current interrupt enable state and halt

The processor exits Halt mode on a hardware reset or upon receipt of an interrupt that should be delivered based on the current process priority. After executing the interrupt that forced the processor out of Halt mode, execution resumes at the instruction immediately after the **halt** instruction. The processor must be in supervisor mode to use this instruction.

Action:

```
implicit_syncf;
if (PC.em != supervisor)
  generate_fault( TYPE.MISMATCH);
switch(src1) {
     case 0:      # Disable interrupts. set ICON.gie.
                  global_interrupt_enable = true;              break;
     case 1:      # Enable interrupts. clear ICON.gie.
                  global_interrupt_enable = false;             break;
     case 2:      # Use the current interrupt enable state.
                  break;
     default:
                  generate_fault( OPERATION.INVALID_OPERAND );
                  break;
}

ensure_bus_is_quiescient;
enter_HALT_mode;
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
                  # ICON.gie = 1, g0 = 1, Interrupts disabled.
halt g0           # Enable interrupts and halt.
```

Opcode: **halt** 65DH REG

Notes: This instruction is implemented on the 80960Rx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.33 icctl

Mnemonic:	**icctl**	Instruction-cache Control		
Format:	**icctl**	*src1*, reg/lit	*src2*, reg/lit	*src/dst* reg

Description: Performs management and control of the instruction cache including disabling, enabling, invalidating, loading and locking, getting status, and storing cache sets to memory. Operations are indicated by the value of *src1*. Some operations also use *src2* and *src/dst*. When needed by the operation, the processor orders the effects of the operation with previous and subsequent operations to ensure correct behavior. For specific function setup, see the following tables and diagrams:

Table 6-8. icctl Operand Fields

Function	*src1*	*src2*	*src/dst*
Disable I-cache	0	NA	NA
Enable I-cache	1	NA	NA
Invalidate I-cache	2	NA	NA
Load and lock I-cache	3	*src*: Starting address of code to lock.	Number of blocks to lock.
Get I-cache status	4	NA	*dst:* Receives status (see Figure 6-4).
Get I-cache locking status	5	NA	*dst:* Receives status (see Figure 6-4)
Store I-cache sets to memory	6	Destination address for cache sets	*src*: I-cache set #'s to be stored (see Figure 6-4).

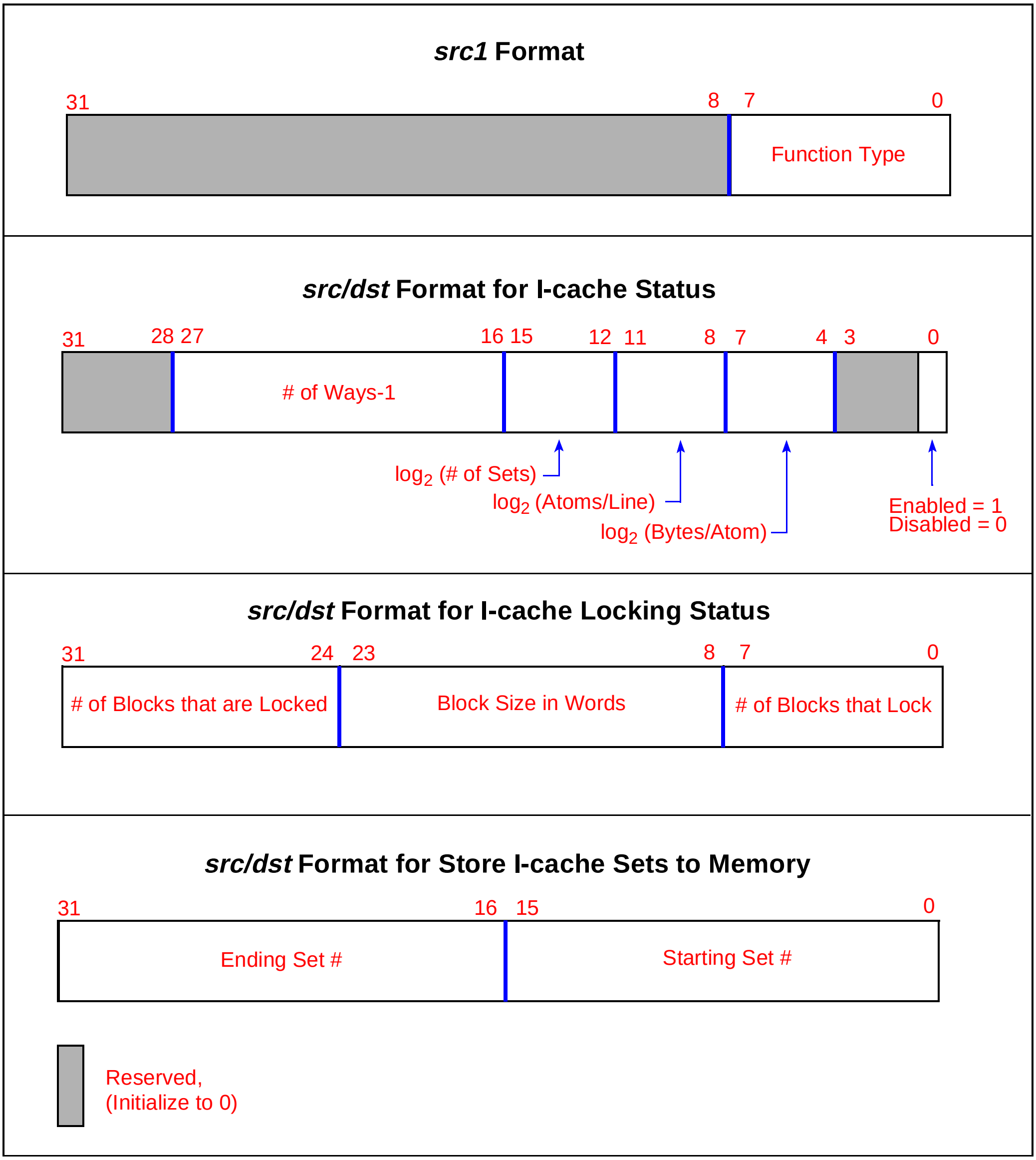
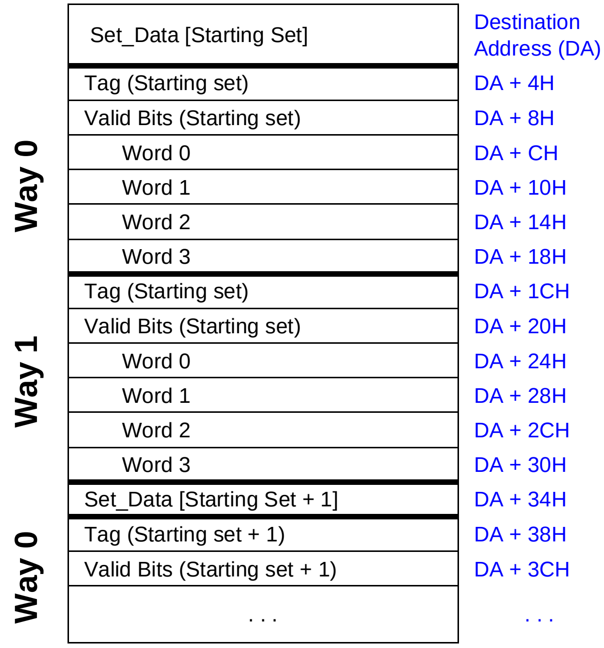


Figure 6-4. icctl *src1* and *src/dst* Formats

Table 6-9. icctl Status Values and I-Cache Parameters

Value	Value on i960RP CPU
bytes per atom	4
atoms per line	4
number of sets	128
number of ways	2
cache size	4-Kbytes
Status[0] (enable / disable)	0 or 1
Status[1:3] (reserved)	0
Status[7:4] (log2(bytes per atom))	2
Status[11:8] (log2(atoms per line))	2
Status[15:12] (log2(number of sets))	7
Status[27:16] (number of ways - 1)	1
Lock Status[7:0] (number of blocks that lock)	1
Lock Status[23:8] (block size in words)	512
Lock Status[31:24] (number of blocks that are locked)	0 or 1



	Set_Data [Starting Set]	Destination Address (DA)
Way 0	Tag (Starting set)	DA + 4H
	Valid Bits (Starting set)	DA + 8H
	Word 0	DA + CH
	Word 1	DA + 10H
	Word 2	DA + 14H
	Word 3	DA + 18H
Way 1	Tag (Starting set)	DA + 1CH
	Valid Bits (Starting set)	DA + 20H
	Word 0	DA + 24H
	Word 1	DA + 28H
	Word 2	DA + 2CH
	Word 3	DA + 30H
	Set_Data [Starting Set + 1]	DA + 34H
Way 0	Tag (Starting set + 1)	DA + 38H
	Valid Bits (Starting set + 1)	DA + 3CH
	. . .	. . .

Figure 6-5. Store Instruction Cache to Memory Output Format

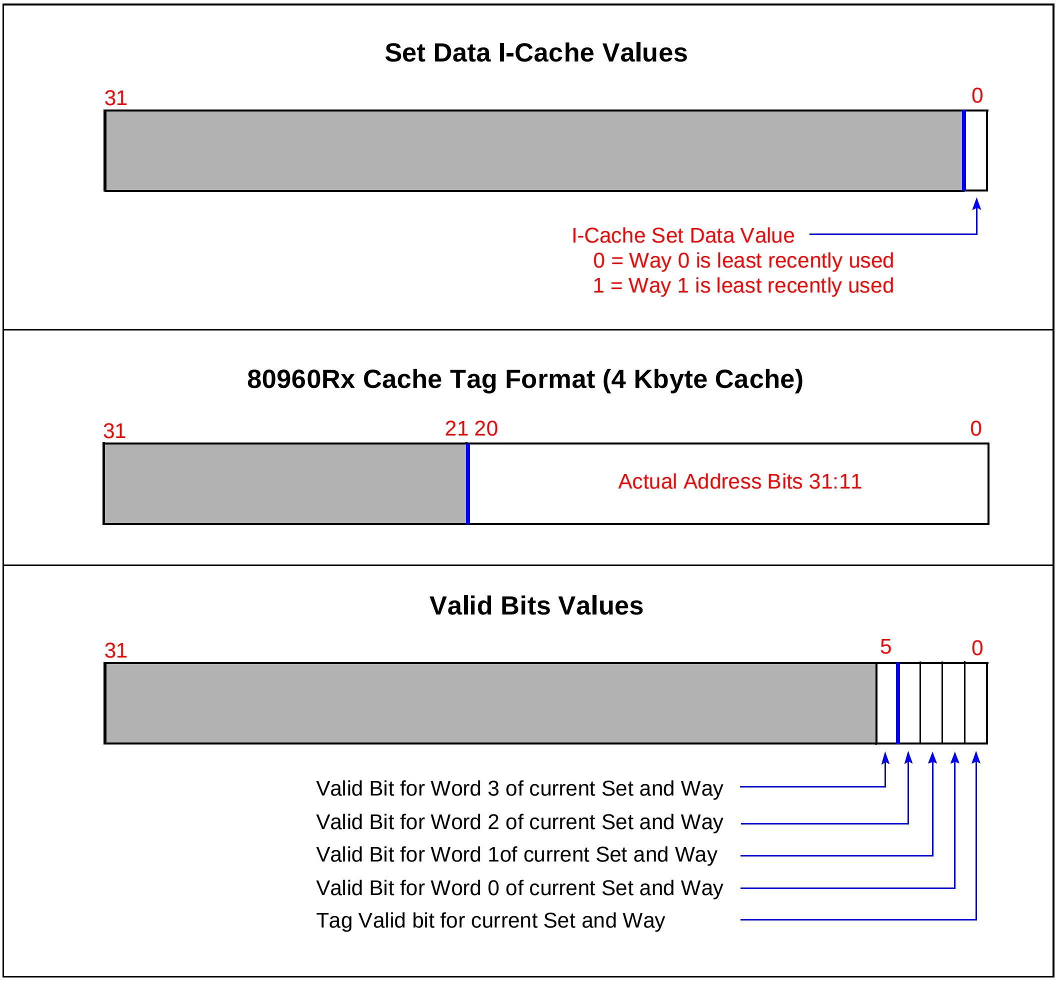

Set Data I-Cache Values
31
0
I-Cache Set Data Value
0 = Way 0 is least recently used
1 = Way 1 is least recently used
80960Rx Cache Tag Format (4 Kbyte Cache)
31
21 20
0
Actual Address Bits 31:11
Valid Bits Values
31
5
0
Valid Bit for Word 3 of current Set and Way
Valid Bit for Word 2 of current Set and Way
Valid Bit for Word 1of current Set and Way
Valid Bit for Word 0 of current Set and Way
Tag Valid bit for current Set and Way

Figure 6-6. I-Cache Set Data, Tag and Valid Bit Formats

Action:

```
if (PC.em != supervisor)
        generate_fault(TYPE.MISMATCH);
switch (src1[7:0]) {
        case 0:         # Disable instruction cache.
                        disable_instruction_cache( );
                        break;
        case 1:         # Enable instruction cache.
                        enable_instruction_cache( );
                        break;
        case 2:         # Globally invalidate instruction cache.
                        # Includes locked lines also.
                        invalidate_instruction_cache( );
                        unlock_icache( );
                        break;
        case 3:         # Load & Lock code into Instruction-Cache
                        # src_dst has number of contiguous blocks to lock.
                        # src2 has starting address of code to lock.
                        # On the i960 RP, src2 is aligned to a quad word boundary
                                aligned_addr = src2 & 0xFFFFFFF0;
                                invalidate(I-cache); unlock(I-cache);
                                for (j = 0; j < src_dst; j++)
                                        {       way = way_associated_with_block(j);
                                                start = src2 + j*block_size;
                                                end = start + block_size;
                                                for (i = start; i < end; i=i+4)
                                                        {   set = set_associated_with(i);
                                                            word = word_associated_with(i);
                                                            Icache_line[set][way][word] =
                                                                                        memory[i];
                                                        update_tag_n_valid_bits(set,way,word)
                                                            lock_icache(set,way,word);
                                                        } } break;
        case 4:         # Get instruction cache status into src_dst.
                        if (Icache_enabled) src_dst[0] = 1;
                                else src_dst[0] = 0;
                        # Atom is 4 bytes.
                          src_dst[7:4] = log2(bytes per atom);
                        # 4 atoms per line.
                          src_dst[11:8] = log2(atoms per line);
                        src_dst[15:12] = log2(number of sets);
                        src_dst[27:16] = number of ways-1; #in lines per set
                        # cache size = ([27:16]+1) << ([7:4] + [11:8] + [15:12])
                        break;
case 5:     # Get instruction cache locking status into dst.
            src_dst[7:0] = number_of_blocks_that_lock;
            src_dst[23:8] = block_size_in_words;
            src_dst[31:24] = number_of_blocks_that_are_locked;
            break;
case 6:     # Store instr cache sets to memory pointed to by src2.
            start = src_dst[15:0]       # Starting set number
            end   = src_dst[31:16]      # Ending set number
                                        # (zero-origin).
            if (end >= Icache_max_sets)
                 end = Icache_max_sets - 1;
            if (start > end)

generate_fault(OPERATION.INVALID_OPERAND);
            memadr = src2;              # Must be word-aligned.
            if(0x3 & memadr != 0)

generate_fault(OPERATION.INVALID_OPERAND);
            for (set = start; set <= end; set++){
                 # Set_Data is described at end of this code flow.
                 memory[memadr] = Set_Data[set];
                 memadr += 4;
                 for (way = 0; way < numb_ways; way++)
                          {memory[memadr] = tags[set][way];
                           memadr += 4;
                           memory[memadr] = valid_bits[set][way];
                           memadr += 4;
                           for (word = 0; word < words_in_line;
                                                             word++)
                                {memory[memadr] =
                                          Icache_line[set][way][word];
                                 memadr += 4;
                                }
                          } } break;

default:    # Reserved.
            generate_fault(OPERATION.INVALID_OPERAND);
            break;}
```

Faults:		
	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
                                    # g0 = 3, g1=0x10000000, g2=1
         icctl g0,g1,g2             # Load and lock 1 block of cache
                                    # (one way) with
                                    # location of code at starting
                                    # 0x10000000.
```

Opcode: **icctl** 65BH REG

See Also: **sysctl**

Notes: This instruction is implemented on the 80960Rx, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.34 intctl

Mnemonic: **intctl** Global Enable and Disable of Interrupts

Format: **intctl** *src1* *dst*

reg/lit reg

Description: Globally enables, disables or returns the current status of interrupts depending on the value of *src1*. Returns the previous interrupt enable state (1 for enabled or 0 for disabled) in *dst*. When the state of the global interrupt enable is changed, the processor ensures that the new state is in full effect before the instruction completes. (This instruction is implemented by manipulating ICON.gie.)

src1 **Value**	**Operation**
0	Disables interrupts
1	Enables interrupts
2	Returns current interrupt enable status

Action:

```
if (PC.em != supervisor)
    generate_fault(TYPE.MISMATCH);
old_interrupt_enable = global_interrupt_enable;
switch(src1) {
    case 0: # Disable. Set ICON.gie to one.
            globally_disable_interrupts;
            global_interrupt_enable = false;
            order_wrt(subsequent_instructions);
            break;
    case 1: # Enable.  Clear ICON.gie to zero.
            globally_enable_interrupts;
            global_interrupt_enable = true;
            order_wrt(subsequent_instructions);
            break;
    case 2: # Return status. Return ICON.gie
            break;
    default:
            generate_fault(OPERATION.INVALID_OPERAND);
            break;
}
if(old_interrupt_enable)
    dst = 1;
else
    dst = 0;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
                # ICON.gie = 0, interrupts enabled
intctl 0, g4    # Disable interrupts (ICON.gie = 1)
                # g4 = 1
```

Opcode: **intctl** 658H REG

See Also: **intdis, inten**

Notes: This instruction is implemented on the 80960Rx, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.35 intdis

Mnemonic:	**intdis**	Global Interrupt Disable
Format:	**intdis**	

Description: Globally disables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by setting ICON.gie to one.

Action:

```
if (PC.em != supervisor)
    generate_fault(TYPE.MISMATCH);
# Implemented by setting ICON.gie to one.
globally_disable_interrupts;
interrupt_enable = false;
order_wrt(subsequent_instructions);
```

Faults:		
	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
                   # ICON.gie = 0, interrupts enabled
intdis             # Disable interrupts.
                   # ICON.gie = 1
```

Opcode:	**intdis**	5B4H	REG

See Also: **intctl, inten**

Notes: This instruction is implemented on the 80960Rx, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.36 inten

Mnemonic: **inten** global interrupt enable

Format: **inten**

Description: Globally enables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by clearing ICON.gie to zero.

Action:

```
if (PC.em != supervisor)
    generate_fault(TYPE.MISMATCH);
# Implemented by clearing ICON.gie to zero.
globally_enable_interrupts;
interrupt_enable = true;
order_wrt(subsequent_instructions);
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
               # ICON.gie = 1, interrupts disabled.
inten          # Enable interrupts.
               # ICON.gie = 0
```

Opcode:

inten	5B5H	REG

See Also: **intctl, intdis**

Notes: This instruction is implemented on the 80960Rx, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.37 LOAD

Mnemonic:	**ld**	Load
	ldob	Load Ordinal Byte
	ldos	Load Ordinal Short
	ldib	Load Integer Byte
	ldis	Load Integer Short
	ldl	Load Long
	ldt	Load Triple
	ldq	Load Quad

Format:	**ld***	*src,*	*dst*
		mem	reg

Description: Copies byte or byte string from memory into a register or group of successive registers.

The *src* operand specifies the address of first byte to be loaded. The full range of addressing modes may be used in specifying *src*. Refer to Chapter 2, DATA TYPES AND MEMORY ADDRESSING MODES for more information.

dst specifies a register or the first (lowest numbered) register of successive registers.

ldob and **ldib** load a byte and **ldos** and **ldis** load a half word and convert it to a full 32-bit word. Data being loaded is sign-extended during integer loads and zero-extended during ordinal loads.

ld, **ldl**, **ldt** and **ldq** instructions copy 4, 8, 12 and 16 bytes, respectively, from memory into successive registers.

For **ldl**, *dst* must specify an even numbered register (i.e., g0, g2...). For **ldt** and **ldq**, *dst* must specify a register number that is a multiple of four (i.e., g0, g4, g8, g12, r4, r8, r12). Results are unpredictable if registers are not aligned on the required boundary or if data extends beyond register g15 or r15 for **ldl**, **ldt** or **ldq**.

Action:

ld:

```
dst = read_memory(effective_address)[31:0];
if((effective_address[1:0] != $00_2$ ) && unaligned _fault_enabled)
    generate_fault(OPERATION.UNALIGNED);
```

ldob:

```
dst[7:0] = read_memory(effective_address)[7:0];
dst[31:8] = 0x000000;
```

ldib:

```
dst[7:0] = read_memory(effective_address)[7:0];
if(dst[7] == 0)
    dst[31:8] = 0x000000;
else
    dst[31:8] = 0xFFFFFF;
```

ldos:

```
dst = read_memory(effective_address)[15:0];
                                    # Order depends on endianism.
dst[31:16] = 0x0000;
if((effective_address[0] != $0_2$) && unaligned_fault_enabled)
    generate_fault(OPERATION.UNALIGNED);
```

ldis:

```
dst[15:0] = read_memory(effective_address)[15:0];
                                    # Order depends on endianism.
if(dst[15] == $0_2$)
    dst[31:16] = 0x0000;
else
    dst[31:16] = 0xFFFF;
if((effective_address[0] != $0_2$) && unaligned_fault_enabled)
    generate_fault(OPERATION.UNALIGNED);
```

ldl:

```
if((reg_number(dst) % 2) != 0)
    generate_fault(OPERATION.INVALID_OPERAND);
    # dst not modified.
else
{   dst = read_memory(effective_address)[31:0];
    dst_+_1 = read_memory(effective_address_+_4)[31:0];
    if((effective_address[2:0] != $000_2$) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
}
```

ldt:

```
if((reg_number(dst) % 4) != 0)
    generate_fault(OPERATION.INVALID_OPERAND);
    # dst not modified.
else
{   dst = read_memory(effective_adddress)[31:0];
    dst_+_1 = read_memory(effective_adddress_+_4)[31:0];
    dst_+_2 = read_memory(effective_adddress_+_8)[31:0];
    if((effective_address[3:0] != 0000_2) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
}
```

ldq:

```
if((reg_number(dst) % 4) != 0)
    generate_fault(OPERATION.INVALID_OPERAND);
    # dst not modified.
else
{   dst = read_memory(effective_adddress)[31:0];
                            # Order depends on endianism.
    dst_+_1 = read_memory(effective_adddress_+_4)[31:0];
    dst_+_2 = read_memory(effective_adddress_+_8)[31:0];
    dst_+_3 = read_memory(effective_adddress_+_12)[31:0];
    if((effective_address[3:0] != 0000_2) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).
OPERATION.UNALIGNED
OPERATION.INVALID_OPERAND

Example:

```
ldl 2450 (r3), r10  # r10, r11 = r3 + 2450 in
                    # memory
```

Opcode:

ld	90H	MEM
ldob	80H	MEM
ldos	88H	MEM
ldib	C0H	MEM
ldis	C8H	MEM
ldl	98H	MEM
ldt	A0H	MEM
ldq	B0H	MEM

See Also: **MOVE, STORE**

6.2.38 lda

Mnemonic:	**lda**	Load Address	
Format:	**lda**	*src*,	*dst*
		mem	reg
		efa	

Description: Computes the effective address specified with *src* and stores it in *dst*. The *src* address is not checked for validity. Any addressing mode may be used to calculate *efa*.

An important application of this instruction is to load a constant longer than 5 bits into a register. (To load a register with a constant of 5 bits or less, **mov** can be used with a literal as the *src* operand.)

Action: dst = effective_address;

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
lda 58 (g9), g1     # g1 = g9+58
lda 0x749, r8       # r8 = 0x749
```

Opcode:	**lda**	8CH	MEM

6.2.39 mark

Mnemonic:	**mark**	Mark
Format:	**mark**	

Description: Generates mark trace fault if mark trace mode is enabled. Mark trace mode is enabled if the PC register trace enable bit (bit 0) and the TC register mark trace mode bit (bit 7) are set.

If mark trace mode is not enabled, **mark** behaves like a no-op.

For more information on trace fault generation, refer to CHAPTER 10, TRACING AND DEBUGGING.

Action:

```
if(PC.te && TC.mk)
    generate_fault(TRACE.MARK)
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
TRACE.MARK	Trace fault is generated if PC.te=1 and TC.mk=1.

Example:

```
# Assume that the mark trace mode is enabled.
ld xyz, r4
addi r4, r5, r6
mark
# Mark trace event is generated at this point in the
# instruction stream.
```

Opcode:	**mark** 66BH	REG

See Also: **fmark, modpc, modtc**

6.2.40 modac

Mnemonic:	**modac**	Modify AC		
Format:	**modac**	*mask,* reg/lit	*src,* reg/lit	*dst* reg

Description: Reads and modifies the AC register. *src* contains the value to be placed in the AC register; *mask* specifies bits that may be changed. Only bits set in *mask* are modified. Once the AC register is changed, its initial state is copied into *dst*.

Action:

```
temp = AC;
AC = (src & mask) | (AC & ~mask);
dst = temp;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
modac g1, g9, g12   # AC = g9, masked by g1.
                    # g12 = initial value of AC.
```

Opcode:	**modac**	645H	REG

See Also: **modpc, modtc**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.41 modi

Mnemonic:	**modi**	Modulo Integer		
Format:	**modi**	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Divides *src2* by *src1*, where both are integers and stores the modulo remainder of the result in *dst*. If the result is nonzero, *dst* has the same sign as *src1*.

Action:

```
if(src1 == 0)
    {    dst = undefined_value;
         generate_fault(ARITHMETIC.ZERO_DIVIDE);
    }
dst = src2 - (src2/src1) * src1;
if((src2 *src1 < 0 ) && (dst != 0))
    dst = dst + src1;
```

Faults:

STANDARD	See section 6.1.6, Faults (pg. 6-5).
ARITHMETIC.ZERO_DIVIDE	The *src1* operand is zero.

Example:

```
modi r9, r2, r5     # r5 = modulo (r2/r9)
```

Opcode: **modi** 749H REG

See Also: **divi, divo, remi, remo**

Notes: **modi** generates the correct result (0) when computing -2^{31} mod -1, although the corresponding 32-bit division does overflow, it does not generate a fault.

6.2.42 modify

Mnemonic:	**modify**	Modify		
Format:	**modify**	*mask*, reg/lit	*src*, reg/lit	*src/dst* reg

Description: Modifies selected bits in *src/dst* with bits from *src*. The *mask* operand selects the bits to be modified: only bits set in the *mask* operand are modified in *src/dst*.

Action: src_dst = (src & mask) | (src_dst & ~mask);

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
modify g8, g10, r4  # r4 = g10 masked by g8.
```

Opcode:	**modify**	650H	REG

See Also: **alterbit, extract**

6.2.43 modpc

Mnemonic:	**modpc**	Modify Process Controls	
Format:	**modpc**	*src*, reg/lit	*mask*, reg/lit	*src/dst* reg

Description:

Reads and modifies the PC register as specified with mask and *src/dst*. *src/dst* operand contains the value to be placed in the PC register; *mask* operand specifies bits that may be changed. Only bits set in the *mask* are modified. Once the PC register is changed, its initial value is copied into *src/dst*. The *src* operand is a dummy operand that should specify a literal or the same register as the mask operand.

The processor must be in supervisor mode to use this instruction with a non-zero mask value. If mask=0, this instruction can be used to read the process controls, without the processor being in supervisor mode.

If the action of this instruction lowers the processor priority, the processor checks the interrupt table for pending interrupts.

When process controls are changed, the processor recognizes the changes immediately except in one situation: if **modpc** is used to change the trace enable bit, the processor may not recognize the change before the next four non-branch instructions are executed. For more information see 3.6.3, Process Controls Register – PC (pg. 3-19).

Action:

```
if(mask != 0)
{    if(PC.em != supervisor)
          generate_fault(TYPE.MISMATCH);
     temp = PC;
     PC = (mask & src_dst) | (PC & ~mask);
     src_dst = temp;
     if(temp.priority > PC.priority)
          check_pending_interrupts;
}
else
     src_dst = PC;
```

Faults: STANDARD — Refer to section 6.1.6, Faults (pg. 6-5).
TYPE.MISMATCH

Example:

```
modpc g9, g9, g8    # process controls = g8
                    # masked by g9.
```

Opcode: **modpc** 655H REG

See Also: **modac, modtc**

Notes: Since **modpc** does not switch stacks, it should not be used to switch the mode of execution from supervisor to user (the supervisor stack can get corrupted in this case). The call and return mechanism should be used instead.

6.2.44 modtc

Mnemonic:	**modtc**	Modify Trace Controls		
Format:	**modtc**	*mask*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Reads and modifies TC register as specified with *mask* and *src2*. The *src2* operand contains the value to be placed in the TC register; *mask* operand specifies bits that may be changed. Only bits set in *mask* are modified. *mask* must not enable modification of reserved bits. Once the TC register is changed, its initial state is copied into *dst*.

The changed trace controls may take effect immediately or may be delayed. If delayed, the changed trace controls may not take effect until after the first non-branching instruction is fetched from memory or after four non-branching instructions are executed.

For more information on the trace controls, refer to CHAPTER 9, FAULTS and CHAPTER 10, TRACING AND DEBUGGING.

Action:

```
mode_bits = 0x000000FE;
event_flags = 0X0F000000
temp = TC;
tempa = (event_flags & TC & mask) | (mode_bits & mask);
TC = (tempa & src2) | (TC & ~tempa);
dst = temp;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
modtc g12, g10, g2  # trace controls = g10 masked
                    # by g12; previous trace
                    # controls stored in g2.
```

Opcode:	**modtc**	654H	REG

See Also: **modac, modpc**

6.2.45 MOVE

Mnemonic:	**mov**	Move	
	movl	Move Long	
	movt	Move Triple	
	movq	Move Quad	
Format:	**mov***	*src1,*	*dst*
		reg/lit	reg

Description: Copies the contents of one or more source registers (specified with *src*) to one or more destination registers (specified with *dst*).

For **movl**, **movt** and **movq**, *src1* and *dst* specify the first (lowest numbered) register of several successive registers. *src1* and *dst* registers must be even numbered (e.g., g0, g2, ... or r4, r6, ...) for **movl** and an integral multiple of four (e.g., g0, g4, ... or r4, r8, ...) for **movt** and **movq**.

The moved register values are unpredictable when: 1) the *src* and *dst* operands overlap; 2) registers are not properly aligned.

Action:

mov:

```
if(is_reg(src1))
    dst = src1;
else
{   dst[4:0] = src1;    #src1 is a 5-bit literal.
    dst[31:5] = 0;
}
```

movl:

```
if((reg_num(src1)%2 != 0) || (reg_num(dst)%2 != 0))
{   dst = undefined_value;
    dst_+_1 = undefined_value;
    generate_fault(OPERATION.INVALID_OPERAND);
}
else if(is_reg(src1))
{   dst = src1;
    dst_+_1 = src1_+_1;
}
else
{   dst[4:0] = src1;    #src1 is a 5-bit literal.
    dst[31:5] = 0;
    dst_+_1[31:0] = 0;
}
```

movt:

```
if((reg_num(src1)%4 != 0) || (reg_num(dst)%4 != 0))
{     dst = undefined_value;
      dst_+_1 = undefined_value;
      dst_+_2 = undefined_value;
      generate_fault(OPERATION.INVALID_OPERAND);
}
else if(is_reg(src1))
{     dst = src1;
      dst_+_1 = src1_+_1;
      dst_+_2 = src1_+_2;
}
else
{     dst[4:0] = src1;    #src1 is a 5-bit literal.
      dst[31:5] = 0;
      dst_+_1[31:0] = 0;
      dst_+_2[31:0] = 0;
}
```

movq:

```
if((reg_num(src1)%4 != 0) || (reg_num(dst)%4 != 0))
{     dst = undefined_value;
      dst_+_1 = undefined_value;
      dst_+_2 = undefined_value;
      dst_+_3 = undefined_value;
      generate_fault(OPERATION.INVALID_OPERAND);
}
else if(is_reg(src1))
{     dst = src1;
      dst_+_1 = src1_+_1;
      dst_+_2 = src1_+_2;
      dst_+_3 = src1_+_3;
}
else
{     dst[4:0] = src1;    #src1 is a 5 bit literal.
      dst[31:5] = 0;
      dst_+_1[31:0] = 0;
      dst_+_2[31:0] = 0;
      dst_+_3[31:0] = 0;
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
movt g8, r4          # r4, r5, r6 = g8, g9, g10
```

Opcode:	**mov**	5CCH	REG
	movl	5DCH	REG
	movt	5ECH	REG
	movq	5FCH	REG

See Also: **LOAD, STORE, lda**

6.2.46 muli, mulo

Mnemonic:	**muli**	Multiply Integer		
	mulo	Multiply Ordinal		
Format:	**mul***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Multiplies the *src2* value by the *src1* value and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that muli can signal an integer overflow.

Action:

mulo:

```
dst = (src2 * src1)[31:0];
```

muli:

```
true_result = (src1 * src2);
dst = true_result[31:0];
if((true_result > (2**31) - 1) || (true_result < -2**31)) # Check for overflow
{    if(AC.om == 1)
          AC.of = 1;
     else
          generate_fault(ARITHMETIC.OVERFLOW);
}
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
ARITHMETIC.OVERFLOW	Result is too large for destination register (**muli** only). If a condition of overflow occurs, the least significant 32 bits of the result are stored in the destination register.

Example:

```
muli r3, r4, r9     # r9 = r4 * r3
```

Opcode:

muli	741H	REG
mulo	701H	REG

See Also: **emul, ediv, divi, divo**

6.2.47 nand

Mnemonic:	**nand**	Nand		
Format:	**nand**	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Performs a bitwise NAND operation on *src2* and *src1* values and stores the result in *dst*.

Action: dst = ~src2 | ~src1;

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
nand g5, r3, r7      # r7 = r3 NAND g5
```

Opcode:	**nand**	58EH	REG

See Also: **and, andnot, nor, not, notand, notor, or, ornot, xnor, xor**

6.2.48 nor

Mnemonic:	**nor**	Nor	
Format:	**nor**	*src1*, reg/lit	*src2*, reg/lit

Format (continued): *dst* reg

Description: Performs a bitwise NOR operation on the *src2* and *src1* values and stores the result in *dst*.

Action: dst = ~src2 & ~src1;

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
nor g8, 28, r5      # r5 = 28 NOR g8
```

Opcode: **nor** 588H REG

See Also: **and, andnot, nand, not, notand, notor, or, ornot, xnor, xor**

6.2.49 not, notand

Mnemonic:	**not**	Not	
	notand	Not And	
Format:	**not**	*src1*,	*dst*
		reg/lit	reg
	notand	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Performs a bitwise NOT (**not** instruction) or NOT AND (**notand** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

not:
dst = ~src1;

notand:
dst = ~src2 & src1;

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
not g2, g4            # g4 = NOT g2
notand r5, r6, r7     # r7 = NOT r6 AND r5
```

Opcode:	**not**	58AH	REG
	notand	584H	REG

See Also: **and, andnot, nand, nor, notor, or, ornot, xnor, xor**

6.2.50 notbit

Mnemonic: **notbit** Not Bit

Format:	**notbit**	*bitpos*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Copies the *src2* value to *dst* with one bit toggled. The *bitpos* operand specifies the bit to be toggled.

Action: dst = src2 ^ 2**(src1%32);

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
notbit r3, r12, r7  # r7 = r12 with the bit
                    # specified in r3 toggled.
```

Opcode: **notbit** 580H REG

See Also: **alterbit, chkbit, clrbit, setbit**

6.2.51 notor

Mnemonic: **notor** Not Or

Format: **notor** *src1,* *src2,* *dst*
reg/lit reg/lit reg

Description: Performs a bitwise NOTOR operation on *src2* and *src1* values and stores result in *dst*.

Action: dst = ~src2 | src1;

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
notor g12, g3, g6   # g6 = NOT g3 OR g12
```

Opcode: **notor** 58DH REG

See Also: **and, andnot, nand, nor, not, notand, or, ornot, xnor, xor**

6.2.52 or, ornot

Mnemonic:	**or**	Or		
	ornot	Or Not		
Format:	**or**	*src1*, reg/lit	*src2*, reg/lit	*dst* reg
	ornot	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Performs a bitwise OR (**or** instruction) or ORNOT (**ornot** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

or:
dst = src2 | src1;

ornot:
dst = src2 | ~src1;

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
or 14, g9, g3        # g3 = g9 OR 14
ornot r3, r8, r11    # r11 = r8 OR NOT r3
```

Opcode:	**or**	587H	REG
	ornot	58BH	REG

See Also: **and, andnot, nand, nor, not, notand, notor, xnor, xor**

6.2.53 remi, remo

Mnemonic:	**remi** **remo**	Remainder Integer Remainder Ordinal		
Format:	**rem***	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Divides *src2* by *src1* and stores the remainder in *dst*. The sign of the result (if nonzero) is the same as the sign of *src2*.

Action: **remi, remo:**

```
if(src1 == 0)
    generate_fault(ARITHMETIC.ZERO_DIVIDE);
dst = src2 - (src2/src1)*src1;
```

Faults:	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	ARITHMETIC.ZERO_DIVIDE	The *src1* operand is 0.

Example:

```
remo r4, r5, r6     # r6 = r5 rem r4
```

Opcode:	**remi**	748H	REG
	remo	708H	REG

See Also: **modi**

Notes: **remi** produces the correct result (0) even when computing -2^{31} **remi** -1, which would cause the corresponding division to overflow, although no fault is generated.

6.2.54 ret

Mnemonic: **ret** Return

Format: **ret**

Description: Returns program control to the calling procedure. The current stack frame (i.e., that of the called procedure) is deallocated and the FP is changed to point to the calling procedure's stack frame. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the return-status field and prereturn-trace flag determine the action that the processor takes on the return. These fields are contained in bits 0 through 3 of register r0 of the called procedure's local registers.

See CHAPTER 7, PROCEDURE CALLS for more on **ret**.

Action:

```
implicit_syncf();
if(pfp.p && PC.te && TC.p)
{    pfp.p = 0;
     generate_fault(TRACE.PRERETURN);
}
switch(return_status_field)
{
     case $000_2$:          #local return
          get_FP_and_IP();
          break;
     case $001_2$:          #fault return
          tempa = memory(FP-16);
          tempb = memory(FP-12);
          get_FP_and_IP();
          AC = tempb;
          if(execution_mode == supervisor)
               PC = tempa;
          break;
     case $010_2$:          #supervisor return, trace on return disabled
          if(execution_mode != supervisor)
               get_FP_and_IP();
          else
          {    PC.te = 0;
               execution_mode = user;
               get_FP_and_IP();
          }
          break;
case 011₂:          # supervisor return, trace on return enabled
          if(execution_mode != supervisor)
               get_FP_and_IP();
          else
          {    PC.te = 1;
               execution_mode = user;
               get_FP_and_IP();
          }
          break;
case 100₂:          #reserved -  unpredictable behavior
     break;
case 101₂:          #reserved -  unpredictable behavior
     break;
case 110₂:          #reserved -  unpredictable behavior
     break;
case 111₂:          #interrupt return
     tempa = memory(FP-16);
     tempb = memory(FP-12);
     get_FP_and_IP();
     AC = tempb;
     if(execution_mode == supervisor)
          PC = tempa;
          check_pending_interrupts();
          break;
}

get_FP_and_IP()
{    FP =PFP;
     free(current_register_set);
     if(not_allocated(FP))
          retrieve_from_memory(FP);
     IP = RIP;
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
ret                  # Program control returns to
                     # context of calling procedure.
```

Opcode: **ret** 0AH CTRL

See Also: **call, calls, callx**

6.2.55 rotate

Mnemonic:	**rotate**	Rotate	
Format:	**rotate**	*len*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Copies *src2* to *dst* and rotates the bits in the resulting *dst* operand to the left (toward higher significance). Bits shifted off left end of word are inserted at right end of word. The *len* operand specifies number of bits that the *dst* operand is rotated.

This instruction can also be used to rotate bits to the right. The number of bits the word is to be rotated right should be subtracted from 32 and the result used as the *len* operand.

Action: *src2* is rotated by *len* mod 32. This value is stored in *dst*.

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
rotate 13, r8, r12  # r12 = r8 with bits rotated
                    # 13 bits to left.
```

Opcode: **rotate** 59DH REG

See Also: **SHIFT, eshro**

6.2.56 scanbit

Mnemonic:	**scanbit**	Scan For Bit	
Format:	**scanbit**	*src1,* reg/lit	*dst* reg

Description: Searches *src1* for a set bit (1 bit). If a set bit is found, the bit number of the most significant set bit is stored in the *dst* and the condition code is set to 010_2. If *src* value is zero, all 1's are stored in *dst* and condition code is set to 000_2.

Action:

```
dst = 0xFFFFFFFF;
AC.cc = 000_2;
for(i = 31; i >= 0; i--)
{    if((src1 & 2**i) != 0)
{         dst = i;
          AC.cc = 010_2;
          break;
}
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# assume g8 is nonzero
scanbit g8, g10     # g10 = bit number of most-
                    # significant set bit in g8;
                    # AC.cc = 010₂.
```

Opcode:	**scanbit**	641H	REG

See Also: **spanbit, setbit**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.57 scanbyte

Mnemonic: **scanbyte** Scan Byte Equal

Format: **scanbyte** *src1*, *src2*
reg/lit reg/lit

Description: Performs byte-by-byte comparison of *src1* and *src2* and sets condition code to 010_2 if any two corresponding bytes are equal. If no corresponding bytes are equal, condition code is set to 000_2.

Action:

```
if((src1 & 0x000000FF) == (src2 & 0x000000FF)
    || (src1 & 0x0000FF00) == (src2 & 0x0000FF00)
    || (src1 & 0x00FF0000) == (src2 & 0x00FF0000)
    || (src1 & 0xFF000000) == (src2 & 0xFF000000))
        AC.cc = $010_2$;
else
    AC.cc = $000_2$;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Assume r9 = 0x11AB1100
scanbyte 0x00AB0011, r9# AC.cc = 010₂
```

Opcode: **scanbyte** 5ACH REG

See Also: **bswap**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.58 SEL<cc>

Mnemonic:		
	selno	Select Based on Unordered
	selg	Select Based on Greater
	sele	Select Based on Equal
	selge	Select Based on Greater or Equal
	sell	Select Based on Less
	selne	Select Based on Not Equal
	selle	Select Based on Less or Equal
	selo	Select Based on Ordered

Format:				
	sel*	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Selects either *src1* or *src2* to be stored in *dst* based on the condition code bits in the arithmetic controls. If for Unordered the condition code is 0, or if for the other cases the logical AND of the condition code and the mask part of the opcode is not zero, then the value of *src2* is stored in the destination. Else, the value of *src1* is stored in the destination.

Instruction	Mask	Condition
selno	000_2	Unordered
selg	001_2	Greater
sele	010_2	Equal
selge	011_2	Greater or equal
sell	100_2	Less
selne	101_2	Not equal
selle	110_2	Less or equal
selo	111_2	Ordered

Action:

```
if ((mask & AC.cc) || (mask == AC.cc))
    dst = src2;
else
    dst = src1;
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
                    # AC.cc = 010₂
sele g0,g1,g2       # g2 = g1

                    # AC.cc = 001₂
sell g0,g1,g2       # g2 = g0
```

Opcode:	**selno**	784H	REG
	selg	794H	REG
	sele	7A4H	REG
	selge	7B4H	REG
	sell	7C4H	REG
	selne	7D4H	REG
	selle	7E4H	REG
	selo	7F4H	REG

See Also: **MOVE, TEST<cc>, cmpi, cmpo, SUB<cc>**

Notes: These core instructions are not implemented on i960 Cx, Kx and Sx processors.

6.2.59 setbit

Mnemonic:	**setbit**	Set Bit		
Format:	**setbit**	*bitpos*, reg/lit	*src*, reg/lit	*dst* reg

Description: Copies *src* value to *dst* with one bit set. *bitpos* specifies bit to be set.

Action: dst = src | (2**(bitpos%32));

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
setbit 15, r9, r1   # r1 = r9 with bit 15 set.
```

Opcode:	**setbit**	583H	REG

See Also: **alterbit, chkbit, clrbit, notbit**

6.2.60 SHIFT

Mnemonic:	**shlo**	Shift Left Ordinal
	shro	Shift Right Ordinal
	shli	Shift Left Integer
	shri	Shift Right Integer
	shrdi	Shift Right Dividing Integer

Format:	**sh***	*len,*	*src,*	*dst*
		reg/lit	reg/lit	reg

Description:

Shifts *src* left or right by the number of bits indicated with the *len* operand and stores the result in *dst*. Bits shifted beyond register boundary are discarded. For values of *len* > 32, the processor interprets the value as 32.

shlo shifts zeros in from the least significant bit; **shro** shifts zeros in from the most significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

shli shifts zeros in from the least significant bit. An overflow fault is generated if the bits shifted out are not the same as the most significant bit (bit 31). If overflow occurs, *dst* will equal *src* shifted left as much as possible without overflowing.

shri performs a conventional arithmetic shift-right operation by shifting in the most significant bit (bit 31). When this instruction is used to divide a negative integer operand by the power of 2, it produces an incorrect quotient (discarding the bits shifted out has the effect of rounding the result toward negative).

shrdi is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the *src* operand was negative, which produces the correct result for negative operands.

shli and **shrdi** are equivalent to **muli** and **divi** by the power of 2.

Action:

```
shlo:
if(src1 < 32)
    dst = src * (2**len);
else
    dst = 0;
shro:
if(src1 < 32)
    dst = src / (2**len);
else
    dst = 0;
```

shli:

```
if(len > 32)
    count = 32;
else
    count = src1;
temp = src;
while((temp[31] == temp[30]) && (count > 0))
{   temp = (temp * 2)[31:0];
    count = count - 1;
}
dst = temp;
if(count > 0)
{   if(AC.om == 1)
        AC.of = 1;
    else
        generate_fault(ARITHMETIC.OVERFLOW);
}
```

shri:

```
if(len > 32)
    count = 32;
else
    count = src1;
temp = src;
while(count > 0)
{   temp = (temp >> 1)[31:0];
    temp[31] = src[31];
    count = count - 1;
}
dst = temp;
```

shrdi:

```
dst = src / (2**len);
```

Faults:	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	ARITHMETIC.OVERFLOW	For **shli.**

Example:

```
shli 13, g4, r6     # g6 = g4 shifted left 13 bits.
```

Opcode:

shlo	59CH	REG
shro	598H	REG
shli	59EH	REG
shri	59BH	REG
shrdi	59AH	REG

See Also: **divi, muli, rotate, eshro**

Notes: **shli** and **shrdi** are identical to multiplications and divisions for all positive and negative values of *src2*. **shri** is the conventional arithmetic right shift that does not produce a correct quotient when *src2* is negative.

6.2.61 spanbit

Mnemonic:	**spanbit**	Span Over Bit	
Format:	**spanbit**	*src*, reg/lit	*dst* reg

Description: Searches *src* value for the most significant clear bit (0 bit). If a most significant 0 bit is found, its bit number is stored in *dst* and condition code is set to 010_2. If *src* value is all 1's, all 1's are stored in *dst* and condition code is set to 000_2.

Action:

```
dst = 0xFFFFFFFF;
AC.cc = 000₂;
for(i = 31; i > = 0; i--)
{     if((src1 & 2**i) == 0))
{          dst = i;
           AC.cc = 010₂;
           break;
      }
}
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
# Assume r2 is not 0xffffffff
spanbit r2, r9      # r9 = bit number of most-
                    # significant clear bit in r2;
                    # AC.cc = 010₂
```

Opcode:	**spanbit**	640H	REG

See Also: **scanbit**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.62 STORE

Mnemonic:	**st**	Store
	stob	Store Ordinal Byte
	stos	Store Ordinal Short
	stib	Store Integer Byte
	stis	Store Integer Short
	stl	Store Long
	stt	Store Triple
	stq	Store Quad

Format:	**st***	*src1*,	*dst*
		reg	mem

Description:

Copies a byte or group of bytes from a register or group of registers to memory. *src* specifies a register or the first (lowest numbered) register of successive registers.

dst specifies the address of the memory location where the byte or first byte or a group of bytes is to be stored. The full range of addressing modes may be used in specifying *dst*. Refer to 2.3, MEMORY ADDRESSING MODES (pg. 2-5) for a complete discussion.

stob and **stib** store a byte and **stos** and **stis** store a half word from the *src* register's low order bytes. Data for ordinal stores is truncated to fit the destination width. If the data for integer stores cannot be represented correctly in the destination width, an Arithmetic Integer Overflow fault is signaled.

st, **stl**, **stt** and **stq** copy 4, 8, 12 and 16 bytes, respectively, from successive registers to memory.

For **stl**, *src* must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For **stt** and **stq**, *src* must specify a register number that is a multiple of four (e.g., g0, g4, g8, ... or r0, r4, r8, ...).

Action:

st:

```
if (illegal_write_to_on_chip_RAM)
    generate_fault(TYPE.MISMATCH);
else if ((effective_address[1:0] != $00_2$) && unaligned_fault_enabled)
    {store_to_memory(effective_address)[31:0]  = src1;
    generate_fault(OPERATION.UNALIGNED);}
else
    store_to_memory(effective_address)[31:0] = src1;
```

Action:

stob:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
```

```
else
    store_to_memory(effective_address)[7:0] = src1[7:0];
```

stib:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if ((src1[31:8] != 0) && (src1[31:8] != 0xFFFFFF))
    {   store_to_memory(effective_address)[7:0] = src1[7:0];
        if (AC.om == 1)
            AC.of = 1;
        else
            generate_fault(ARITHMETIC.OVERFLOW);
    }
else
    store_to_memory(effective_address)[7:0] = src1[7:0];
end if;
```

stos:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if ((effective_address[0] != $0_2$) && unaligned_fault_enabled)
    {   store_to_memory(effective_address)[15:0] = src1[15:0];
        generate_fault(OPERATION.UNALIGNED);
    }
else
    store_to_memory(effective_address)[15:0] = src1[15:0];
```

stis:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if ((effective_address[0] != $0_2$) && unaligned_fault_enabled)
    {   store_to_memory(effective_address)[15:0] = src1[15:0];
        generate_fault(OPERATION.UNALIGNED);
    }
else if ((src1[31:16] != 0) && (src1[31:16] != 0xFFFF))
    {   store_to_memory(effective_address)[15:0] = src1[15:0];
        if (AC.om == 1)
        AC.of = 1;
    else
        generate_fault(ARITHMETIC.OVERFLOW);
    }
else
    store_to_memory(effective_address)[15:0] = src1[15:0];
```

stl:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if (reg_number(src1) % 2 != 0)
    generate_fault(OPERATION.INVALID_OPERAND);
else if ((effective_address[2:0] != $000_2$) && unaligned_fault_enabled)
    {   store_to_memory(effective_address)[31:0] = src1;
        store_to_memory(effective_address + 4)[31:0] = src1_+_1;
        generate_fault (OPERATION.UNALIGNED);
    }
else
    {   store_to_memory(effective_address)[31:0] = src1;
        store_to_memory(effective_address + 4)[31:0] = src1_+_1;
    }
```

stt:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if (reg_number(src1) % 4 != 0)
    generate_fault(OPERATION.INVALID_OPERAND);
else if ((effective_address[3:0] != $0000_2$) && unaligned_fault_enabled)
    {   store_to_memory(effective_address)[31:0] = src1;
        store_to_memory(effective_address + 4)[31:0] = src1_+_1;
        store_to_memory(effective_address + 8)[31:0] = src1_+_2;
        generate_fault (OPERATION.UNALIGNED);
    }
else
    {   store_to_memory(effective_address)[31:0] = src1;
        store_to_memory(effective_address + 4)[31:0] = src1_+_1;
        store_to_memory(effective_address + 8)[31:0] = src1_+_2;
    }
```

stq:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if (reg_number(src1) % 4 != 0)
    generate_fault(OPERATION.INVALID_OPERAND);
else if ((effective_address[3:0] != $0000_2$) && unaligned_fault_enabled)
    {   store_to_memory(effective_address)[31:0] = src1;
        store_to_memory(effective_address + 4)[31:0] = src1_+_1;
        store_to_memory(effective_address + 8)[31:0] = src1_+_2;
        store_to_memory(effective_address + 12)[31:0] = src1_+_3;
        generate_fault (OPERATION.UNALIGNED);
    }
```

```
else
    {   store_to_memory(effective_address)[31:0] = src1;
        store_to_memory(effective_address + 4)[31:0] = src1_+_1;
        store_to_memory(effective_address + 8)[31:0] = src1_+_2;
        store_to_memory(effective_address + 12)[31:0] = src1_+_3;
    }
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
ARITHMETIC.OVERFLOW	For **stib**, **stis**.

Example:

```
st g2, 1254 (g6)    # Word beginning at offset
                    # 1254 + (g6) = g2.
```

Opcode:

st	92H	MEM
stob	82H	MEM
stos	8AH	MEM
stib	C2H	MEM
stis	CAH	MEM
stl	9AH	MEM
stt	A2H	MEM
stq	B2H	MEM

See Also: **LOAD, MOVE**

Notes: illegal_write_to_on_chip_RAM is an implementation-dependent mechanism. The mapping of register bits to memory(*efa*) depends on the endianism of the memory region and is implementation-dependent.

6.2.63 subc

Mnemonic:	**subc**	Subtract Ordinal With Carry		
Format:	**subc**	*src1*, reg/lit	*src2*, reg/lit	*dst* reg

Description: Subtracts *src1* from *src2*, then subtracts the opposite of condition code bit 1 (used here as the carry bit) and stores the result in *dst*. If the ordinal subtraction results in a carry, condition code bit 1 is set to 1, otherwise it is set to 0.

This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, condition code bit 0 is set.

subc does not distinguish between ordinals and integers: it sets condition code bits 0 and 1 regardless of data type.

Action:

```
dst = (src2 - src1 -1 + AC.cc[1])[31:0];
AC.cc[2:0] = 000$_2$;
if((src2[31] == src1[31]) && (src2[31] != dst[31]))
    AC.cc[0] = 1;           # Overflow bit.
AC.cc[1] = (src2 - src1 -1 + AC.cc[1])[32];      # Carry out.
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
subc g5, g6, g7
# g7 = g6 - g5 - not(condition code bit 1)
```

Opcode: **subc** 5B2H REG

See Also: **addc, addi, addo, subi, subo**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.64 SUB<cc>

Mnemonic:	**subono**	Subtract Ordinal if Unordered
	subog	Subtract Ordinal if Greater
	suboe	Subtract Ordinal if Equal
	suboge	Subtract Ordinal if Greater or Equal
	subol	Subtract Ordinal if Less
	subone	Subtract Ordinal if Not Equal
	subole	Subtract Ordinal if Less or Equal
	suboo	Subtract Ordinal if Ordered
	subino	Subtract Integer if Unordered
	subig	Subtract Integer if Greater
	subie	Subtract Integer if Equal
	subige	Subtract Integer if Greater or Equal
	subil	Subtract Integer if Less
	subine	Subtract Integer if Not Equal
	subile	Subtract Integer if Less or Equal
	subio	Subtract Integer if Ordered

Format:	**sub***	*src1,*	*src2,*	*dst*
		reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2* conditionally based on the condition code bits in the arithmetic controls.

If for Unordered the condition code is 0, or if for the other cases the logical AND of the condition code and the mask part of the opcode is not zero; then *src1* is subtracted from *src2* and the result stored in the destination.

Instruction	Mask	Condition
subono, subino	000_2	Unordered
subog, subig	001_2	Greater
suboe, subie	010_2	Equal
suboge, subige	011_2	Greater or equal
subol, subil	100_2	Less
subone, subine	101_2	Not equal
subole, subile	110_2	Less or equal
suboo, subio	111_2	Ordered

Action:

SUBO<cc>:

```
if ((mask & AC.cc) || (mask == AC.cc))
    dst = (src2 - src1)[31:0];
```

SUBI<cc>:

```
if ((mask & AC.cc) || (mask == AC.cc))
{
    {       true_result = (src2 - src1);
            dst = true_result[31:0];
    }
    if((true_result > (2**31) - 1) || (true_result < -2**31))
                                                   # Check for overflow
        {   if (AC.om == 1)
                AC.of  = 1;
            else
                generate_fault (ARITHMETIC.OVERFLOW);
        }
}
```

Faults:

STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
ARITHMETIC.OVERFLOW	For the **SUBI<cc>** class.

Example:

```
                      # AC.cc = $010_2$
suboge g0,g1,g2       # g2 = g1 - g0

                      # AC.cc = $001_2$
subile g0,g1,g2       # g2 not modified
```

Opcode:

subono	782H	REG
subog	792H	REG
suboe	7A2H	REG
suboge	7B2H	REG
subol	7C2H	REG
subone	7D2H	REG
subole	7E2H	REG
suboo	7F2H	REG
subino	783H	REG
subig	793H	REG
subie	7A3H	REG
subige	7B3H	REG
subil	7C3H	REG
subine	7D3H	REG
subile	7E3H	REG
subio	7F3H	REG

See Also: **subc, subi, subo, SEL<cc>, TEST<cc>**

Notes: These core instructions are not implemented on 80960Cx, Kx and Sx processors.

6.2.65 subi, subo

Mnemonic:	**subi**	Subtract Integer		
	subo	Subtract Ordinal		
Format:	**sub***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2* and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that **subi** can signal an integer overflow.

Action:

subo:

```
dst = (src2 - src1)[31:0];
```

subi:

```
true_result = (src2 - src1);
dst = true_result[31:0];
if((true_result > (2**31) - 1) || (true_result < -2**31)) # Check for overflow
{     if(AC.om == 1)
              AC.of = 1;
      else
              generate_fault(ARITHMETIC.OVERFLOW);
}
```

Faults:	STANDARD	Refer to section 6.1.6, Faults (pg. 6-5).
	ARITHMETIC.OVERFLOW	For **subi.**

Example:

```
subi g6, g9, g12    # g12 = g9 - g6
```

Opcode:	**subi**	593H	REG
	subo	592H	REG

See Also: **addi, addo, subc, addc**

6.2.66 syncf

Mnemonic: **syncf** Synchronize Faults

Format: **syncf**

Description: Waits for all faults to be generated that are associated with any prior uncompleted instructions.

Action:

```
if(AC.nif == 1)
    break;
else
    wait_until_all_previous_instructions_in_flow_have_completed();
    #   This also means that all of the faults on these instructions have
    #   been reported.
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
ld xyz, g6
addi r6, r8, r8
syncf
and g6, 0xFFFF, g8
# The syncf instruction ensures that any faults
# that may occur during the execution of the
# ld and addi instructions occur before the
# and instruction is executed.
```

Opcode: **syncf** 66FH REG

See Also: **mark, fmark**

6.2.67 sysctl

Mnemonic:	**sysctl**	System Control		
Format:	**sysctl**	*src1*, reg/lit	*src2*, reg/lit	*src/dst* reg

Description: Performs system management and control operations including requesting software interrupts, invalidating the instruction cache, configuring the instruction cache, processor reinitialization, modifying memory-mapped registers, and acquiring breakpoint resource information.

Processor control function specified by the message field of *src1* is executed. The type field of *src1* is interpreted depending upon the command. Remaining *src1* bits are reserved. The *src2* and *src3* operands are also interpreted depending upon the command.

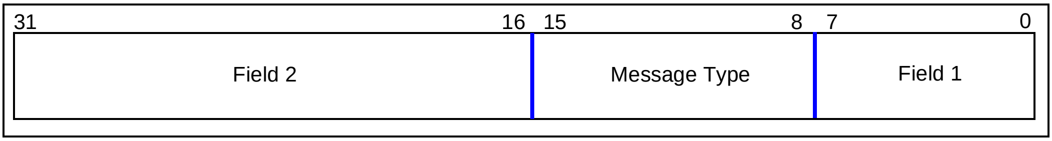


Figure 6-7. Src1 Operand Interpretation

Table 6-10. sysctl Field Definitions

Message	*src1*			*src2*	*src/dst*
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	0x0	Vector Number	N/U	N/U	N/U
Invalidate Cache	0x1	N/U	N/U	N/U	N/U
Configure Instruction Cache	0x2	Cache Mode Configuration (See Table 6-11)	N/U	Cache load address	N/U
Reinitialize	0x3	N/U	N/U	Starting IP	PRCB Pointer
Modify Memory-Mapped Control Register (MMR)	0x5	N/U	Lower 2 bytes of MMR address	Value to write	Mask
Breakpoint Resource Request	0x6	N/U	N/U	N/U	Breakpoint info (See Figure 6-8)

NOTE: Sources and fields that are not used (designated N/U) are ignored.

Table 6-11. Cache Mode Configuration

Mode Field	Mode Description	80960Rx
000_2	Normal cache enabled	4 Kbyte
$XX1_2$	Full cache disabled	4 Kbyte
100_2 or 110_2	Load and lock one way of the cache	2 Kbyte

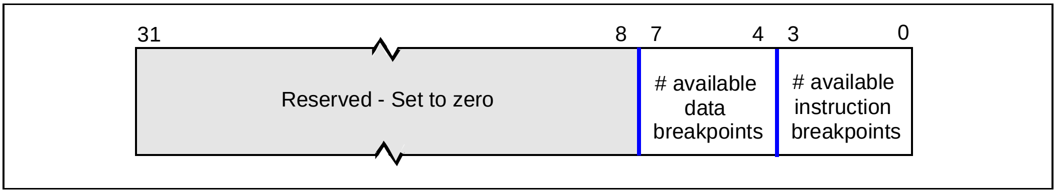

31
8
7
4
3
0
Reserved - Set to zero
# available data breakpoints
# available instruction breakpoints

Figure 6-8. *src/dst* Interpretation for Breakpoint Resource Request

Action:

```
if (PC.em != supervisor)
      generate_fault(TYPE.MISMATCH);
order_wrt(previous_operations);
OPtype = (src1 & 0xff00) >> 8;
switch (OPtype) {
  case 0:        # Signal Software Interrupt
      vector_to_post = 0xff & src1;
      priority_to_post = vector_to_post >> 3;
      pend_ints_addr = interrupt_table_base + 4 + priority_to_post;
      pend_priority = memory_read(interrupt_table_base,atomic_lock);
      # Priority zero just recans Interrupt Table
      if (priority_to_post != 0)
         {pend_ints = memory_read(pend_ints_addr, non-cacheable)
          pend_ints[7 & vector] = 1;
          pend_priority[priority_to_post] = 1;
          memory_write(pend_ints_addr, pend_ints); }
      memory_write(interrupt_table_base,pend_priority,atomic_unlock);
      # Update internal software priority with highest priority interrupt
      # from newly adjusted Pending Priorities word.  The current internal
      # software priority is always replaced by the new, computed one. (If
      # there is no bit set in pending_priorities word for the current
      # internal one, then it is discarded by this action.)
      if (pend_priority == 0)
            SW_Int_Priority = 0;
      else { msb_set = scan_bit(pend_priority);
```

```
        SW_Int_Priority = msb_set;   }

# Make sure change to internal software priority takes full effect
# before next instruction.
order_wrt(subsequent_operations);
                    break;
case 1:         # Global Invalidate Instruction Cache
                invalidate_instruction_cache( );
                unlock_instruction_cache( );
                break;
case 2:         # Configure Instruction-Cache
                mode = src1 & 0xff;
                if (mode & 1) disable_instruction_cache;
                else switch (mode) {
                    case 0:        enable_instruction_cache; break;
                    case 4,6:      # Load & Lock code into I-Cache
                        # All contiguous blocks are locked.
                        # Note:   block = way on 80960Rx.
                        # src2 has starting address of code to lock.
                        # src2 is aligned to a quad word
                        # boundary.
                        aligned_addr = src2 & 0xfffffff0;
                        invalidate(I-cache); unlock(I-cache);
                        for (j = 0; j < number_of_blocks_that_lock; j++)
                        {way = block_associated_with_block(j);
                         start = src2 + j*block_size;
                         end = start + block_size;
                         for (i = start; i < end; i=i+4)
                             {     set = set_associated_with(i);
                                   word = word_associated_with(i);
                                   Icache_line[set][way][word] =
                                                       memory[i];
                                   update_tag_n_valid_bits(set,way,word)
                                   lock_icache(set,way,word);
                             } } break;
                    default:
                             generate_operation_invalid_operand_fault;
                    } break;
case 3:         # Software Re-init
                disable(I_cache); invalidate(I_cache);
                disable(D_cache); invalidate(D_cache);
                Process_PRCB(dst);  # dst has ptr to new PRCB
                IP = src2;
                break;
```

```
          case 5:     # Modify One Memory-Mapped Control Register (MMR)
                      # src1[31:16] has lower 2 bytes of MMR address
                      # src2 has value to write; dst has mask.
                      # After operation, dst has old value of MMR
                      addr = (0xff00 << 16) | (src1 >> 16);
                      temp = memory[addr];
                      memory[addr] = (src2 & dst) | (temp & ~dst);
                      dst = temp;
                      break;
          case 6:     # Breakpoint Resource Request
                      acquire_available_instr_breakpoints( );
                      dst[3:0] = number_of_available_instr_breakpoints;
                      acquire_available_data_breakpoints( );
                      dst[7:4] = number_of_available_data_breakpoints;
                      dst[31:8] = 0;
                      break;
          default:    # Reserved, fault occurs
                      generate_fault(OPERATION.INVALID_OPERAND);
                      break;
}
order_wrt(subsequent_operations);
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
ldconst 0x100,r6           # Set up message.
sysctl r6,r7,r8            # Invalidate I-cache.
                           # r7, r8 are not used.
ldconst 0x204, g0          # Set up message type and
                           # cache configuration mode.
                           # Lock half cache.
ldconst 0x20000000,g2      # Starting address of code.
sysctl g0,g2,g2            # Execute Load and Lock.
```

Opcode: **sysctl** 659H REG

See Also: **dcctl, icctl**

Notes: This instruction is implemented on 80960Rx, Hx, Jx and Cx processors, and may or may not be implemented on future i960 processors.

6.2.68 TEST<cc>

Mnemonic:	**teste**	Test For Equal
	testne	Test For Not Equal
	testl	Test For Less
	testle	Test For Less Or Equal
	testg	Test For Greater
	testge	Test For Greater Or Equal
	testo	Test For Ordered
	testno	Test For Not Ordered

Format: **test*** *dst:src1*
reg

Description: Stores a true (01H) in *dst* if the logical AND of the condition code and opcode mask part is not zero. Otherwise, the instruction stores a false (00H) in *dst*. For **testno** (Unordered), a true is stored if the condition code is 000_2, otherwise a false is stored.

The following table shows the condition-code mask for each instruction. The mask is in bits 0-2 of the opcode.

Instruction	Mask	Condition
testno	000_2	Unordered
testg	001_2	Greater
teste	010_2	Equal
testge	011_2	Greater or equal
testl	100_2	Less
testne	101_2	Not equal
testle	110_2	Less or equal
testo	111_2	Ordered

Action: For all **TEST<cc>** except **testno**:

```
if((mask & AC.cc) != 000₂)
    src1 = 1;    #true value
else
    src1 = 0;    #false value
```

testno:

```
if(AC.cc == 000₂)
    src1 = 1;    #true value
else
    src1 = 0;    #false value
```

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example: `# Assume AC.cc = 100`$_2$

```
testl g9             # g9 = 0x00000001
```

Opcode:

teste	22H	COBR
testne	25H	COBR
testl	24H	COBR
testle	26H	COBR
testg	21H	COBR
testge	23H	COBR
testo	27H	COBR
testno	20H	COBR

See Also: **cmpi, cmpdeci, cmpinci**

6.2.69 xnor, xor

Mnemonic:

xnor	Exclusive Nor
xor	Exclusive Or

Format:

xnor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg
xor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise XNOR (**xnor** instruction) or XOR (**xor** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

xnor:

dst = ~(src2 | src1) | (src2 & src1);

xor:

dst = (src2 | src1) & ~(src2 & src1);

Faults: STANDARD Refer to section 6.1.6, Faults (pg. 6-5).

Example:

```
xnor r3, r9, r12    # r12 = r9 XNOR r3
xor g1, g7, g4      # g4 = g7 XOR g1
```

Opcode:

xnor	589H	REG
xor	586H	REG

See Also: **and, andnot, nand, nor, not, notand, notor, or, ornot**

7

PROCEDURE CALLS

CHAPTER 7
PROCEDURE CALLS

This chapter describes mechanisms for making procedure calls, which include branch-and-link instructions, built-in call and return mechanism, call instructions (**call**, **callx**, **calls**), return instruction (**ret**) and call actions caused by interrupts and faults.

The i960® processor architecture supports two methods for making procedure calls:

- A RISC-style branch-and-link: a fast call best suited for calling procedures that do not call other procedures.
- An integrated call and return mechanism: a more versatile method for making procedure calls, providing a highly efficient means for managing a large number of registers and the program stack.

On a branch-and-link (**bal**, **balx**), the processor branches and saves a return IP in a register. The called procedure uses the same set of registers and the same stack as the calling procedure. On a call (**call**, **callx**, **calls**) or when an interrupt or fault occurs, the processor also branches to a target instruction and saves a return IP. Additionally, the processor saves the local registers and allocates a new set of local registers and a new stack for the called procedure. The saved context is restored when the return instruction (**ret**) executes.

In many RISC architectures, a branch-and-link instruction is used as the base instruction for coding a procedure call. The user program then handles register and stack management for the call. Since the i960 architecture provides a fully integrated call and return mechanism, coding calls with branch-and-link are not necessary. Additionally, the integrated call is much faster than typical RISC-coded calls.

The branch-and-link instruction in the i960 processor family, therefore, is used primarily for calling leaf procedures. Leaf procedures call no other procedures; they reside at the “leaves” of the call tree.

In the i960 architecture the integrated call and return mechanism is used in two ways:

- explicit calls to procedures in a user’s program
- implicit calls to interrupt and fault handlers

The remainder of this chapter explains the generalized call mechanism used for explicit and implicit calls and call and return instructions.

The processor performs two call actions:

local When a local call is made, execution mode remains unchanged and the stack frame for the called procedure is placed on the *local stack*. The local stack refers to the stack of the calling procedure.

supervisor When a supervisor call is made from user mode, execution mode is switched to supervisor and the stack frame for the called procedure is placed on the *supervisor stack.*

When a supervisor call is issued from supervisor mode, the call degenerates into a local call (i.e., no mode nor stack switch).

Explicit procedure calls can be made using several instructions. Local call instructions **call** and **callx** perform a local call action. With **call** and **callx**, the called procedure's IP is included as an operand in the instruction.

A system call is made with **calls**. This instruction is similar to **call** and **callx**, except that the processor obtains the called procedure's IP from the *system procedure table*. A system call, when executed, is directed to perform either the local or supervisor call action. These calls are referred to as *system-local* and *system-supervisor* calls, respectively. A system-supervisor call is also referred to as a *supervisor call*.

7.1 CALL AND RETURN MECHANISM

At any point in a program, the i960 processor has access to the global registers, a local register set and the procedure stack. A subset of the stack allocated to the procedure is called the stack frame.

- When a call executes, a new stack frame is allocated for the called procedure. The processor also saves the current local register set, freeing these registers for use by the newly called procedure. In this way, every procedure has a unique stack and a unique set of local registers.
- When a return executes, the current local register set and current stack frame are deallocated. The previous local register set and previous stack frame are restored.

7.1.1 Local Registers and the Procedure Stack

The processor automatically allocates a set of 16 local registers for each procedure. Since local registers are on-chip, they provide fast access storage for local variables. Of the 16 local registers, 13 are available for general use; r0, r1 and r2 are reserved for linkage information to tie procedures together.

The processor does not always clear or initialize the set of local registers assigned to a new procedure. Therefore, initial register contents are unpredictable. Also, because the processor does not initialize the local register save area in the newly created stack frame for the procedure, its contents are equally unpredictable.

The procedure stack can be located anywhere in the address space and grows from low addresses to high addresses. It consists of contiguous frames, one frame for each active procedure. Local registers for a procedure are assigned a save area in each stack frame (Figure 7-1). The procedure stack, available to the user, begins after this save area.

To increase procedure call speed, the architecture allows an implementation to cache the saved local register sets on-chip. Thus, when a procedure call is made, the contents of the current set of local registers often do not have to be written out to the save area in the stack frame in memory. Refer to section 7.1.4, Caching Local Register Sets (pg. 7-7) and section 7.1.4.1, Reserving Local Register Sets for High Priority Interrupts (pg. 7-8) for more about local registers and procedure stack interrelations.

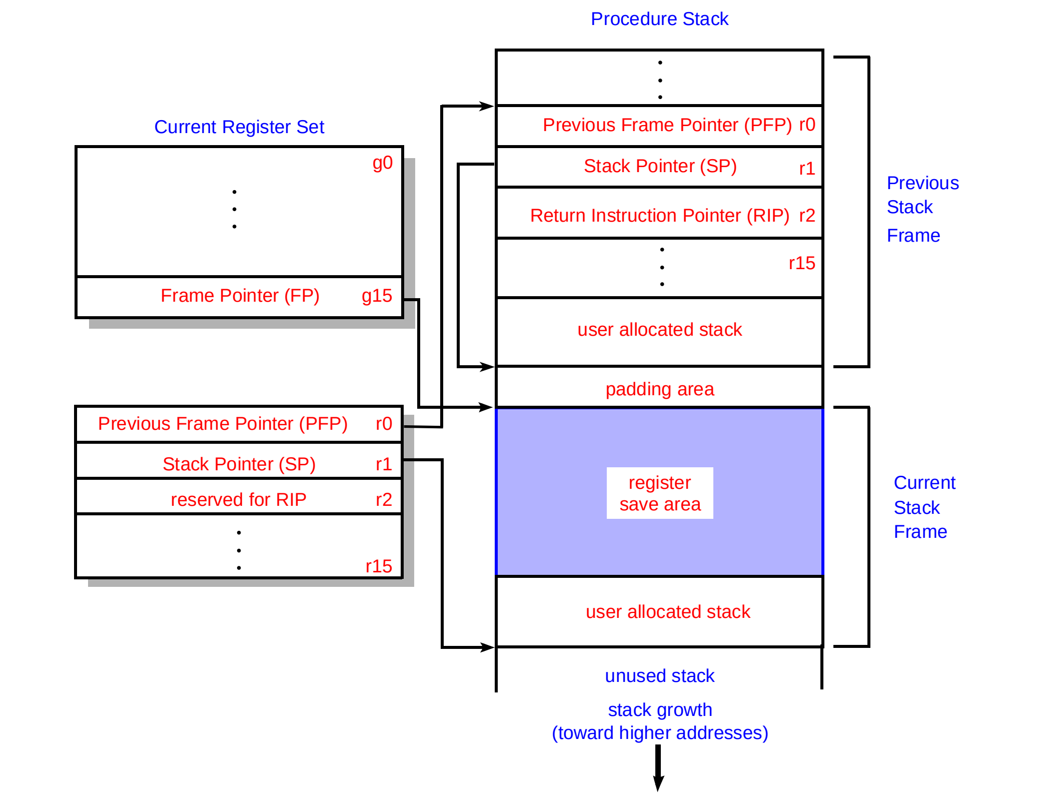


Figure 7-1. Procedure Stack Structure and Local Registers

7.1.2 Local Register and Stack Management

Global register g15 (FP) and local registers r0 (PFP), r1 (SP) and r2 (RIP) contain information to link procedures together and link local registers to the procedure stack (Figure 7-1). The following subsections describe this linkage information.

7.1.2.1 Frame Pointer

The frame pointer is the current stack frame's first byte address. It is stored in global register g15, the frame pointer (FP) register. The FP register is always reserved for the frame pointer; do not use g15 for general storage.

Stack frame alignment is defined for each implementation of the i960 processor family, according to an SALIGN parameter. In the i960® Rx I/O processor, stacks are aligned on 16-byte boundaries (see Figure 7-1). When the processor needs to create a new frame on a procedure call, it adds a padding area to the stack so that the new frame starts on a 16-byte boundary.

7.1.2.2 Stack Pointer

The stack pointer is the byte-aligned address of the stack frame's next unused byte. The stack pointer value is stored in local register r1, the stack pointer (SP) register. The procedure stack grows upward (i.e., toward higher addresses). When a stack frame is created, the processor automatically adds 64 to the frame pointer value and stores the result in the SP register. This action creates the register save area in the stack frame for the local registers.

The program must modify the SP register value when data is stored or removed from the stack. The i960 architecture does not provide an explicit push or pop instruction to perform this action. This is typically done by adding the size of all pushes to the stack in one operation.

7.1.2.3 Considerations When Pushing Data onto the Stack

Care should be taken in writing to stack in the presence of unforeseen faults and interrupts. In the general case, to ensure that the data written to the stack is not corrupted by a fault or interrupt record, the SP should be incremented first to allocate the space, and then the data should be written to the allocated space:

```
mov   sp,r4
addo  24,sp,sp
st    data,(r4)
      ...
st    data,20(r4)
```

7.1.2.4 Considerations When Popping Data off the Stack

For reasons similar to those discussed in the previous section, care should be taken in reading the stack in the presence of unforeseen faults and interrupts. In the general case, to ensure that data about to be popped off the stack is not corrupted by a fault or interrupt record, the data should be read first and then the sp should be decremented:

```
subo  24,sp,r4
ld    20(r4),rn
      ...
ld    (r4),rn
mov   r4,sp
```

7.1.2.5 Previous Frame Pointer

The previous frame pointer is the previous stack frame's first byte address. This address's upper 28 bits are stored in local register r0, the previous frame pointer (PFP) register. The four least-significant bits of the PFP are used to store the return type field. See Table 7-2 and Table 7-3 for more information on the PFP and the return-type field.

7.1.2.6 Return Type Field

PFP register bits 0 through 3 contain return type information for the calling procedure. When a procedure call is made — either explicit or implicit — the processor records the call type in the return type field. The processor then uses this information to select the proper return mechanism when returning to the calling procedure. The use of this information is described in section 7.8, RETURNS (pg. 7-20).

7.1.2.7 Return Instruction Pointer

The actual RIP register (r2) is reserved by the processor to support the call and return mechanism and must not be used by software; the actual value of RIP is unpredictable at all times. For example, an implicit procedure call (fault or interrupt) can occur at any time and modify the RIP. An OPERATION.INVALID_OPERAND fault is generated when attempting to write the RIP.

The image of the RIP register in the stack frame is used by the processor to determine that frame's return instruction address. When a call is made, the processor saves the address of the instruction after the call in the image of the RIP register in the calling frame.

7.1.3 Call and Return Action

To clarify how procedures are linked and how the local registers and stack are managed, the following sections describe a general call and return operation and the operations performed with the FP, SP, PFP and RIP registers.

The events for call and return operations are given in a logical order of operation. The i960 Rx I/O processor can execute independent operations in parallel; therefore, many of these events execute simultaneously. For example, to improve performance, the processor often begins prefetching of the target instruction for the call or return before the operation is complete.

7.1.3.1 Call Operation

When a **call, calls** or **callx** instruction is executed or an implicit call is triggered:

1. The processor stores the instruction pointer for the instruction following the call in the current stack's RIP register (r2).

2. The current local registers — including the PFP, SP and RIP registers — are saved, freeing these for use by the called procedure. The local registers are saved in the on-chip local register cache if space is available.

3. The frame pointer (g15) for the calling procedure is stored in the current stack's PFP register (r0). The return type field in the PFP register is set according to the call type which is performed. See section 7.8, RETURNS (pg. 7-20).

4. For a local or system-local call, a new stack frame is allocated by using the old stack pointer value saved in step 2. This value is first rounded to the next 16-byte boundary to create a new frame pointer, then stored in the FP register. Next, 64 bytes are added to create the new frame's register save area. This value is stored in the SP register.

   For an interrupt call from user mode, the current interrupt stack pointer value is used instead of the value saved in step 2.

   For a system-supervisor call from user mode, the current Supervisor Stack Pointer (SSP) value is used instead of the value saved in step 2.

5. The instruction pointer is loaded with the address of the first instruction in the called procedure. The processor gets the new instruction pointer from the **call**, the system procedure table, the interrupt table or the fault table, depending on the type of call executed.

Upon completion of these steps, the processor begins executing the called procedure. Sometime before a return or nested call, the local register set is bound to the allocated stack frame.

7.1.3.2 Return Operation

A return from any call type — explicit or implicit — is always initiated with a return (**ret**) instruction. On a return, the processor performs these operations:

1. The current stack frame and local registers are deallocated by loading the FP register with the value of the PFP register.
2. The local registers for the return target procedure are retrieved. The registers are usually read from the local register cache; however, in some cases, these registers have been flushed from register cache to memory and must be read directly from the save area in the stack frame.
3. The processor sets the instruction pointer to the value of the RIP register.

Upon completion of these steps, the processor executes the instruction to which it returns. The frames created before the **ret** instruction was executed will be overwritten by later implicit or explicit call operations.

7.1.4 Caching Local Register Sets

Actual implementations of the i960 architecture may cache some number of local register sets within the processor to improve performance. Local registers are typically saved and restored from the local register cache when calls and returns are executed. Other overhead associated with a call or return is performed in parallel with this data movement.

When the number of nested procedures exceeds local register cache size, local register sets must at times be saved to (and restored from) their associated save areas in the procedure stack. Because these operations require access to external memory, this local cache miss affects call and return performance.

When a call is made and no frames are available in the register cache, a register set in the cache must be saved to external memory to make room for the current set of local registers in the cache. See 4.2, LOCAL REGISTER CACHE (pg. 4-2). This action is referred to as a frame spill. The oldest set of local registers stored in the cache is spilled to the associated local register save area in the procedure stack. Figure 7-2 illustrates a call operation with and without a frame spill.

Similarly, when a return is made and the local register set for the target procedure is not available in the cache, these local registers must be retrieved from the procedure stack in memory. This operation is referred to as a frame fill. Figure 7-3 illustrates return operations with and without frame fills.

The **flushreg** instruction, described in 6.2.30, flushreg (pg. 6-54), writes all local register sets (except the current one) to their associated stack frames in memory. The register cache is then invalidated, meaning that all flushed register sets are restored from their save areas in memory.

For most programs, the existence of the multiple local register sets and their saving/restoring in the stack frames should be transparent. However, there are some special cases:

- A store to the register save area in memory does not necessarily update a local register set, unless user software executes **flushreg** first.
- Reading from the register save area in memory does not necessarily return the current value of a local register set, unless user software executes **flushreg** first.
- There is no mechanism, including **flushreg**, to access the current local register set with a read or write to memory.
- **flushreg** must be executed sometime before returning from the current frame if the current procedure modifies the PFP in register r0, or else the behavior of the **ret** instruction is not predictable.
- The values of the local registers r2 to r15 in a new frame are undefined.

flushreg is commonly used in debuggers or fault handlers to gain access to all saved local registers. In this way, call history may be traced back through nested procedures.

7.1.4.1 Reserving Local Register Sets for High Priority Interrupts

To decrease interrupt latency for high priority interrupts, software can limit the number of frames available to all remaining code. This includes code that is either in the executing state (non-interrupted) or code that is in the interrupted state but has a process priority less than 28. For the purposes of discussion here, this remaining code will be referred to as *non-critical code*. Specifying a limit for non-critical code ensures that some number of free frames are available to high-priority interrupt service routines. Software can specify the limit for non-critical code by writing bits 10 through 8 of the register cache configuration word in the PRCB (see Table 11-8., Process Control Block Configuration Words (pg. 11-18)). The value indicates how many frames within the register cache may be used by non-critical code before a frame needs to be flushed to external memory. The programmed limit is used only when a frame is pushed, which occurs only for an implicit or explicit call.

Allowed values of the programmed limit range from 0 to 7. Setting the value to 0 reserves no frames for high-priority interrupts. Setting the value to 7 causes the register cache to become disabled for non-critical code. See section 11.4.2, Process Control Block – PRCB (pg. 11-17).

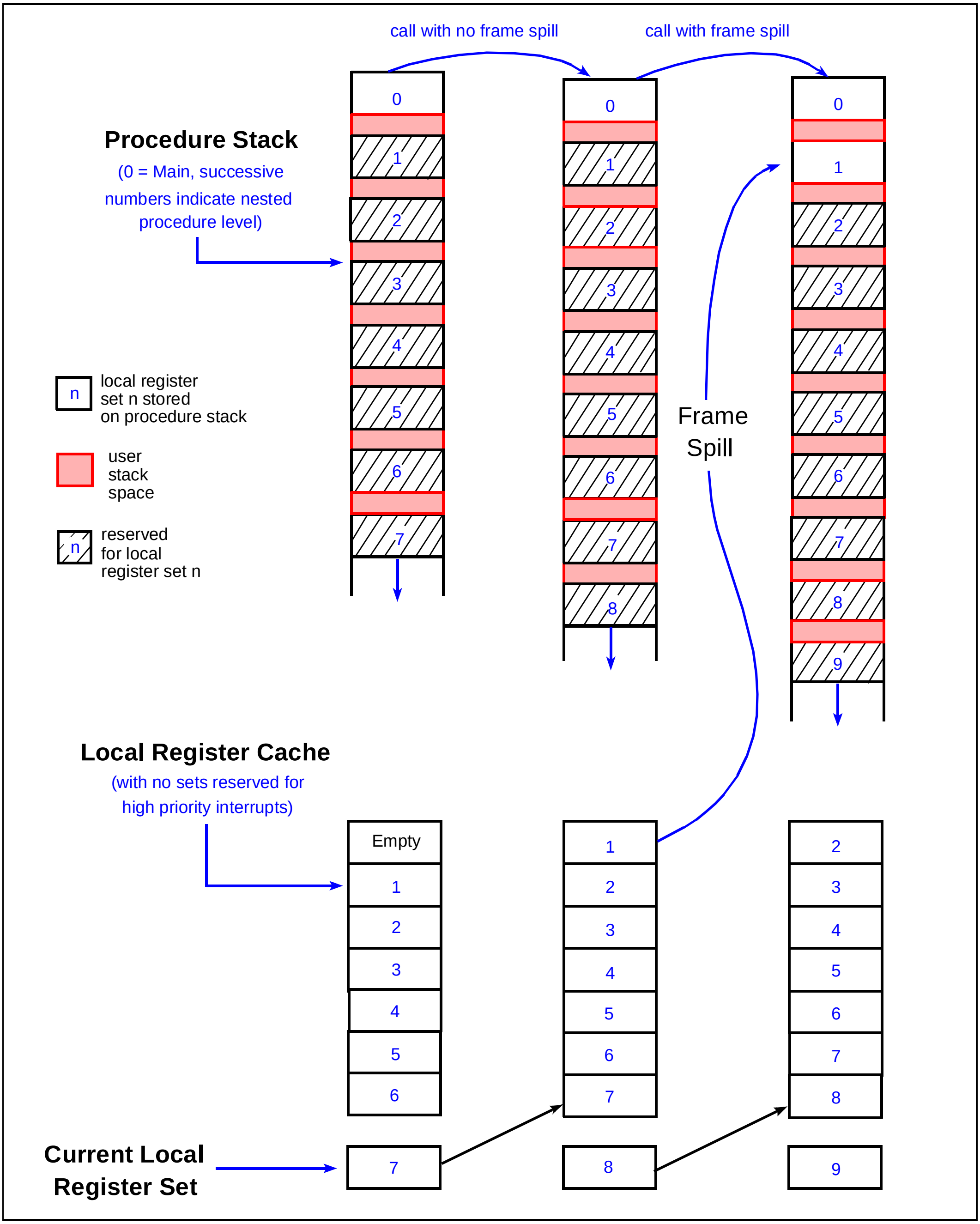

call with no frame spill
call with frame spill
Procedure Stack
(0 = Main, successive numbers indicate nested procedure level)
n
local register set n stored on procedure stack
user stack space
n
reserved for local register set n
0
1
2
3
4
5
6
7
8
9
Frame Spill
Local Register Cache
(with no sets reserved for high priority interrupts)
Empty
Current Local Register Set

Figure 7-2. Frame Spill

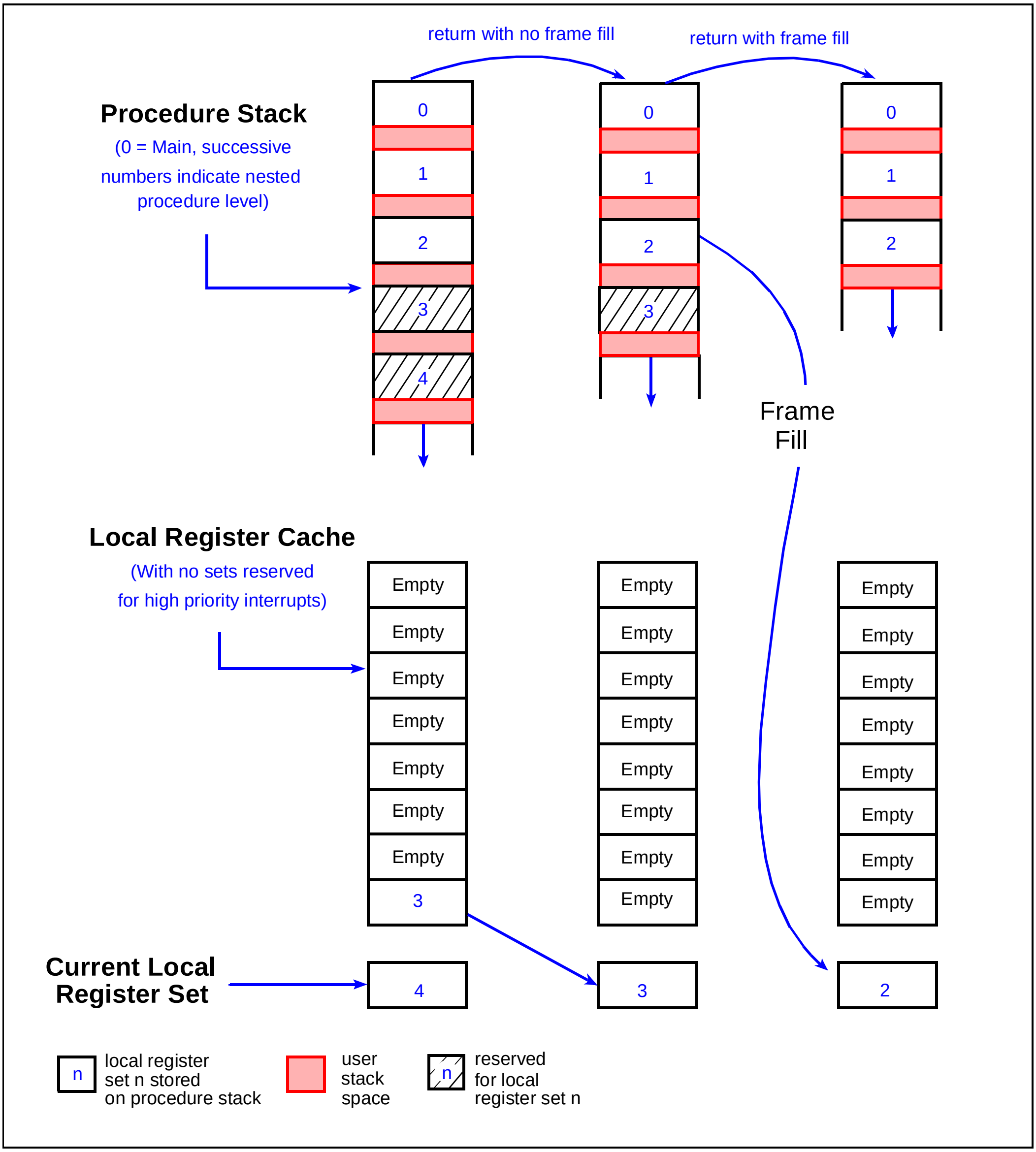


Figure 7-3. Frame Fill

7.1.5 Mapping Local Registers to the Procedure Stack

Each local register set is mapped to a register save area of its respective frame in the procedure stack (Figure 7-1). Saved local register sets are frequently cached on-chip rather than saved to memory. This is not a write-through cache. Local register set contents are not saved automatically to the save area in memory when the register set is cached. This would cause a significant performance loss for call operations.

Also, no automatic update policy is implemented for the register cache. If the register save area in memory for a cached register set is modified, there is no guarantee that the modification will be reflected when the register set is restored. For a frame spill, the set must be flushed to memory prior to the modification for the modification to be valid.

The **flushreg** instruction causes the contents of all cached local register sets to be written (flushed) to their associated stack frames in memory. The register cache is then invalidated, meaning that all flushed register sets are restored from their save areas in memory. The current set of local registers is not written to memory. **flushreg** is commonly used in debuggers or fault handlers to gain access to all saved local registers. In this way, call history may be traced back through nested procedures. **flushreg** is also used when implementing task switches in multitasking kernels. The procedure stack is changed as part of the task switch. To change the procedure stack, **flushreg** is executed to update the current procedure stack and invalidate all entries in the local register cache. Next, the procedure stack is changed by directly modifying the FP and SP registers and executing a call operation. After **flushreg** executes, the procedure stack may also be changed by modifying the previous frame in memory and executing a return operation.

When a set of local registers is assigned to a new procedure, the processor may or may not clear or initialize these registers. Therefore, initial register contents are unpredictable. Also, the processor does not initialize the local register save area in the newly created stack frame for the procedure; its contents are equally unpredictable.

7.2 MODIFYING THE PFP REGISTER

The FP must not be directly modified by user software or risk corrupting the local registers. Instead, implement context switches by modifying the PFP.

Modification of the PFP is typically for context switches; as part of the switch, the active procedure changes the pointer to the frame that it will return to (previous frame pointer — PFP). Great care should be taken in modifying the PFP. In the general case, a **flushreg** must be issued before and after modifying the PFP when the local register cache is enabled (see Example 7-1). This requirement ensures the correct operation of a context switch on all i960 processors in all situations.

Example 7-1. flushreg

```
# Do a context switch.
# Assume PFP = 0x5000.
flushreg        # Flush Frames to correct address.
lda 0x8000,pfp
flushreg        # Ensure that "ret" gets updated PFP.
ret
```

The **flushreg** before the modification is necessary to ensure that the frame of the previous context (mapped to 0x5000 in the example) is “spilled” to the proper external memory address and removed from the local register cache. If the **flushreg** before the modification was omitted, a **flushreg** (or implicit frame spill due to an interrupt) after the modification of PFP would cause the frame of the previous context to be written to the wrong location in external memory.

The **flushreg** after the modification ensures that outstanding results are completely written to the PFP before a subsequent **ret** instruction can be executed. Recall that the **ret** instruction uses the low-order 4 bits of the PFP to select which **ret** function to perform. Requiring the **flushreg** after the PFP modification allows an i960 implementation to implement a simple mechanism that quickly selects the **ret** function at the time the **ret** instruction is issued and provides a faster return operation.

Note the **flushreg** after the modification will execute very quickly because the local register cache has already been flushed by the **flushreg** before; only synchronization of the PFP will be performed. i960 processor implementations may provide other mechanisms to ensure PFP synchronization in addition to **flushreg**, but a **flushreg** after a PFP modification is ensured to work on all i960 processors.

7.3 PARAMETER PASSING

Parameters are passed between procedures in two ways:

value — Parameters are passed directly to the calling procedure as part of the call and return mechanism. This is the fastest method of passing parameters.

reference — Parameters are stored in an argument list in memory and a pointer to the argument list is passed in a global register.

When passing parameters by value, the calling procedure stores the parameters to be passed in global registers. Since the calling procedure and the called procedure share the global registers, the called procedure has direct access to the parameters after the call.

When a procedure needs to pass more parameters than will fit in the global registers, they can be passed by reference. Here, parameters are placed in an argument list and a pointer to the argument list is placed in a global register.

The argument list can be stored anywhere in memory; however, a convenient place to store an argument list is in the stack for a calling procedure. Space for the argument list is created by incrementing the SP register value. If the argument list is stored in the current stack, the argument list is automatically deallocated when no longer needed.

A procedure receives parameters from — and returns values to — other calling procedures. To do this successfully and consistently, all procedures must agree on the use of the global registers.

Parameter registers pass values into a function. Up to 12 parameters can be passed by value using the global registers. If the number of parameters exceeds 12, additional parameters are passed using the calling procedure's stack; a pointer to the argument list is passed in a pre-designated register. Similarly, several registers are set aside for return arguments and a return argument block pointer is defined to point to additional parameters. If the number of return arguments exceeds the available number of return argument registers, the calling procedure passes a pointer to an argument list on its stack where the remaining return values will be placed. Example 7-2 illustrates parameter passing by value and by reference.

Local registers are automatically saved when a call is made. Because of the local register cache, they are saved quickly and with no external bus traffic. The efficiency of the local register mechanism plays an important role in two cases when calls are made:

1. When a procedure is called which contains other calls, global parameter registers should be moved to working local registers at the beginning of the procedure. In this way, parameter registers are freed and nested calls are easily managed. The register move instruction necessary to perform this action is very fast; the working parameters — now in local registers — are saved efficiently when nested calls are made.

2. When other procedures are nested within an interrupt or fault procedure, the procedure must preserve all normally non-preserved parameter registers, such as the global registers. This is necessary because the interrupt or fault occurs at any point in the user's program and a return from an interrupt or fault must restore the exact processor state. The interrupt or fault procedure can move non-preserved global registers to local registers before the nested call.

Example 7-2. Parameter Passing Code Example

```
# Example of parameter passing . . .
# C-source:int a,b[10];
#          a = proc1(a,1,’x’,&b[0]);
#          assembles to ...
     mov   r3,g0# value of a
     ldconst1,g1# value of 1
     ldconst120,g2# value of “x”
     lda   0x40(fp),g3# reference to b[10]
     call _proc1
     mov   g0,r3#save return value in “a”
           .
           .
_proc1:
     movq  g0,r4# save parameters
           .
           .    # other instructions in procedure
           .    # and nested calls
     mov   r3,g0# load return parameter
     ret
```

7.4 LOCAL CALLS

A local call does not cause a stack switch. A local call can be made two ways:

- with the **call** and **callx** instructions; or
- with a system-local call as described in section 7.5, SYSTEM CALLS (pg. 7-15).

call specifies the address of the called procedures as the IP plus a signed, 24-bit displacement (i.e., -2^{23} to 2^{23} - 4). **callx** allows any of the addressing modes to be used to specify the procedure address. The IP-with-displacement addressing mode allows full 32-bit IP-relative addressing.

When a local call is made with a **call** or **callx**, the processor performs the same operation as described in section 7.1.3.1, Call Operation (pg. 7-6). The target IP for the call is derived from the instruction’s operands and the new stack frame is allocated on the current stack.

7.5 SYSTEM CALLS

A system call is a call made via the system procedure table. It can be used to make a system-local call — similar to a local call made with **call** and **callx** in the sense that there is no stack nor mode switch — or a system supervisor call. A system call is initiated with **calls**, which requires a procedure number operand. The procedure number provides an index into the system procedure table, where the processor finds IPs for specific procedures.

Using an i960 processor language assembler, a system procedure is directly declared using the .sysproc directive. At link time, the optimized call directive, callj, is replaced with a **calls** when a system procedure target is specified. (Refer to current i960 processor assembler documentation for a description of the .sysproc and callj directives.)

The system call mechanism offers two benefits. First, it supports application software portability. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not need to be changed each time the implementation of the kernel services is modified. Only the entries in the system procedure table must be changed. Second, the ability to switch to a different execution mode and stack with a system supervisor call allows kernel procedures and data to be insulated from applications code. This benefit is further described in 3.7, USER-SUPERVISOR PROTECTION MODEL (pg. 3-21).

7.5.1 System Procedure Table

The system procedure table is a data structure for storing IPs to system procedures. These can be procedures which software can access through (1) a system call or (2) the fault handling mechanism. Using the system procedure table to store IPs for fault handling is described in 9.1, FAULT HANDLING OVERVIEW (pg. 9-1).

Figure 7-4 shows the system procedure table structure. It is 1088 bytes in length and can have up to 260 procedure entries. At initialization, the processor caches a pointer to the system procedure table. This pointer is located in the PRCB. The following subsections describe this table’s fields.

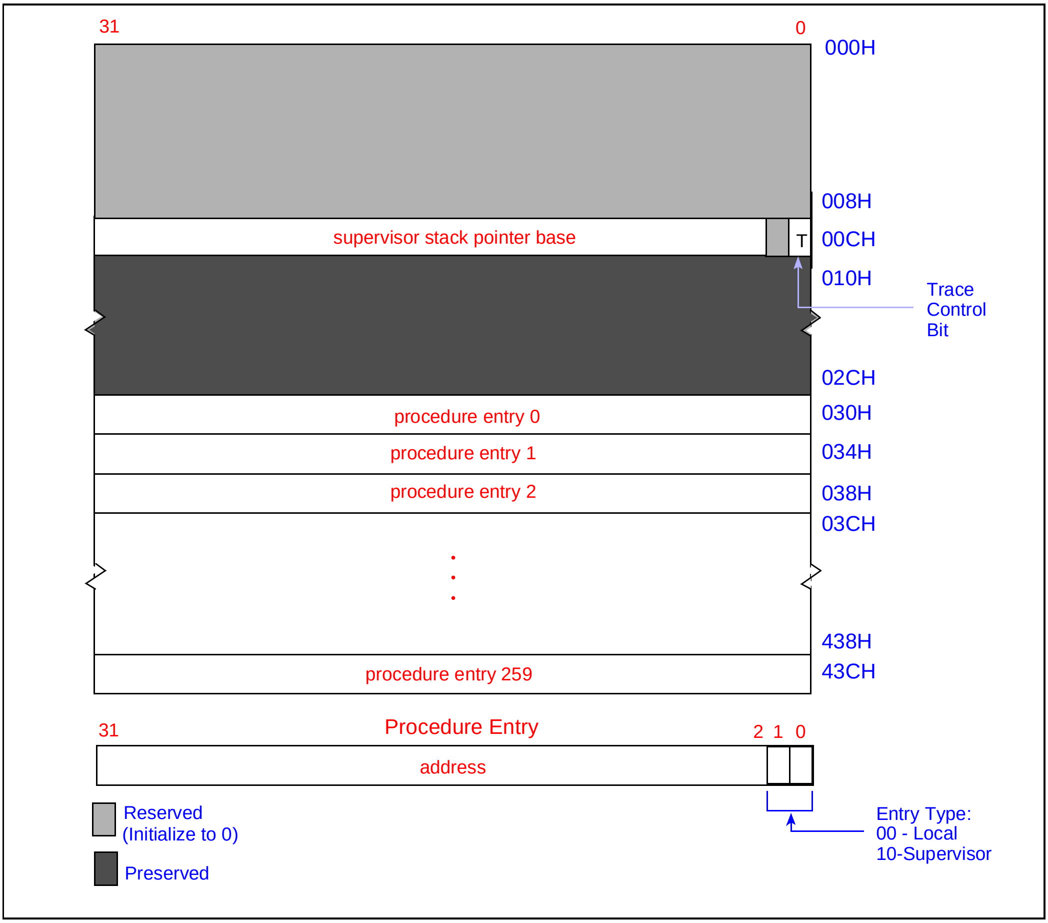


Figure 7-4. System Procedure Table

7.5.1.1 Procedure Entries

A procedure entry in the system procedure table specifies a procedure's location and type. Each entry is one word in length and consists of an address (IP) field and a type field. The address field gives the address of the first instruction of the target procedure. Since all instructions are word aligned, only the entry's 30 most significant bits are used for the address. The entry's two least-significant bits specify entry type. The procedure entry type field indicates call type: system-local call or system-supervisor call (Table 7-1). On a system call, the processor performs different actions depending on the type of call selected.

Table 7-1. Encodings of Entry Type Field in System Procedure Table

Encoding	Call Type
00	System-Local Call
01	Reserved[1]
10	System-Supervisor Call
11	Reserved[1]

[1] Calls with reserved entry types have unpredictable behavior.

7.5.1.2 Supervisor Stack Pointer

When a system-supervisor call is made, the processor switches to a new stack called the *supervisor stack*, if not already in supervisor mode. The processor gets a pointer to this stack from the supervisor stack pointer field in the system procedure table (Figure 7-4) during the reset initialization sequence and caches the pointer internally. Only the 30 most significant bits of the supervisor stack pointer are given. The processor aligns this value to the next 16-byte boundary to determine the first byte of the new stack frame.

7.5.1.3 Trace Control Bit

The trace control bit (byte 12, bit 0) specifies the new value of the trace enable bit in the PC register (PC.te) when a system-supervisor call causes a switch from user mode to supervisor mode. Setting this bit to 1 enables tracing in the supervisor mode; setting it to 0 disables tracing. The use of this bit is described in section 10.1.2, PC Trace Enable Bit and Trace-Fault-Pending Flag (pg. 10-3).

7.5.2 System Call to a Local Procedure

When a **calls** instruction references an entry in the system procedure table with an entry type of 00, the processor executes a system-local call to the selected procedure. The action that the processor performs is the same as described in section 7.1.3.1, Call Operation (pg. 7-6). The call's target IP is taken from the system procedure table and the new stack frame is allocated on the current stack, and the processor does not switch to supervisor mode. The **calls** algorithm is described in 6.2.14, calls (pg. 6-24).

7.5.3 System Call to a Supervisor Procedure

When a **calls** instruction references an entry in the system procedure table with an entry type of 10_2, the processor executes a system-supervisor call to the selected procedure. The call's target IP is taken from the system procedure table.

The processor performs the same action as described in section 7.1.3.1, Call Operation (pg. 7-6), with the following exceptions:

- If the processor is in user mode, it switches to supervisor mode.
- If a mode switch occurs, SP is read from the Supervisor Stack Pointer (SSP) base. A new frame for the called procedure is placed at the location pointed to after alignment of SP.
- If no mode switch occurs, the new frame is allocated on the current stack.
- If a mode switch occurs, the state of the trace enable bit in the PC register is saved in the return type field in the PFP register. The trace enable bit is then loaded from the trace control bit in the system procedure table.
- If no mode switch occurs, the value 000_2 (**calls** instruction) or 001_2 (fault call) is saved in the return type field of the pfp register.

When the processor switches to supervisor mode, it remains in that mode and creates new frames on the supervisor stack until a return is performed from the procedure that caused the original switch to supervisor mode. While in supervisor mode, either the local call instructions (**call** and **callx**) or **calls** can be used to call procedures.

The user-supervisor protection model and its relationship to the supervisor call are described in section 3.7, USER-SUPERVISOR PROTECTION MODEL (pg. 3-21).

7.6 USER AND SUPERVISOR STACKS

When using the user-supervisor protection mechanism, the processor maintains separate stacks in the address space. One of these stacks — the user stack — is for procedures executed in user mode; the other stack — the supervisor stack — is for procedures executed in supervisor mode.

The user and supervisor stacks are identical in structure (Figure 7-1). The base stack pointer for the supervisor stack is automatically read from the system procedure table and cached internally during initialization. Each time a user-to-supervisor mode switch occurs, the cached supervisor stack pointer base is used for the starting point of the new supervisor stack. The base stack pointer for the user stack is usually created in the initialization code. See section 11.2, 80960Rx INITIALIZATION (pg. 11-2). The base stack pointers must be aligned to a 16-byte boundary; otherwise, the first frame pointer on the interrupt stack is rounded up to the previous 16-byte boundary.

7.7 INTERRUPT AND FAULT CALLS

The architecture defines two types of implicit calls that make use of the call and return mechanism: interrupt-handling procedure calls and fault-handling procedure calls. A call to an interrupt procedure is similar to a system-supervisor call. Here, the processor obtains pointers to the interrupt procedures through the interrupt table. The processor always switches to supervisor mode on an interrupt procedure call.

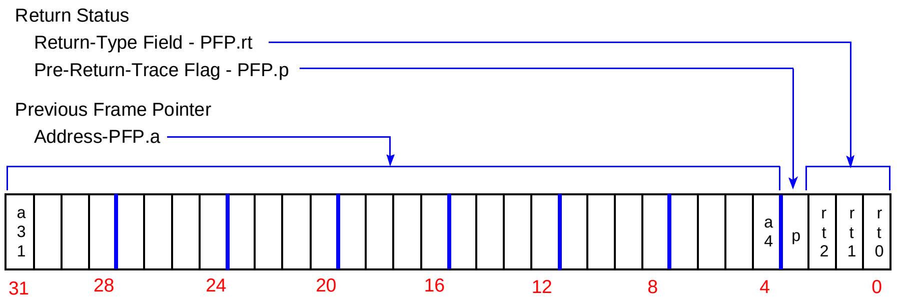
A call to a fault procedure is similar to a system call. Fault procedure calls can be local calls or supervisor calls. The processor obtains pointers to fault procedures through the fault table and (optionally) through the system procedure table.

When a fault call or interrupt call is made, a fault record or interrupt record is placed in the newly generated stack frame for the call. These records hold the machine state and information to identify the fault or interrupt. When a return from an interrupt or fault is executed, machine state is restored from these records. See CHAPTER 8, INTERRUPTS and CHAPTER 9, FAULTS for more information on the structure of the fault and interrupt records.

7.8 RETURNS

The return (**ret**) instruction provides a generalized return mechanism that can be used to return from any procedure that was entered by **call**, **calls**, **callx**, an interrupt call or a fault call. When **ret** executes, the processor uses the information from the return-type field in the PFP register (Table 7-2) to determine the type of return action to take.

Table 7-2. Previous Frame Pointer Register – PFP



return-type field indicates the type of call which was made. Table 7-3 shows the return-type field encoding for the various calls: local, supervisor, interrupt and fault.

trace-on-return flag (PFP.rt0 or bit 0 of the return-type field) stores the trace enable bit value when an explicit system-supervisor call is made from user mode. When the call is made, the PC register trace enable bit is saved as the trace-on-return flag and then replaced by the trace controls bit in the system procedure table. On a return, the trace enable bit's original value is restored. This mechanism allows instruction tracing to be turned on or off when a supervisor mode switch occurs. See section 10.5.2.1, Tracing on Explicit Call (pg. 10-13).

prereturn-trace flag (PFP.p) is used in conjunction with call-trace and prereturn-trace modes. If call-trace mode is enabled when a call is made, the processor sets the prereturn-trace flag; otherwise it clears the flag. Then, if this flag is set and prereturn-trace mode is enabled, a prereturn trace event is generated on a return, before any actions associated with the return operation are performed. See section 10.2, TRACE MODES (pg. 10-3) for a discussion of interaction between call-trace and prereturn-trace modes with the prereturn-trace flag.

Table 7-3. Encoding of Return Status Field

Return Status Field	Call Type	Return Action
000	Local call (system-local call or system-supervisor call made from supervisor mode)	Local return (return to local stack; no mode switch)
001	Fault call	Fault return
01t	System-supervisor from user mode	Supervisor return (return to user stack, mode switch to user mode, trace enable bit is replaced with the t[1] bit stored in the PFP register on the call)
100	reserved [2]	
101	reserved[2]	
110	reserved[2]	
111	Interrupt call	Interrupt return

NOTES:

1. “t” denotes the trace-on-return flag; used only for system supervisor calls which cause a user-to-supervisor mode switch.
2. This return type results in unpredictable behavior.

7.9 BRANCH-AND-LINK

A branch-and-link is executed using either the branch-and-link instruction (**bal**) or branch-and-link-extended instruction (**balx**). When either instruction executes, the processor branches to the first instruction of the called procedure (the target instruction), while saving a return IP for the calling procedure in a register. The called procedure uses the same set of local registers and stack frame as the calling procedure:

- For **bal**, the return IP is automatically saved in global register g14
- For **balx**, the return IP instruction is saved in a register specified by one of the instruction’s operands

A return from a branch-and-link is generally carried out with a **bx** (branch extended) instruction, where the branch target is the address saved with the branch-and-link instruction. The branch-and-link method of making procedure calls is recommended for calls to leaf procedures. Leaf procedures typically call no other procedures. Branch-and-link is the fastest way to make a call, providing the calling procedure does not require its own registers or stack frame.

8

INTERRUPTS

CHAPTER 8
INTERRUPTS

This chapter describes the i960® core processor architecture interrupt mechanism, the i960 Rx I/O processor interrupt controller, peripheral interrupts and secondary PCI interrupt routing. Key topics include the i960 Rx I/O processor's facilities for requesting and posting interrupts, the programmer's interface to the on-chip interrupt controller, interrupt implementation, interrupt latency and how to optimize interrupt performance.

8.1 OVERVIEW

An interrupt is an event that causes a temporary break in program execution so the processor can handle another task. Interrupts commonly request I/O services or synchronize the processor with some external hardware activity. For interrupt handler portability across the i960 processor family, the architecture defines a consistent interrupt state and interrupt-priority-handling mechanism. To manage and prioritize interrupt requests in parallel with processor execution, the i960 Rx I/O processor provides an on-chip programmable interrupt controller.

When the processor is redirected to service an interrupt, it uses a vector number that accompanies the interrupt request to locate the vector entry in the interrupt table. From that entry, it gets an address to the first instruction of the selected interrupt procedure. The processor then makes an implicit call to that procedure.

When the interrupt call is made, the processor uses a dedicated interrupt stack. The processor creates a new frame for the interrupt on this stack and a new set of local registers is allocated to the interrupt procedure. The interrupted program's current state is also saved.

Upon return from the interrupt procedure, the processor restores the interrupted program's state, switches back to the stack that the processor was using prior to the interrupt and resumes program execution.

Since interrupts are handled based on priority, requested interrupts are often saved for later service rather than handled immediately. The mechanism for saving the interrupt is referred to as interrupt posting. Interrupt posting is described in section 8.1.6, Posting Interrupts (pg. 8-7).

The i960 core architecture defines two data structures to support interrupt processing: the interrupt table (see Figure 8-1) and interrupt stack. The interrupt table contains 248 vectors for interrupt handling procedures (eight of which are reserved) and an area for posting software requested interrupts. The interrupt stack prevents interrupt handling procedures from using the stack in use by the application program. It also locates the interrupt stack in a different area of memory than the user and supervisor stack (e.g., fast SRAM).

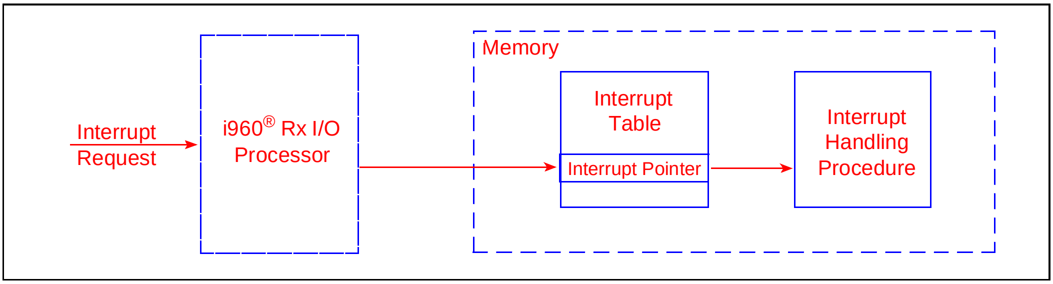


Figure 8-1. Interrupt Handling Data Structures

Requests for interrupt service come from many sources and are prioritized such that instruction execution is redirected only when an interrupt request is of higher priority than that of the executing task. On the i960 Rx I/O processor, interrupt requests may originate from external hardware sources, internal peripherals or software. The i960 Rx I/O processor contains a number of integrated peripherals which may generate interrupts, including:

- DMA Channel 0
- DMA Channel 1
- DMA Channel 2
- Bridge Primary Interface
- Bridge Secondary Interface
- Timers 0 & 1
- Primary ATU
- Secondary ATU
- I^{2}C Bus Interface Unit
- APIC Bus Interface Unit
- Messaging Unit
- Memory Controller

The interrupt controller can also intercept external secondary PCI interrupts and forward them to the primary PCI interrupt pins.

Interrupts are detected with the chip’s 8-bit interrupt port and with a dedicated Non-Maskable Interrupt (NMI#) input in the i960 core processor’s interrupt controller. Interrupt requests originate from software by the **sysctl** instruction. To manage and prioritize all possible interrupts, the processor integrates an on-chip programmable interrupt controller.

8.1.1 The i960® Rx I/O Processor Core Interrupt Architecture

The 80960Rx contains the same core interrupt architecture as many other 80960 family members. Some of the core features include the interrupt record and stack, the way interrupts are posted, and the way interrupt priorities are resolved. These basic architectural features are detailed in the following sections.

8.1.2 Software Requirements For Interrupt Handling

To use the processor's interrupt handling facilities, user software must provide the following items in memory:

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack

These items are established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data structures, the processor handles interrupts automatically and independently from software.

8.1.3 Interrupt Priority

Each procedure pointer's priority is defined by dividing the procedure pointer number by eight. Thus, at each priority level, there are eight possible procedure pointers (e.g., procedure pointers 8-15 have a priority of 1 and procedure pointers 246-255 have a priority of 31). Procedure pointers 0-7 cannot be used because a priority-0 interrupt would never successfully stop execution of a program of any priority. In addition, procedure pointers 244-247 and 249-251 are reserved; therefore, 241 procedure pointers are available to the user.

The processor compares its current priority with the interrupt request priority to determine whether to service the interrupt immediately or to delay service:

- The interrupt is serviced immediately when its priority is higher than the priority of the program or interrupt the processor is currently executing.
- The interrupt is posted as a pending interrupt (not serviced immediately) when the interrupt priority is less than or equal to the processor's current priority.

See section 8.1.4.2, Pending Interrupts (pg. 8-5). When multiple interrupt requests are pending at the same priority level, the request with the highest vector number is serviced first.

Priority-31 interrupts are handled as a special case. Even when the processor is executing at priority level 31, a priority-31 interrupt will interrupt the processor. On the i960 Rx I/O processor, the non-maskable interrupt (NMI#) interrupts priority-31 execution; no interrupt can interrupt an NMI# handler.

8.1.4 Interrupt Table

The interrupt table (see Figure 8-2) is 1028 bytes in length and can be located anywhere in the non-reserved address space. It must be aligned on a word boundary. The processor reads a pointer to the interrupt table byte 0 during initialization. The interrupt table must be located in RAM so the processor can read and write the table's pending interrupt section for software or externally generated interrupts.

The interrupt table is divided into two sections: *vector entries* and *pending interrupts*. Each are described in the subsections that follow.

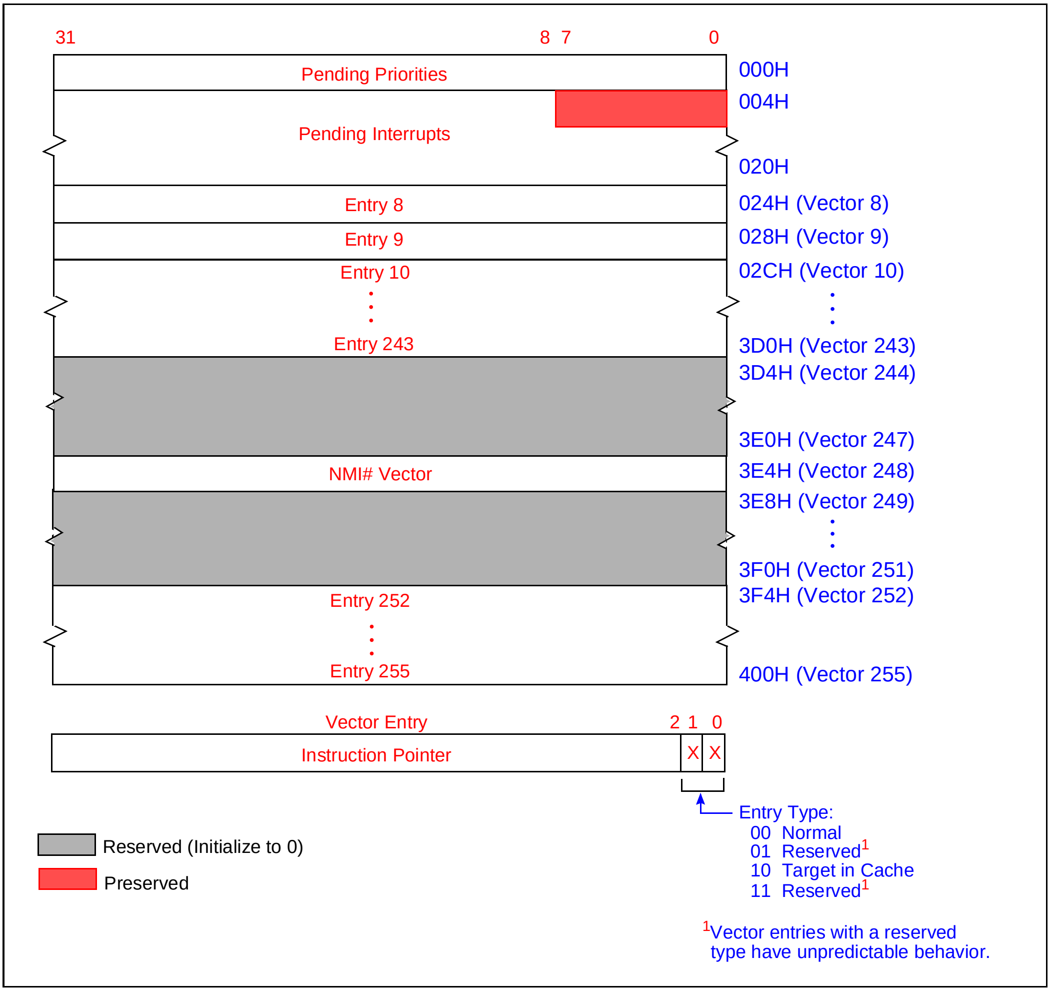


Figure 8-2. Interrupt Table

8.1.4.1 Vector Entries

A vector entry contains a specific interrupt handler's address. When an interrupt is serviced, the processor branches to the address specified by the vector entry.

Each interrupt is associated with an 8-bit vector number that points to a vector entry in the interrupt table. The vector entry section contains 248 word-length entries. Vector numbers 8-243 and 252-255 and their associated vector entries are used for conventional interrupts. Vector number 248 is the NMI# vector. Vector numbers 244-247 and 249-251 are reserved. Vector number 248 and its associated vector entry is used for the non-maskable interrupt (NMI#). Vector numbers 0-7 cannot be used.

Vector entry 248 contains the NMI# handler address. When the processor is initialized, the NMI# vector located in the interrupt table is automatically read and stored in location 0H of internal data RAM. The NMI# vector is subsequently fetched from internal data RAM to improve this interrupt's performance.

The vector entry structure is given at the bottom of Figure 8-2. Each interrupt procedure must begin on a word boundary, so the processor assumes that the vector's two least significant bits are 0. Bits 0 and 1 of an entry indicate entry type: type 00 indicates that the interrupt procedure should be fetched normally; type 10 indicates that the interrupt procedure should be fetched from the locked partition of the instruction cache. Refer to section 8.5.2.2, Caching Interrupt Routines and Reserving Register Frames (pg. 8-47). The other possible entry types are reserved and must not be used.

8.1.4.2 Pending Interrupts

The pending interrupts section comprises the interrupt table's first 36 bytes, divided into two fields: pending priorities (byte offset 0 through 3) and pending interrupts (4 through 35).

Each of the 32 bits in the pending priorities field indicate an interrupt priority. When the processor posts a pending interrupt in the interrupt table, the bit corresponding to the interrupt's priority is set; e.g., when an interrupt with a priority of 10 is posted in the interrupt table, bit 10 is set.

Each of the pending interrupts field's 256 bits represent an interrupt procedure pointer. Byte offset 5 is for vectors 8 through 15, byte offset 6 is for vectors 16 through 23, and so on. Byte offset 4, the first byte of the pending interrupts field, is reserved. When an interrupt is posted, its corresponding bit in the pending interrupt field is set.

This encoding of the pending priority and pending interrupt fields permits the processor to first check for any pending interrupts with a priority greater than the current program and then determine the vector number of the interrupt with the highest priority.

8.1.4.3 Caching Portions of the Interrupt Table

The architecture allows all or part of the interrupt table to be cached internally to the processor. The purpose of caching these fields is to reduce interrupt latency by allowing the processor to access certain interrupt procedure pointers and the pending interrupt information without having to make external memory accesses. The i960 Rx I/O processor caches the following:

- The value of the highest priority posted in the pending priorities field.
- A predefined subset of interrupt procedure pointers (entries from the interrupt table).
- Pending interrupts received from external interrupt pins.

This caching mechanism is non-transparent; the processor may modify fields in a cached interrupt table without modifying the same fields in the interrupt table itself. Vector caching is described in section 8.5.2.1, Vector Caching Option (pg. 8-46).

8.1.5 Interrupt Stack And Interrupt Record

The interrupt stack can be located anywhere in the non-reserved address space. The processor obtains a pointer to the base of the stack during initialization. The interrupt stack has the same structure as the local procedure stack described in section 7.1.1, Local Registers and the Procedure Stack (pg. 7-2). As with the local stack, the interrupt stack grows from lower addresses to higher addresses.

The processor saves the state of an interrupted program, or an interrupted interrupt procedure, in a record on the interrupt stack. Figure 8-3 shows the structure of this interrupt record.

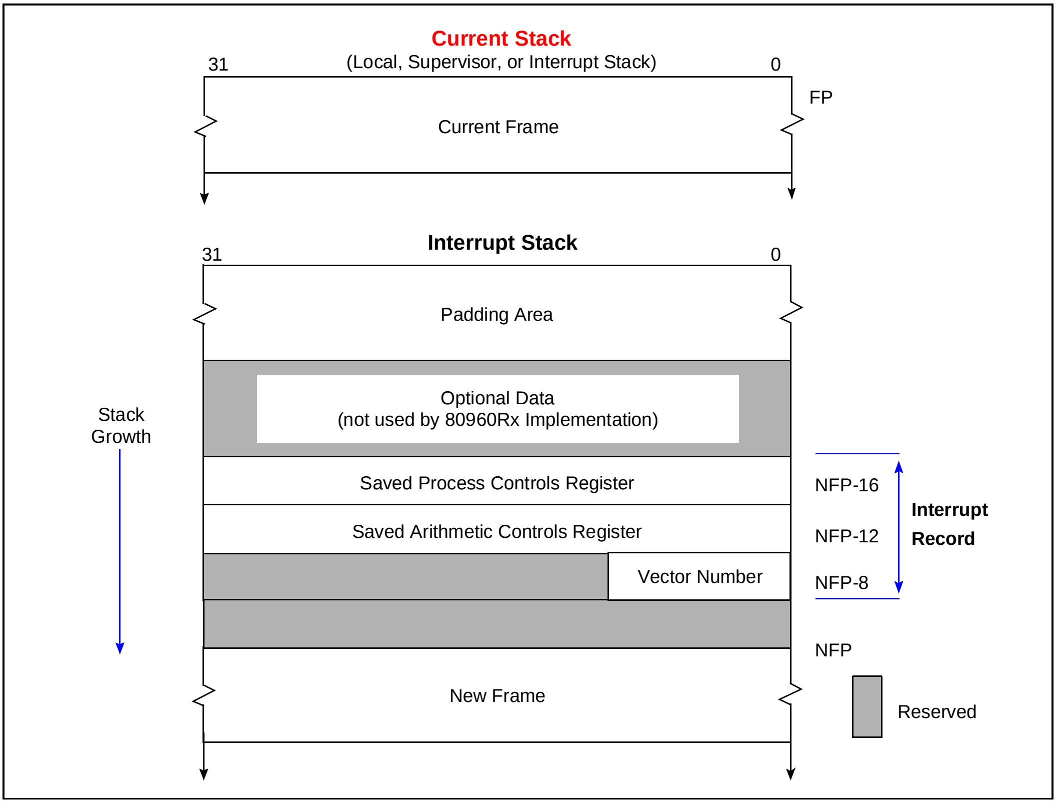


Figure 8-3. Storage of an Interrupt Record on the Interrupt Stack

The interrupt record is always stored on the interrupt stack adjacent to the new frame that is created for the interrupt handling procedure. It includes the state of the AC and PC registers at the time the interrupt was serviced and the interrupt procedure pointer number used. Relative to the new frame pointer (NFP), the saved AC register is located at address NFP-12, the saved PC register is located at address NFP-16.

In the i960 Rx I/O processor, the stack is aligned to a 16-byte boundary. When the processor needs to create a new frame on an interrupt call, it adds a padding area to the stack so that the new frame starts on a 16-byte boundary.

8.1.6 Posting Interrupts

Interrupts are posted to the processor by a number of different mechanisms; these are described in the following sections.

- Software interrupts: interrupts posted through the interrupt table, by software running on the i960 Rx I/O processor.
- External Interrupts: interrupts posted through the interrupt table, by an external agent to the i960 Rx I/O processor.
- Hardware interrupts: interrupts posted directly to the i960 Rx I/O processor through an implementation-dependent mechanism that may avoid using the interrupt table.

8.1.6.1 Posting Software Interrupts via sysctl

In the i960 Rx I/O processor, **sysctl** is typically used to request an interrupt in a program (see Example 8-1). The request interrupt message type (00H) is selected and the interrupt procedure pointer number is specified in the least significant byte of the instruction operand. See section 6.2.67, sysctl (pg. 6-114) for a complete discussion of **sysctl**.

Example 8-1. Using sysctl to Request an Interrupt

```
ldconst 0x53,g5# Vector number 53H is loaded
            # into byte 0 of register g5 and
            # the value is zero extended into
            # byte 1 of the register
sysctl g5, g5, g5# Vector number 53H is posted
```

A literal can be used to post an interrupt with a vector number from 8 to 31. Here, the required value of 00H in the second byte of a register operand is implied.

The action of the processor when it executes the **sysctl** instruction is as follows:

1. The processor performs an atomic write to the interrupt table and sets the bits in the pending-interrupts and pending-priorities fields that correspond to the requested interrupt.
2. The processor updates the internal software priority register with the value of the highest pending priority from the interrupt table. This may be the priority of the interrupt that was just posted.

The interrupt controller continuously compares the following three values: software priority register, current process priority, priority of the highest pending hardware-generated interrupt. When the software priority register value is the highest of the three, the following actions occur:

1. The interrupt controller signals the core that a software-generated interrupt is to be serviced.
2. The core checks the interrupt table in memory, determines the vector number of the highest priority pending interrupt and clears the pending-interrupts and pending-priorities bits in the table that correspond to that interrupt.
3. The core detects the interrupt with the next highest priority that is posted in the interrupt table (if any) and writes that value into the software priority register.
4. The core services the highest priority interrupt.

When more than one pending interrupt is posted in the interrupt table at the same interrupt priority, the core handles the interrupt with the highest vector number first. The software priority register is an internal register and, as such, is not visible to the user. The core only updates this register's value when **sysctl** requests an interrupt or when a software-generated interrupt is serviced.

8.1.6.2 Posting Software Interrupts Directly in the Interrupt Table

In special cases within a single processor system, software can post interrupts by setting the desired pending-interrupt and pending-priorities bits directly. Direct posting requires that software ensure that no external I/O agents post a pending interrupt simultaneously, and that an interrupt cannot occur after one bit is set but before the other is set. Note, however, that this method is not recommended.

8.1.6.3 Posting External Interrupts

An external agent posts (sets) a pending interrupt with vector “v” to the i960 Rx I/O processor through the interrupt table by executing the following algorithm:

```
External_Agent_Posting:

x = atomic_read(pending_priorities); #synchronize;
z = read(pending_interrupts[v/8]);
x[v/8] = 1;
z[v mod 8] = 1;
write(pending_interrupts[v/8]) = z;
atomic_write(pending_priorities) = x;
```

Generally, software cannot use this algorithm to post interrupts because there is no way for software to have an atomic (locking) read/write span multiple instructions.

8.1.6.4 Posting Hardware Interrupts

Certain interrupts are posted directly to the processor by an implementation-dependent mechanism that can bypass the interrupt table. This is often done for performance reasons.

8.1.7 Resolving Interrupt Priority

The interrupt controller continuously compares the processor's priority to the priorities of the highest-posted software interrupt and the highest-pending hardware interrupt. The core is interrupted when a pending interrupt request is higher than the processor priority or has a priority of 31. (Note that a priority-31 interrupt handler can be interrupted by another priority-31 interrupt.) There are no priority-0 interrupts, since such an interrupt would never have a priority higher than the current process, and would therefore never be serviced.

In the event that both hardware and software requested interrupts are posted at the same level, the hardware interrupt is delivered first while the software interrupt is left pending. As a result, when both priority-31 hardware- and software-requested interrupts are pending, control is first transferred to the interrupt handler for the hardware-requested interrupt. However, before the first instruction of that handler can be executed, the pending software-requested interrupt is delivered and control is transferred to the corresponding interrupt handler.

Example 8-2. Interrupt Resolution

```
/* Model used to resolve interrupts between execution of all macro instructions */
if (NMI#_pending && !block_NMI)
   { block_NMI = true;  /* Reset on return from NMI INTR handler */
     vecnum = 248; vector_addr = 0;
     PC.priority = 31;
     push_local_register_set();
     goto common_interrupt_process; }
if (ICON.gie == enabled) {
   expand_HW_int();
   temp = max(HW_Int_Priority, SW_Int_Priority);
   if (temp == 31 || temp > PC.priority)
      { PC.priority = temp;
        if (SW_Int_Priority > HW_Int_Priority) goto Deliver_SW_Int;
        else{ vecnum = HW_vecnum; goto Deliver_HW_Int;}
       }
    }
```

8.1.8 Sampling Pending Interrupts in the Interrupt Table

At specific points, the processor checks the interrupt table for pending interrupts posted. When one is found, it is handled as if the interrupt occurred at that time. In the i960 Rx I/O processor, a check for pending interrupts in the interrupt table is made when requesting a software interrupt with **sysctl** or when servicing a software interrupt.

When a check of the interrupt table is made, the following algorithm is used. Since the pending interrupts may be cached, the check for pending interrupt operation may not involve any memory operations. The algorithm uses synchronization because there may be multiple agents posting and unposting interrupts. In the algorithm, w, x, y, and z are temporary registers within the processor.

```
Check_For_Pending_Interrupts:

x = read(pending_priorities);
if(x == 0) return(); #nothing to do
y = most_significant_bit(x);
if(y != 31 && y <= current_priority) return();
x = atomic_read(pending_priorities); #synchronize
if(x == 0)
    {atomic_write(pending_priorities) = x;
     return();} #interrupts disappeared
        # (e.g., handled by another processor)
y = most_significant_bit(x); #must be repeated
if(y != 31 && y <= current_priority)
    {atomic_write(pending_priorities) = x;
    return();} #interrupt disappeared
z = read(pending_interrupts[y]); #z is a byte
if(z == 0)
    {x[y] = 0; #false alarm, should not happen
    atomic_write(pending_priorities) = x;
    return();}
else
    {w = most_significant_bit[z];
    z[w] = 0;
    write(pending_interrupts[y]) = z;
    if(z == 0) x[y] = 0; #no others at this level
    atomic_write(pending_priorities) = x;
    take_interrupt();}
```

The algorithm shows that the pending interrupts are marked by a bit in the Pending Interrupts Field, and that the Pending Priorities Field is an optimization. The processor examines Pending Interrupts only when the corresponding bit in Pending Priorities is set.

The steps prior to the `atomic_read` are another optimization. Note that these steps must be repeated within the synchronized critical section, since another processor could have spotted and accepted the same pending interrupt(s).

Use **sysctl** with a vector in the range 0 to 7 to force the core to check the interrupt table for pending interrupts. When an external agent is posting interrupts to a shared interrupt table, use **sysctl** periodically to guarantee recognition of pending interrupts posted in the table by the external agent.

8.1.9 Saving the Interrupt Mask

Whenever an interrupt requested by the external interrupt pins or by the internal timers is serviced, the IMSK register is automatically saved in register r3 of the new local register set allocated for the interrupt handler. After the mask is saved, the IMSK register is optionally cleared. This masks all interrupts except NMI#s while an interrupt is serviced. Since the IMSK register value is saved, the interrupt procedure can restore the value before returning. The option of clearing the mask is selected by programming the ICON register as described in section 8.4.2, Interrupt Control Register – ICON (pg. 8-34).

Priority-31 interrupts are interrupted by other priority-31 interrupts. For level-activated interrupt inputs, instructions within the interrupt handler are typically responsible for causing the source to deactivate. If these priority-31 interrupts are not masked, another priority-31 interrupt is signaled and serviced before the handler can deactivate the source. The first instruction of the interrupt handling procedure is never reached, unless the option is selected to clear the IMSK register on entry to the interrupt.

Another use of the mask is to lock out other interrupts when executing time-critical portions of an interrupt handling procedure. All hardware-generated interrupts are masked until software explicitly replaces the mask.

The processor does not restore r3 to the IMSK register when the interrupt return is executed. When the IMSK register is cleared, the interrupt handler must restore the IMSK register to enable interrupts after return from the handler.

8.2 THE i960® CORE PROCESSOR INTERRUPT CONTROLLER

The i960 Rx I/O processor Interrupt Controller Unit (ICU) provides a flexible, low-latency means for requesting and posting interrupts and minimizing the core's interrupt handling burden. Acting independently from the core, the interrupt controller posts interrupts requested by hardware and software sources and compares the priorities of posted interrupts with the current process priority.

The interrupt controller provides the following features for managing hardware-requested interrupts:

- Low latency, high throughput handling.
- Eight external interrupt pins.
- One non-maskable interrupt pin.
- Two internal timers sources.
- Peripheral interrupt sources.

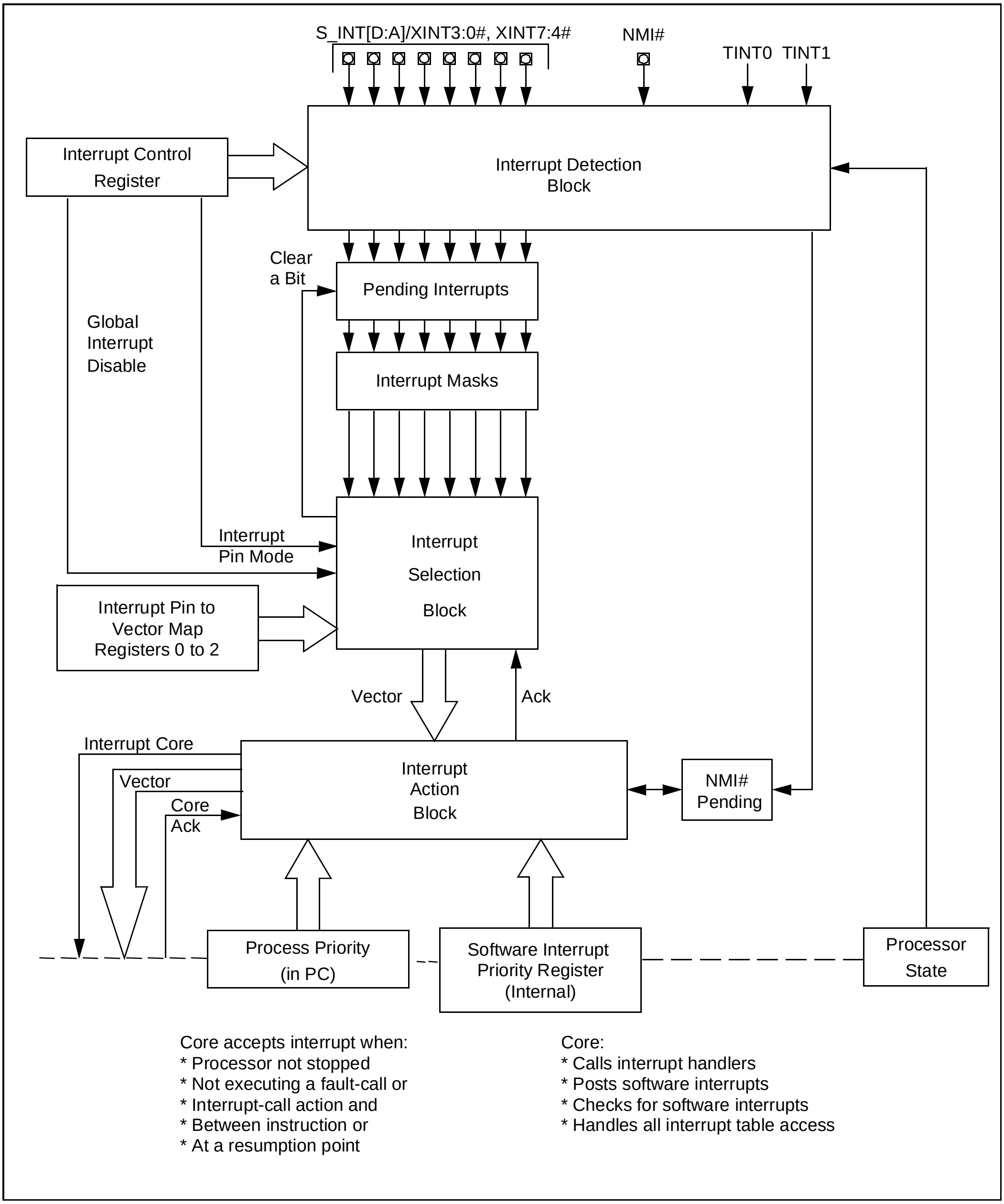

S_INT[D:A]/XINT3:0#, XINT7:4#
NMI#
TINT0
TINT1
Interrupt Control Register
Interrupt Detection Block
Clear a Bit
Pending Interrupts
Global Interrupt Disable
Interrupt Masks
Interrupt Pin Mode
Interrupt Selection Block
Interrupt Pin to Vector Map Registers 0 to 2
Vector
Ack
Interrupt Core
Vector
Core Ack
Interrupt Action Block
NMI# Pending
Process Priority (in PC)
Software Interrupt Priority Register (Internal)
Processor State
Core accepts interrupt when:
* Processor not stopped
* Not executing a fault-call or
* Interrupt-call action and
* Between instruction or
* At a resumption point
Core:
* Calls interrupt handlers
* Posts software interrupts
* Checks for software interrupts
* Handles all interrupt table access

Figure 8-4. Interrupt Controller

The user program interfaces to the interrupt controller with ten memory-mapped control registers. The Interrupt Control Register (ICON) and Interrupt Map Control Registers (IMAP0-IMAP2) provide configuration information. The Interrupt Pending Register (IPND) posts hardware-requested interrupts. The Interrupt Mask Register (IMSK) selectively masks hardware-requested interrupts.

8.2.1 Interrupt Controller Dedicated Mode

The 80960Rx interrupt controller external pins are set up for dedicated mode operation, where each external interrupt pin is assigned a vector number. Vector numbers that may be assigned to a pin are those with the encoding PPPP 0010_2 (Figure 8-5), where bits marked P are programmed with bits in the interrupt map (IMAP) registers. This encoding of programmable bits and preset bits can designate 15 unique vector numbers, each with a unique, even-numbered priority. (Vector 0000 0010_2 is undefined; it has a priority of 0.)

Interrupts are posted in the interrupt pending (IPND) register. Single bits in the IPND register correspond to each of the eight dedicated external interrupt inputs, or the two timer inputs to the interrupt controller. The interrupt mask (IMSK) register selectively masks each of the interrupts. Optionally, the IMSK register can be saved and cleared when an interrupt is serviced. This locks out other hardware-generated interrupts until the mask is restored. See section 8.4, MEMORY-MAPPED CONTROL REGISTERS (pg. 8-31) for a further description of the IMSK, IPND and IMAP registers.

Interrupt vectors are assigned to timer inputs in the same way external pins are assigned vectors.

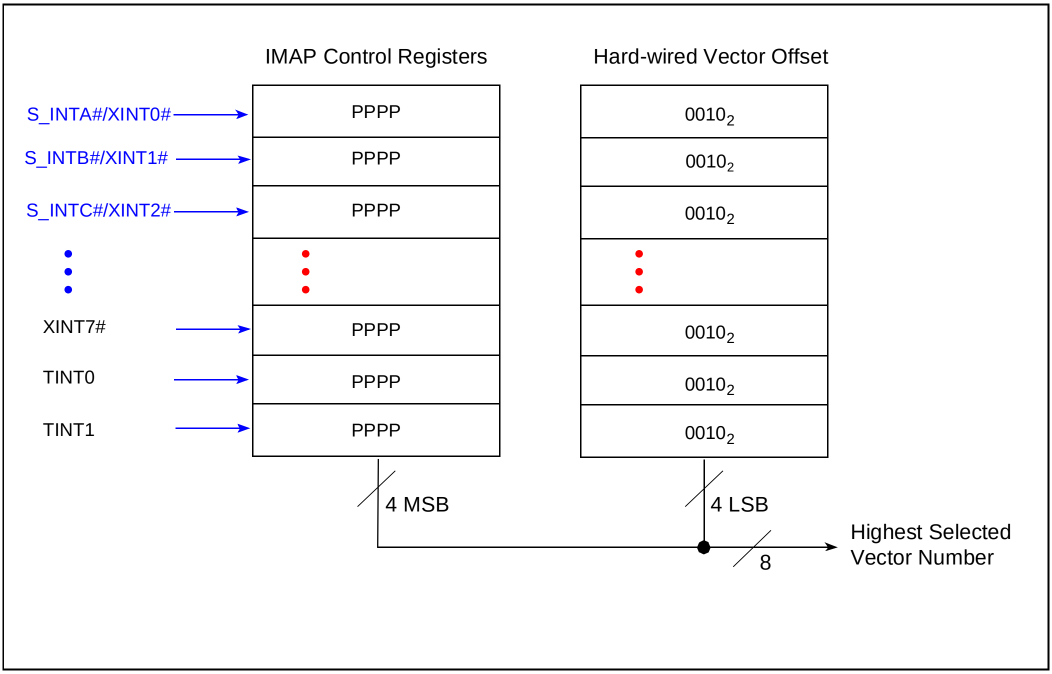


Figure 8-5. Interrupt Pin Vector Assignment

8.2.2 Interrupt Detection

The XINT7:0# pins use level-low detection. All of the interrupt pins use fast sampling.

For low-level detection, the pin's bit in the IPND register remains set as long as the pin is asserted (low). The processor attempts to clear the IPND bit on entry into the interrupt handler. However, if the active level on the pin is not removed at this time, the bit in the IPND register remains set until the source of the interrupt is deactivated and the IPND bit is explicitly cleared by software. Software may attempt to clear an interrupt pending bit before the active level on the corresponding pin is removed. In this case, the active level on the interrupt pin causes the pending bit to remain asserted.

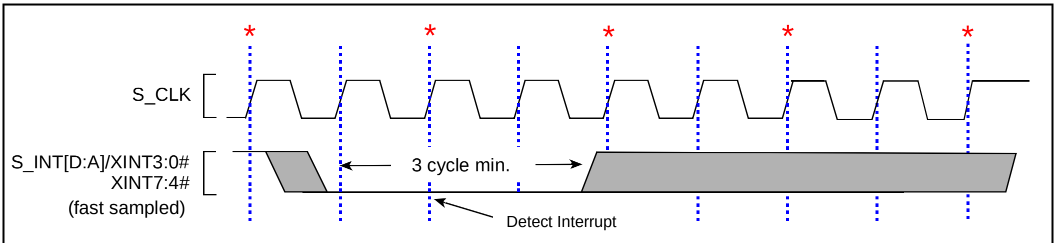
After the interrupt signal is deasserted, the handler then clears the interrupt pending bit for that source before return from handler is executed. If the pending bit is not cleared, the interrupt is re-entered after the return is executed.

Example 8-3 demonstrates how a level detect interrupt is typically handled. The example assumes that the **ld** from address "timer_0," deactivates the interrupt input.

Example 8-3. Return from a Level-detect Interrupt

```
# Clear level-detect interrupts before return from handler
  lda  IPND_MMR, g1  # Get address of IPND Memory-Mapped Register
  ld   timer_0, g0   # Get timer value and clear TMRO
  lda  0x1000, g2
wait:
  mov   0, g3
  atmod g1, g2, g3
  bbs   0xC, g3, wait
  ret                # Return from handler
```

Interrupt pins are asynchronous inputs. Setup or hold times relative to S_CLK are not needed to ensure proper pin detection. Note in Figure 8-6, which shows how a signal is sampled using fast sampling, that interrupt inputs are sampled once every two S_CLK cycles. For practical purposes, this means that asynchronous interrupting devices must generate an interrupt signal that is asserted for at least three S_CLK cycles. See your 80960Rx Data Sheet for setup and hold specifications that guarantee detection of the interrupt on particular edges of S_CLK. These specifications are useful in designs that use synchronous logic to generate interrupt signals to the processor. These specification must also be used to calculate the minimum signal width, as shown in Figure 8-6.



* Denotes sampling clock edge. Interrupt pins are sampled one time for every two S_CLK (external bus clock) cycles.

Figure 8-6. Interrupt Fast Sampling

8.2.3 Non-Maskable Interrupt (NMI#)

The NMI# pin generates an interrupt for implementation of critical interrupt routines. Error interrupts from the internal peripheral units also come into the i960 core through the NMI# pin. NMI# provides an interrupt that cannot be masked and that has a priority of 31. The interrupt vector for NMI# resides in the interrupt table as vector number 248. During initialization, the core caches the vector for NMI# on-chip, to reduce NMI# latency. The NMI# vector is cached in location 0H of internal data RAM.

The core immediately services NMI# requests. While servicing an NMI#, the core does not respond to any other interrupt requests, even another NMI# request. The processor remains in this non-interruptible state until any return-from-interrupt (in supervisor mode) occurs. An interrupt request on the NMI# pin is always falling-edge detected. (Note that a return-from-interrupt in user mode does not unblock NMI# events and should be avoided by software.)

8.2.4 Timer Interrupts

Each of the two timer units has an associated interrupt to allow the application to accept or post the interrupt request. The timer interrupts are connected directly to the i960 Rx I/O processor interrupt controller and are posted in the IPND register. These interrupts are set up through the timer control registers described in CHAPTER 19, TIMERS.

8.2.5 Software Interrupts

The application program may use the **sysctl** instruction to request interrupt service. The vector that **sysctl** requests is serviced immediately or posted in the interrupt table's pending interrupts section, depending upon the current processor priority and the request's priority. The interrupt controller caches the priority of the highest priority interrupt posted in the interrupt table. The processor cannot request vector 248 (NMI#) as a software interrupt.

8.2.6 Interrupt Operation Sequence

The interrupt controller, microcode and core resources handle all stages of interrupt service. Interrupt service is handled in the following stages:

Requesting Interrupt — In the i960 Rx I/O processor, the programmable on-chip interrupt controller transparently manages all interrupt requests. Interrupts are generated by hardware (external events) or software (the application program). Hardware requests are signaled on the 8-bit external interrupt port (S_INT[D:A]/XINT3:0#, XINT7:4#), the non-maskable interrupt pin (NMI#) or the two timer channels. Software interrupts are signaled with the sysctl instruction with post-interrupt message type.

Posting Interrupts — When an interrupt is requested, the interrupt is either serviced immediately or saved for later service, depending on the interrupt's priority. Saving the interrupt for later service is referred to as posting. Once posted, an interrupt becomes a pending interrupt. Hardware and software interrupts are posted differently:

- Hardware interrupts are posted by setting the interrupt's assigned bit in the interrupt pending (IPND) memory mapped register
- Software interrupts are posted by setting the interrupt's assigned bit in the interrupt table's pending priorities and pending interrupts fields

Checking Pending Interrupts — The interrupt controller compares each pending interrupt's priority with the current process priority. When process priority changes, posted interrupts of higher priority are then serviced. Comparing the process priority to posted interrupt priority is handled differently for hardware and software interrupts. Each hardware interrupt is assigned a specific priority when the processor is configured. The priority of all posted hardware interrupts is continually compared to the current process priority. Software interrupts are posted in the interrupt table in external memory. The highest priority posted in this table is also saved in an on-chip software priority register; this register is continually compared to the current process priority.

Servicing Interrupts — When the process priority falls below that of any posted interrupt, the interrupt is serviced. The comparator signals the core to begin a microcode sequence to perform the interrupt context switch and branch to the first instruction of the interrupt routine.

Figure 8-4 illustrates interrupt controller function. For best performance, the interrupt flow for hardware interrupt sources is implemented entirely in hardware.

The comparator only signals the core when a posted interrupt is a higher priority than the process priority. Because the comparator function is implemented in hardware, microcode cycles are never consumed unless an interrupt is serviced.

8.2.7 Setting Up the Interrupt Controller

This section provides an example of setting up the interrupt controller. The following example describes how the interrupt controller can be dynamically configured after initialization.

Example 8-4 sets up the interrupt controller to fetch interrupt vectors from internal data RAM rather than external memory. Initially the IMSK register is masked to allow for setup. A value that selects vector caching is loaded into the ICON register and the IMSK is unmasked.

Example 8-4. Programming the Interrupt Controller for Vector Caching

```
# Example vector caching setup . . .
mov   0x0, g0
mov   0x00006000, g1
ld    IMSK, g3        # mask, IMSK MMR at 0xFF008504
st    g1,IMSK
st    g1,ICON
```

8.2.8 Interrupt Service Routines

An interrupt handling procedure performs a specific action that is associated with a particular interrupt procedure pointer. For example, one interrupt handler task might initiate a timer unit request. The interrupt handler procedures can be located anywhere in the non-reserved address space. Since instructions in the i960 Rx I/O processor architecture must be word-aligned, each procedure must begin on a word boundary.

When an interrupt handling procedure is called, the processor allocates a new frame on the interrupt stack and a set of local registers for the procedure. If not already in supervisor mode, the processor always switches to supervisor mode while an interrupt is handled. It also saves the states of the AC and PC registers for the interrupted program.

The interrupt procedure shares the remainder of the execution environment resources (namely the global registers and the address space) with the interrupted program. Thus, interrupt procedures must preserve and restore the state of any resources shared with a non-cooperating program. For example, an interrupt procedure that uses a global register that is not permanently allocated to it should save the register's contents before using the register and restore the contents before returning from the interrupt handler.

To reduce interrupt latency to critical interrupt routines, interrupt handlers may be locked into the instruction cache. See section 8.5.2.2, Caching Interrupt Routines and Reserving Register Frames (pg. 8-47) for a complete description.

8.2.9 Interrupt Context Switch

When the processor services an interrupt, it automatically saves the interrupted program state or interrupt procedure and calls the interrupt handling procedure associated with the new interrupt request. When the interrupt handler completes, the processor automatically restores the interrupted program state. The method used to service an interrupt depends on the processor state when the interrupt is received.

- An *executing-state* interrupt — When the processor is executing a background task and an interrupt request is posted, the interrupt context switch must change stacks to the interrupt stack.
- An *interrupted-state* interrupt — When the processor is already executing an interrupt handler, no stack switch is required since the interrupt stack is already in use.

The following subsections describe interrupt handling actions for executing-state and interrupted-state interrupts. In both cases, it is assumed that the interrupt priority is higher than that of the processor and thus is serviced immediately when the processor receives it.

8.2.9.1 Servicing An Interrupt From Executing State

When the processor receives an interrupt while in the executing state (i.e., executing a program, PC.s = 0), it performs the following actions to service the interrupt. This procedure is the same regardless of whether the processor is in user or supervisor mode when the interrupt occurs. The processor:

1. Switches to the interrupt stack (see Figure 8-3). The interrupt stack pointer becomes the new stack pointer for the processor.
2. Saves the current PC and AC in an interrupt record on the interrupt stack. The processor also saves the interrupt procedure pointer number.
3. Allocates a new frame on the interrupt stack and loads the new frame pointer (NFP) in global register g15.
4. Sets the state flag in PC to interrupted (PC.s = 1), its execution mode to supervisor and its priority to the priority of the interrupt. Setting the processor's priority to that of the interrupt ensures that lower priority interrupts cannot interrupt the servicing of the current interrupt.
5. Clears the trace enable bit in PC. The interrupt is handled without raising trace faults.
6. Sets the frame return status field pfp[2:0] to 111_2.
7. Performs a call operation as described in CHAPTER 7, PROCEDURE CALLS. The address for the called procedure is specified in the interrupt table for the specified interrupt procedure pointer.

After completing the interrupt procedure, the processor:

1. Copies the arithmetic controls field and the process controls field from the interrupt record into the AC and PC, respectively. It therefore switches to the executing state and restores the trace-enable bit to its value before the interrupt occurred.
2. Deallocates the current stack frame and interrupt record from the interrupt stack and switches to the stack it was using before servicing the interrupt.
3. Performs a return operation as described in CHAPTER 7, PROCEDURE CALLS.
4. Resumes work on the program when all pending interrupts and trace faults are serviced.

8.2.9.2 Servicing An Interrupt From Interrupted State

When the processor receives an interrupt while servicing another interrupt, and the new interrupt has a higher priority than one being serviced, the current interrupt-handler routine is interrupted. Here, the processor performs the same interrupt-servicing action as described in section 8.2.9.1 to save the state of the interrupted interrupt-handler routine. The interrupt record is saved on the top of the interrupt stack prior to the new frame that is created for use in servicing the new interrupt. See Figure 8-3.

On the return from the current interrupt handler to the previous interrupt handler, the processor de-allocates the current stack frame and interrupt record, and stays on the interrupt stack.

8.3 PCI AND PERIPHERAL INTERRUPTS

The PCI and peripheral portion of the interrupt controller has two functions:

- Internal Peripheral Interrupt Control
- PCI Interrupt Routing

The peripheral interrupt control mechanism consolidates a number of interrupt sources for a given internal peripheral into a single interrupt driven to the i960 core. In order to provide the executing software with the knowledge of interrupt source, there is a memory-mapped status register that describes the source of the interrupt. All of the internal peripheral interrupts are individually enabled from their respective peripheral control registers.

The PCI interrupt routing mechanism allows the host software (or 80960 software) to route secondary PCI interrupts to either the i960 core or the P_INTA#, P_INTB#, P_INTC#, and P_INTD# output pins. This routing mechanism is controlled through a memory-mapped register accessible from the primary PCI bridge configuration space or the i960 Rx I/O processor local bus.

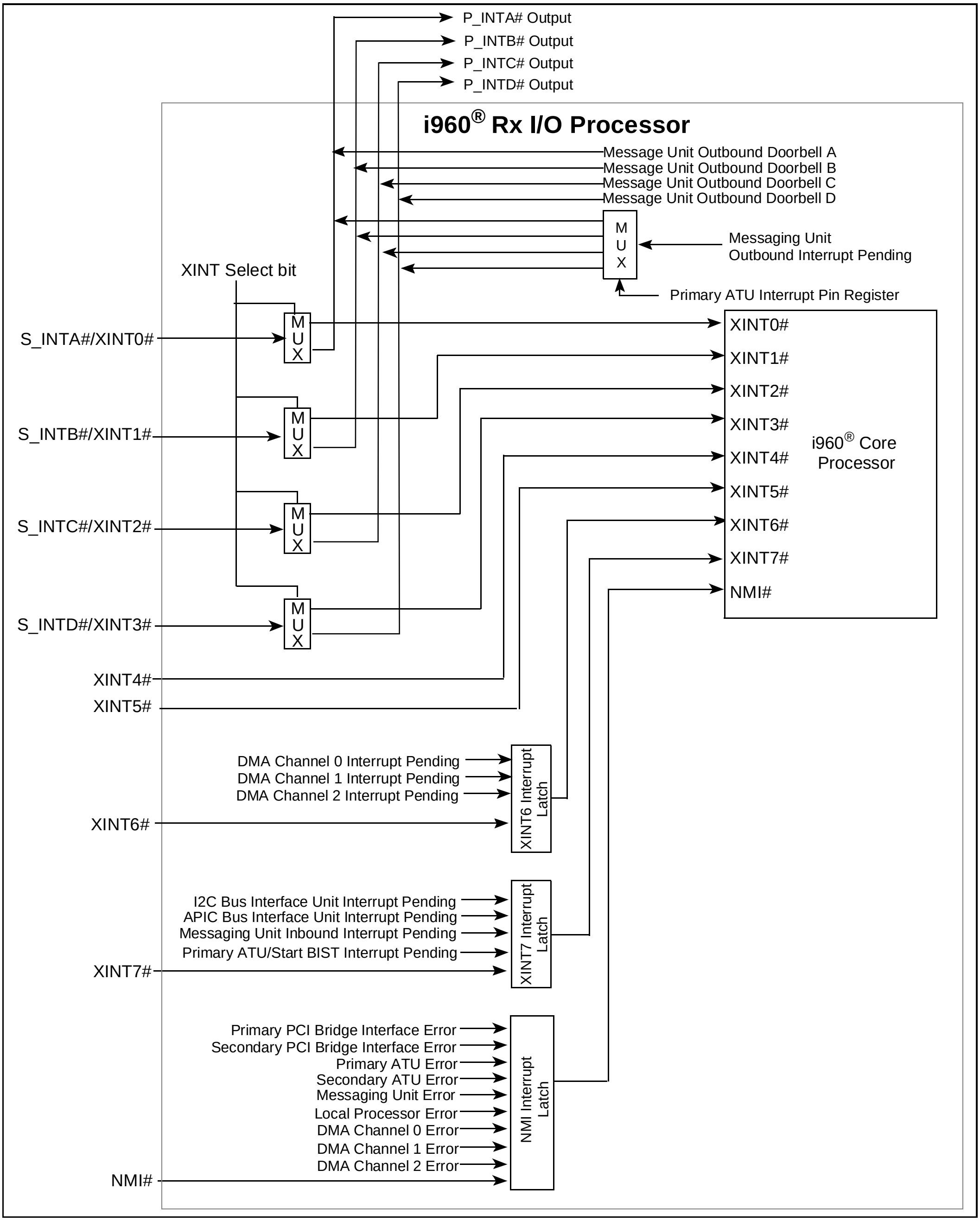

P_INTA# Output
P_INTB# Output
P_INTC# Output
P_INTD# Output
i960® Rx I/O Processor
Message Unit Outbound Doorbell A
Message Unit Outbound Doorbell B
Message Unit Outbound Doorbell C
Message Unit Outbound Doorbell D
MUX
Messaging Unit
Outbound Interrupt Pending
Primary ATU Interrupt Pin Register
XINT Select bit
S_INTA#/XINT0#
S_INTB#/XINT1#
S_INTC#/XINT2#
S_INTD#/XINT3#
XINT4#
XINT5#
XINT0#
XINT1#
XINT2#
XINT3#
XINT4#
XINT5#
XINT6#
XINT7#
NMI#
i960® Core Processor
DMA Channel 0 Interrupt Pending
DMA Channel 1 Interrupt Pending
DMA Channel 2 Interrupt Pending
XINT6#
XINT6 Interrupt Latch
I2C Bus Interface Unit Interrupt Pending
APIC Bus Interface Unit Interrupt Pending
Messaging Unit Inbound Interrupt Pending
Primary ATU/Start BIST Interrupt Pending
XINT7#
XINT7 Interrupt Latch
Primary PCI Bridge Interface Error
Secondary PCI Bridge Interface Error
Primary ATU Error
Secondary ATU Error
Messaging Unit Error
Local Processor Error
DMA Channel 0 Error
DMA Channel 1 Error
DMA Channel 2 Error
NMI#
NMI Interrupt Latch

Figure 8-7. Interrupt Controller Connections for 80960RP 33/5.0 Volt

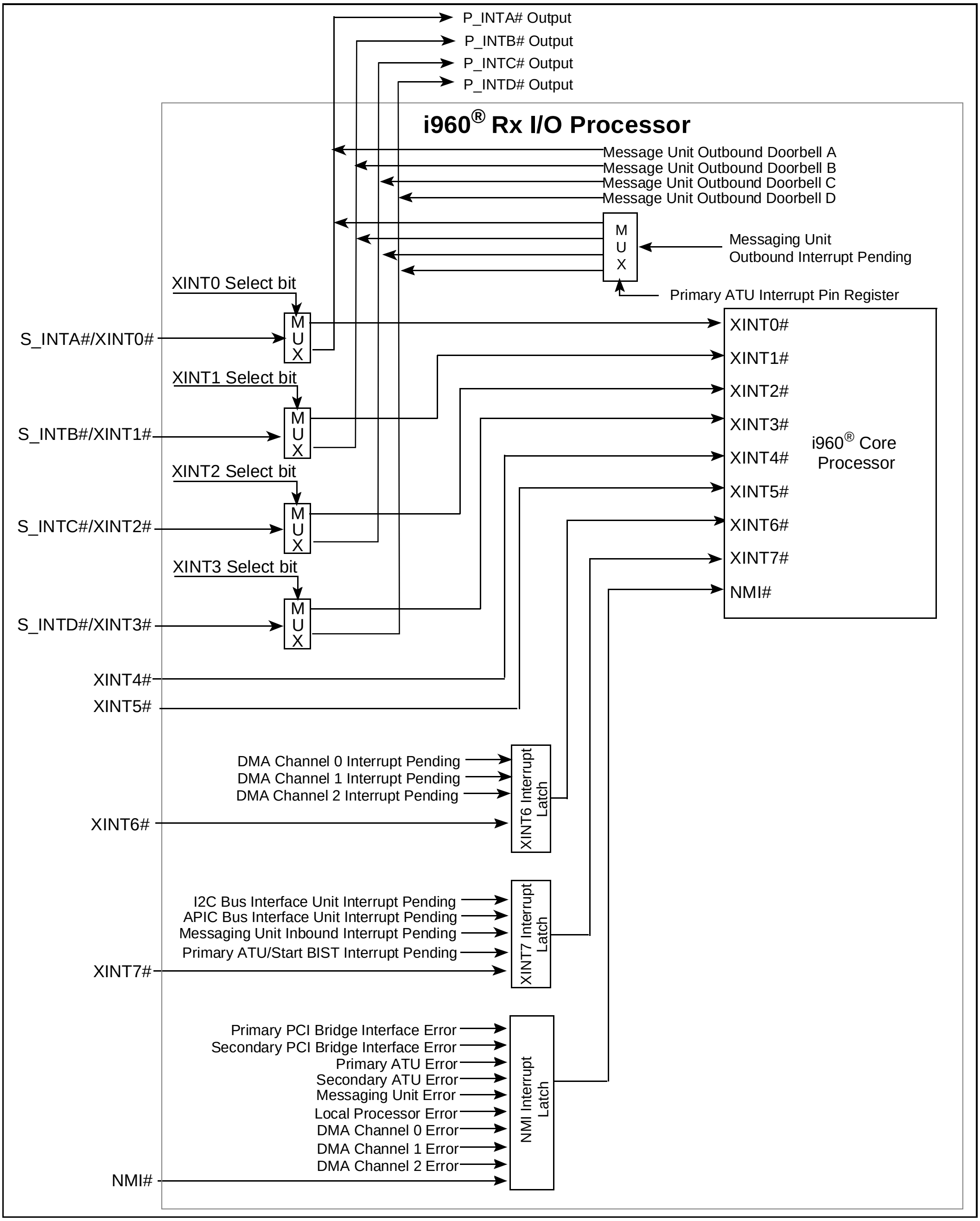

P_INTA# Output
P_INTB# Output
P_INTC# Output
P_INTD# Output
i960® Rx I/O Processor
Message Unit Outbound Doorbell A
Message Unit Outbound Doorbell B
Message Unit Outbound Doorbell C
Message Unit Outbound Doorbell D
MUX
Messaging Unit Outbound Interrupt Pending
Primary ATU Interrupt Pin Register
XINT0 Select bit
S_INTA#/XINT0#
MUX
XINT1 Select bit
S_INTB#/XINT1#
MUX
XINT2 Select bit
S_INTC#/XINT2#
MUX
XINT3 Select bit
S_INTD#/XINT3#
MUX
XINT0#
XINT1#
XINT2#
XINT3#
XINT4#
XINT5#
XINT6#
XINT7#
NMI#
i960® Core Processor
XINT4#
XINT5#
DMA Channel 0 Interrupt Pending
DMA Channel 1 Interrupt Pending
DMA Channel 2 Interrupt Pending
XINT6#
XINT6 Interrupt Latch
I2C Bus Interface Unit Interrupt Pending
APIC Bus Interface Unit Interrupt Pending
Messaging Unit Inbound Interrupt Pending
Primary ATU/Start BIST Interrupt Pending
XINT7#
XINT7 Interrupt Latch
Primary PCI Bridge Interface Error
Secondary PCI Bridge Interface Error
Primary ATU Error
Secondary ATU Error
Messaging Unit Error
Local Processor Error
DMA Channel 0 Error
DMA Channel 1 Error
DMA Channel 2 Error
NMI#
NMI Interrupt Latch

Figure 8-8. Interrupt Controller Connections for 80960Rx 33/3.3 Volt

8.3.1 Pin Descriptions

The i960 Rx I/O processor provides eight external interrupt pins and one non-maskable interrupt pin for detecting external interrupt requests. The eight external pins are configured as dedicated inputs, where each pin is capable of requesting a single interrupt, in some cases from several different sources. The external interrupt input interface for the i960 Rx I/O processor consists of the following pins:

Table 8-1. Interrupt Input Pin Descriptions

Signal	Description
S_INTA#/XINT0#	Can be directed to the P_INTA# output or the i960 core interrupt input XINT0#. When routed to the P_INTA# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT0#, this input is not shared.
S_INTB#/XINT1#	Can be directed to the P_INTB# output or the i960 core interrupt input XINT1#. When routed to the P_INTB1# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT1?#, this input is not shared.
S_INTC#/XINT2#	Can be directed to the P_INTC# output or the i960 core interrupt input XINT2#. When routed to the P_INTC2# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT2?#, this input is not shared.
S_INTD#/XINT3#	Can be directed to the P_INTD# output or the i960 core interrupt input XINT3#. When routed to the P_INTD# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT3#, this input is not shared.
XINT4#	Always connected to the i960 core interrupt input XINT4#.
XINT5#	Always connected to the i960 core interrupt input XINT5#.
XINT6#	Shared with three internal interrupts. They are the interrupts from each of the three internal DMA channels. All of the interrupts are directed to the i960 core interrupt input XINT6#. Software must read the XINT6 Interrupt Status Register to determine the exact source of the interrupt.
XINT7#	Shared with four internal interrupts. They are the interrupts from the APIC Bus Interface Unit, the I^{2}C Bus Interface Unit, the Primary ATU, and the Messaging Unit. All of the interrupts are directed to the i960 core interrupt input XINT7#. Software must read the XINT7 Interrupt Status Register to determine the exact source of the interrupt.
NMI#	Shared with eight internal interrupts. They include error interrupts from the local processor, primary PCI bridge interface, secondary PCI bridge interface, primary ATU, secondary ATU, and the three DMA channels. All of the interrupts are directed to the i960 core NMI# input. Software must read the NMI Interrupt Status Register to determine the exact source of the interrupt. NMI# is the highest priority interrupt recognized. This pin is synchronized internal to the i960 core.

All pins in Table 8-1 are level-low activated. See section 8.2.2, Interrupt Detection (pg. 8-16).

8.3.2 PCI Interrupt Routing

Four PCI interrupt inputs can be routed to either the i960 core interrupt inputs or to the PCI interrupt output pins. This routing is controlled by the XINT Select bit in the PCI interrupt Routing Select Register. See Table 8-3.

Table 8-2. PCI Interrupt Routing Summary for 80960RP 33/5.0 Volt

XINT Select Bit	Description
0	**S_INTA#/XINT0#** Input Pin routed to i960 core processor **XINT0#** Input Pin **S_INTB#/XINT1#** Input Pin routed to i960 core processor **XINT1#** Input Pin **S_INTC#/XINT2#** Input Pin routed to i960 core processor **XINT2#** Input Pin **S_INTD#/XINT3#** Input Pin routed to i960 core processor **XINT3#** Input Pin
1	**S_INTA#/XINT0#** Input Pin routed to **P_INTA#** Output Pin **S_INTB#/XINT1#** Input Pin routed to **P_INTB#** Output Pin **S_INTC#/XINT2#** Input Pin routed to **P_INTC#** Output Pin **S_INTD#/XINT3#** Input Pin routed to **P_INTD#** Output Pin

Table 8-3. PCI Interrupt Routing Summary for 80960RP 33/3.3 Volt

PIRSR Select Bit	Bit Value	Description
bit 0	0	**S_INTA#/XINT0#** Input Pin routed to i960 core processor **XINT0#** Input Pin
	1	**S_INTA#/XINT0#** Input Pin routed to **P_INTA#** Output Pin
bit 1	0	**S_INTB#/XINT1#** Input Pin routed to i960 core processor **XINT1#** Input Pin
	1	**S_INTB#/XINT1#** Input Pin routed to **P_INTB#** Output Pin
bit 2	0	**S_INTC#/XINT2#** Input Pin routed to i960 core processor **XINT2#** Input Pin
	1	**S_INTC#/XINT2#** Input Pin routed to **P_INTC#** Output Pin
bit 3	0	**S_INTD#/XINT3#** Input Pin routed to i960 core processor **XINT3#** Input Pin
	1	**S_INTD#/XINT3#** Input Pin routed to **P_INTD#** Output Pin

8.3.3 Internal Peripheral Interrupt Routing

XINT6#, XINT7# and NMI# interrupt inputs on the i960 core receive inputs from multiple internal interrupt sources. One internal latch before each of these three inputs provides the necessary muxing of the different interrupt sources. Application software can determine which peripheral unit caused an interrupt by reading the corresponding interrupt latch. More detail about the exact cause of the interrupt can be determined by reading status from the peripheral unit.

8.3.3.1 XINT6 Interrupt Sources

The XINT6# interrupt of the i960 core receives interrupts from the external pin and the three DMA channels. A DMA channel can cause an interrupt for a DMA End of Transfer interrupt or a DMA End of Chain interrupt. See section 20.3, DMA TRANSFER (pg. 20-4) for details. A valid interrupt from any of these sources sets the bit in the latch and outputs a level-sensitive interrupt to the i960 core's XINT6# input. The interrupt latch continues to drive an active low input to the i960 core interrupt input while an interrupt is present at the latch. The XINT6 interrupt latch is read through the XINT6 Interrupt Status Register. The XINT6 interrupt latch is cleared by clearing the source of the interrupt at the internal peripheral or deasserting the XINT6# input.

The interrupt sources which drive the inputs to the XINT6 interrupt latch are detailed in Table 8-4

Table 8-4. XINT6 Interrupt Sources

Unit	Interrupt Condition	Interrupt Status		Interrupt MASK	
		Register	Bit	Register	Bit
DMA Channel 0	End of Chain	CSR0	08	DCR0	04
	End of Transfer	CSR0	09		
DMA Channel 1	End of Chain	CSR1	08	DCR1	04
	End of Transfer	CSR1	09		
DMA Channel 2	End of Chain	CSR2	08	DCR2	04
	End of Transfer	CSR2	09		
XINT6# Pin	External Source	N/A	N/A	N/A	N/A

8.3.3.2 XINT7 Interrupt Sources

The XINT7# interrupt on the i960 core receives interrupts from the external pin, the APIC Bus Interface Unit, the I^{2}C Bus Interface Unit, the Primary ATU, and the Messaging Unit. A valid interrupt from any of these sources sets the bit in the latch and outputs a level-sensitive interrupt to the i960 core XINT7# input. The interrupt latch drives an active low input to the i960 core interrupt input as long as an interrupt is present at the latch. The XINT7 interrupt latch is read through the XINT7 Interrupt Status Register. The XINT7 interrupt latch is cleared by clearing the source of the interrupt at the internal peripheral or deasserting the XINT7# input pin.

The interrupt sources which drive the inputs to the XINT7 interrupt latch are detailed in Table 8-5

Table 8-5. XINT7 Interrupt Sources

Unit	Interrupt Condition	Interrupt Status		Interrupt MASK	
		Register	Bit	Register	Bit
APIC Bus Interface Unit	APIC Message Sent	APIC CSR	06	APIC CSR	05
	EOI Message Received	APIC CSR	14	APIC CSR	13
I^{2}C Bus Interface Unit	Slave STOP Detected	ISR	04	ICR	11
	Arbitration Loss Detected	ISR	05	ICR	12
	IDBR Transmit Empty	ISR	06	ICR	08
	IDBR Receive Full	ISR	07	ICR	09
	Slave Address Detected	ISR	09	ICR	13
	Bus Error	ISR	10	ICR	10
Messaging Unit	Inbound Message 0 Interrupt	IISR	00	IIMR	00
	Inbound Message 1 Interrupt	IISR	01	IIMR	01
	Inbound Doorbell Interrupt	IISR	02	IIMR	02
	Inbound Post Queue Interrupt	IISR	04	IIMR	04
	Index Register Interrupt	IISR	06	IIMR	06
	APIC Register Select Interrupt	IISR	07	IIMR	07
	APIC Window Interrupt	IISR	08	IIMR	08
Primary ATU	ATU BIST Start	PATUISR	08	N/A	N/A
XINT7# Pin	External Source	N/A	N/A	N/A	N/A

8.3.3.3 NMI Interrupt Sources

The Non-Maskable Interrupt (NMI#) on the i960 core receives interrupts from the external pin, the primary and secondary ATUs, the primary and secondary bridge interfaces, the i960 core and each of the three DMA channels. Each of the interrupts represents an error condition in the peripheral unit. Several of these conditions can be masked through the Secondary Decode Enable Register. A valid interrupt from any of these sources, when enabled, sets the bit in the latch and outputs an edge-triggered interrupt to the i960 core NMI# input. The NMI interrupt latch is read through the NMI Interrupt Status Register. The NMI interrupt latch is cleared by clearing the sources of all interrupts at the internal peripherals. A new edge triggered interrupt is generated to the i960 core only after all interrupt status bits have been simultaneously cleared.

The interrupt sources which drive the inputs to the NMI interrupt latch are detailed in Table 8-6

Table 8-6. NMI Interrupt Sources (Sheet 1 of 2)

Unit	Error Condition	Interrupt Status		Interrupt MASK	
		Register	Bit	Register	Bit
Primary PCI Bridge Interface	PCI Master Parity Error	PBISR	00	SDER	06
	PCI Target Abort (target)	PBISR	01	SDER	07
	PCI Target Abort (master)	PBISR	02	SDER	08
	PCI Master Abort	PBISR	03	SDER	09
	P_SERR# Asserted	PBISR	04	SDER	10
Secondary PCI Bridge Interface	PCI Master Parity Error	SBISR	00	SDER	11
	PCI Target Abort (target)	SBISR	01	SDER	12
	PCI Target Abort (master)	SBISR	02	SDER	13
	PCI Master Abort	SBISR	03	SDER	14
	S_SERR# Asserted	SBISR	04	SDER	15
Primary ATU	PCI Master Parity Error	PATUISR	00	ATUCR	04
	PCI Target Abort (target)	PATUISR	01	ATUCR	04
	PCI Target Abort (master)	PATUISR	02	ATUCR	04
	PCI Master Abort	PATUISR	03	ATUCR	04
	P_SERR# Asserted	PATUISR	04	ATUCR	04
	80960 Bus Fault	PATUISR	05	N/A	N/A
	80960 Memory Fault	PATUISR	06	N/A	N/A
Secondary ATU	PCI Master Parity Error	SATUISR	00	ATUCR	05
	PCI Target Abort (target)	SATUISR	01	ATUCR	05
	PCI Target Abort (master)	SATUISR	02	ATUCR	05
	PCI Master Abort	SATUISR	03	ATUCR	05
	S_SERR# Asserted	SATUISR	04	ATUCR	05
	80960 Bus Fault	SATUISR	05	N/A	N/A
	80960 Memory Fault	SATUISR	06	N/A	N/A
Messaging Unit	NMI Doorbell	IISR	03	IIMR	03
	Outbound Free Queue Overflow	IISR	05	IIMR	05
i960 Core Processor	80960 Local Bus Fault	LPISR	05	N/A	N/A
	80960 Memory Fault	LPISR	06	N/A	N/A
DMA Channel 0	PCI Master Parity Error	CSR0	0	PATUCMD	06
	PCI Target Abort (master)	CSR0	2	N/A	N/A
	PCI Master Abort	CSR0	3	N/A	N/A
	80960 Bus Fault	CSR0	5	N/A	N/A
	80960 Memory Fault	CSR0	6	N/A	N/A

Table 8-6. NMI Interrupt Sources (Sheet 2 of 2)

Unit	Error Condition	Interrupt Status		Interrupt MASK	
		Register	Bit	Register	Bit
DMA Channel 1	PCI Master Parity Error	CSR1	0	PATUCMD	06
	PCI Target Abort (master)	CSR1	2	N/A	N/A
	PCI Master Abort	CSR1	3	N/A	N/A
	80960 Bus Fault	CSR1	5	N/A	N/A
	80960 Memory Fault	CSR1	6	N/A	N/A
DMA Channel 2	PCI Master Parity Error	CSR2	0	SATUCMD	06
	PCI Target Abort (master)	CSR2	2	SATUCMD	06
	PCI Master Abort	CSR2	3	SATUCMD	06
	80960 Bus Fault	CSR2	5	SATUCMD	06
	80960 Memory Fault	CSR2	6	SATUCMD	06
NMI# Pin	External Source	N/A	N/A	N/A	N/A

8.3.4 PCI Outbound Doorbell Interrupts

The i960 Rx I/O processor has the capability of generating interrupts on any of the four primary PCI interrupt pins. This is done by setting a bit in the messaging unit Outbound Doorbell Port Register. See CHAPTER 17, MESSAGING UNIT for details.

8.4 MEMORY-MAPPED CONTROL REGISTERS

The programmer's interface to the interrupt controller is through eleven memory-mapped control registers. Table 8-7 describes these registers.

Table 8-7. Interrupt Control Registers Memory-Mapped Addresses

Register Name	Description	Address
PIRSR	PCI Interrupt Routing Select Register	0000 1050H
SDER	Secondary Decode Enable Register	0000 105CH
NISR	NMI Interrupt Status Register	0000 1700H
XINT7	XINT7 Interrupt Status Register	0000 1704H
XINT6	XINT6 Interrupt Status Register	0000 1708H
IPND	Interrupt Pending Register	FF00 8500H
IMSK	Interrupt Mask Register	FF00 8504H
ICON	Interrupt Control Register	FF00 8510H
IMAP0	Interrupt Map Register 0	FF00 8520H
IMAP1	Interrupt Map Register 1	FF00 8524H
IMAP2	Interrupt Map Register 2	FF00 8528H

All registers are visible to software as 80960Rx memory-mapped registers and can be accessed through the internal memory bus. The PCI Interrupt Routing Select Register and the Secondary Decode Enable Register are accessible from the internal memory bus and through the PCI configuration register space of the PCI-to-PCI Bridge Unit (function #0). See CHAPTER 15, PCI-TO-PCI BRIDGE UNIT for additional information regarding the PCI configuration cycles that can access these registers.

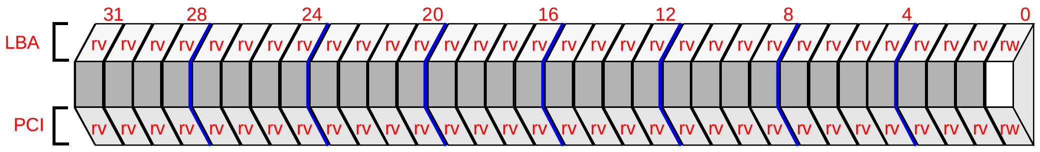
8.4.1 PCI Interrupt Routing Select Register (PIRSR)

The PCI Interrupt Routing Select Register (PIRSR) determines the routing of four of the external interrupt pins. These interrupt pins consist of four secondary PCI interrupt inputs which are routed to either the primary PCI interrupts or the i960 core interrupts.

If the secondary PCI interrupt inputs are routed to the primary PCI interrupt pins, the i960 core XINT3:0# inputs must be set inactive by setting bits 3-0 in the IMSK register to zero.

Table 8-8 and Table 8-9 show the bit definitions for programming the PCI Interrupt Routing Select Register. The XINT Select bit defaults to a 0.

Table 8-8. PCI Interrupt Routing Select Register – PIRSR (80960RP 33/5.0 Volt)

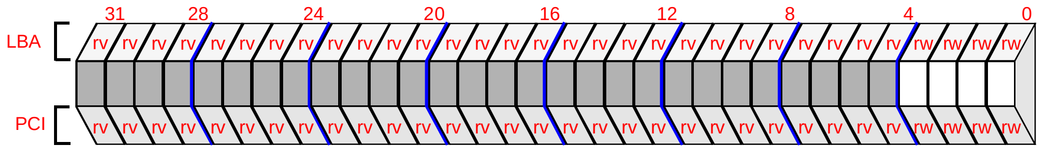


LBA: 1050H
PCI: 50H

Legend:
NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:01	0000 0000H	Reserved. Initialize to 0.
00	0_2	XINT Select Bit - (1) Interrupts Routed To P_INTx# Pins (0) Interrupts Routed To i960 core Interrupt Controller Input

Table 8-9. PCI Interrupt Routing Select Register – PIRSR (80960Rx 33/3.3 Volt)


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw

LBA: 1050H
PCI: 50H

Legend:	NA = Not Accessible	RO = Read Only
RV = Reserved	PR = Preserved	RW = Read/Write
RS = Read/Set	RC = Read Clear	

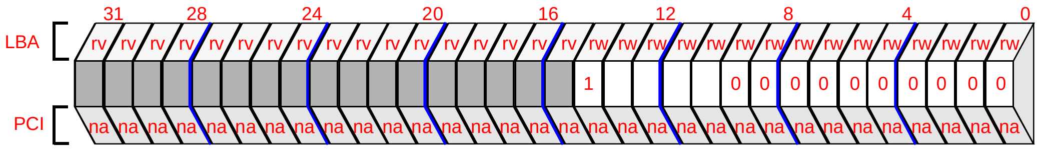
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:04	0000 000H	Reserved. Initialize to 0.
03	0_2	XINT3 Select Bit - (1) Interrupts Routed To P_INTx# Pins (0) Interrupts Routed To i960 core Interrupt Controller Input
02	0_2	XINT2 Select Bit - (1) Interrupts Routed To P_INTx# Pins (0) Interrupts Routed To i960 core Interrupt Controller Input
01	0_2	XINT1 Select Bit - (1) Interrupts Routed To P_INTx# Pins (0) Interrupts Routed To i960 core Interrupt Controller Input
00	0_2	XINT0 Select Bit - (1) Interrupts Routed To P_INTx# Pins (0) Interrupts Routed To i960 core Interrupt Controller Input

8.4.2 Interrupt Control Register – ICON

The ICON register is a 32-bit memory-mapped control register, that sets up the interrupt controller. Software can manipulate this register using the load/store type instructions. The ICON register is also automatically loaded at initialization from the control table in external memory. Table 8-10 describes the layout of the ICON register.

Table 8-10. Interrupt Control Register – ICON


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
1 0 0 0 0 0 0 0 0 0 0
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8510H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:15	Default Value Loaded from Image in Control Table.	Reserved. Initialize to 0.
14		This bit must be set (1).
13		Vector Cache Enable - determines whether interrupt table vector entries are fetched from the interrupt table (bit clear) or from internal data RAM (bit set). Only vectors with the four least-significant bits equal to 0010_2 may be cached in internal data RAM.
12:11		Mask Operation Field - determines the operation the core performs on the mask register when a hardware-generated interrupt is serviced. On an interrupt, the value in IMSK is copied to r3. IMSK is then either left unchanged (00) or cleared (01). IMSK is never cleared for NMI# or software interrupts.
10		Global Interrupts Disable - when set (1) this bit globally disables the i960 core interrupt inputs and the timer unit inputs. When clear (0) this bit globally enables the i960 core interrupt inputs and the timer unit inputs. This does not affect the NMI# input. This bit performs the same function as clearing the IMSK register. This bit is also changed indirectly by the instructions **inten**, **intdis**, **intctl**.
9:0		These bits must be cleared (0).

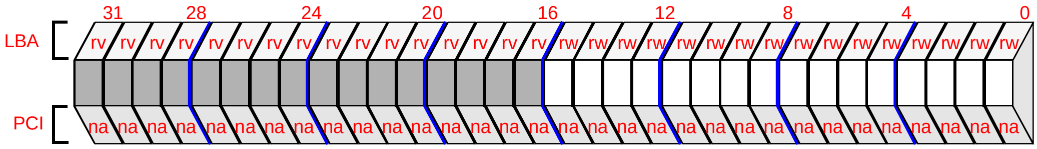
8.4.3 Interrupt Mapping Registers – IMAP0-IMAP2

The IMAP registers (Table 8-11 through Table 8-13) are three 32-bit registers (IMAP0 through IMAP2). These registers are used to program the vector number associated with the interrupt source. IMAP0 and IMAP1 contain mapping information for the external interrupt pins (four bits per pin). IMAP2 contains mapping information for the timer-interrupt inputs (four bits per interrupt).

Each set of four bits contains a vector number's four most-significant bits; the four least-significant bits are always 0010_2. In other words, each source can be programmed for a vector number of PPPP 0010_2, where "P" indicates a programmable bit. For example, IMAP0 bits 4 through 7 contain mapping information for the XINT1 pin. When these bits are set to 0110_2, the pin is mapped to vector number 0110 0010_2 (or vector number 98).

Software can access the mapping registers using load/store type instructions. The mapping registers are also automatically loaded at initialization from the control table in external memory.

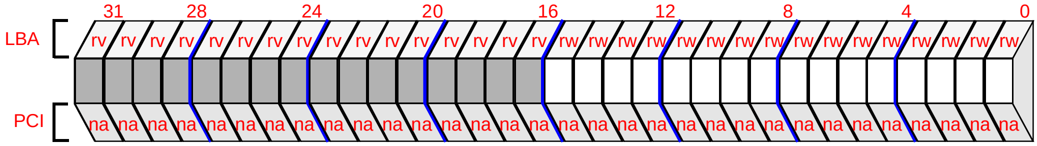
Table 8-11. Interrupt Map Register 0 – IMAP0


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8520H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:15	Default Value Loaded from Image in Control Table.	Reserved. Initialize to 0.
15:12		External Interrupt 3 Field.
11:08		External Interrupt 2 Field.
07:04		External Interrupt 1 Field.
03:00		External Interrupt 0 Field.

Table 8-12. Interrupt Map Register 1 – IMAP1

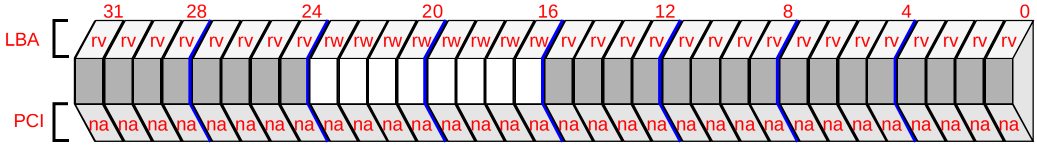

31 28 24 20 16 12 8 4 0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8524H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:15	Default Value Loaded from Image in Control Table.	Reserved. Initialize to 0.
15:12		External Interrupt 7 Field.
11:08		External Interrupt 6 Field.
07:04		External Interrupt 5 Field.
03:00		External Interrupt 4 Field.

Table 8-13. Interrupt Map Register 2 – IMAP2


31 28 24 20 16 12 8 4 0
LBA
rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8528H
PCI: NA

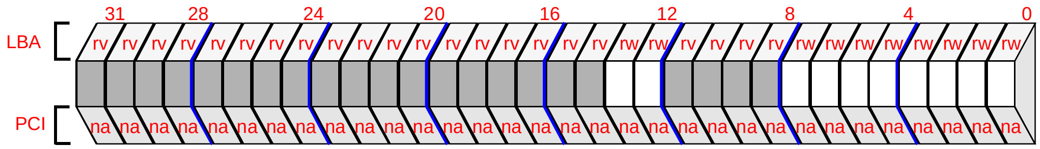
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:24	Default Value Loaded from Image in Control Table.	Reserved. Initialize to 0.
23:20		Timer Interrupt 1 Field.
19:16		Timer Interrupt 0 Field.
15:00		Reserved. Initialize to 0.

8.4.4 Interrupt Mask – IMSK and Interrupt Pending Registers – IPND

The IMSK and IPND registers are both memory-mapped registers. Bits 0 through 7 of these registers are associated with the external interrupt pins (XINT0# - XINT7#) and bits 12 and 13 are associated with the timer-interrupt inputs (TMR0 and TMR1). All other bits are reserved and should be cleared at initialization.

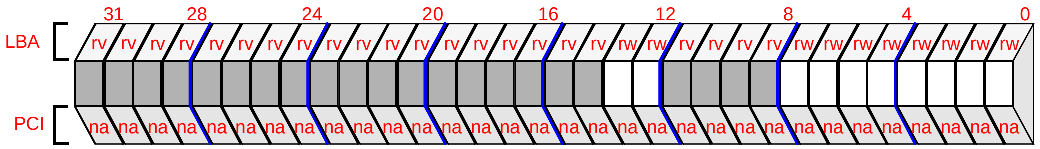
Table 8-14. Interrupt Pending Register – IPND


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rv rv rv rv rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8500H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:14	XXXX XH	Reserved. Initialize to 0.
13:12	XX_2	Timer Interrupt Pending Bits - IPND.tip (1) Pending Interrupt (0) No Interrupt
11:08	XH	Reserved. Initialize to 0.
07:00	XXH	External Interrupt Pending Bits - IPND.xip (1) Pending Interrupt (0) No Interrupt

Table 8-15. Interrupt Mask Register – IMSK


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rv rv rv rv rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8504H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:14	0000 0H	Reserved. Initialize to 0.
13:12	00_2	Timer Interrupt Mask Bits - IMSK.tim (1) Not Masked (0) Masked
11:08	0H	Reserved. Initialize to 0.
07:00	00H	External Interrupt Mask Bits - IMSK.xim (1) Not Masked (0) Masked

The IPND register posts interrupts originating from the eight external dedicated sources and the two timer sources. Asserting one of these inputs latches a 1 into its associated bit in the IPND register. The mask register provides a mechanism for masking individual bits in the IPND register. An interrupt source is disabled when its associated mask bit is cleared (0).

When delivering a hardware interrupt, the interrupt controller conditionally clears IMSK based on the value of the ICON.mo bit. Note that IMSK is never cleared for NMI# or software interrupt.

Although software can read and write IPND and IMSK using any memory-format instruction, it is recommended that a read-modify-write operation using the atomic-modify instruction (**atmod**) be used for reading and writing these registers. Executing an **atmod** on one of these registers causes the interrupt controller to perform regular interrupt processing (including using or automatically updating IPND and IMSK) either before or after, but not during the read-modify-write operation on that register. This requirement ensures that modifications to IPND and IMSK take effect cleanly, completely, and at a well-defined point. Note that the processor does not assert the LOCK# pin externally when executing an atomic instruction to IPND and IMSK.

When the processor core handles a pending interrupt, it attempts to clear the bit that is latched for that interrupt in the IPND register before it begins servicing the interrupt. If that bit is associated with an interrupt source that is programmed for level detection and the true level is still present, the bit remains set. Because of this, the interrupt routine for a level-detected interrupt should clear the external interrupt source and explicitly clear the IPND bit before return from the handler is executed.

An alternative method of posting interrupts in the IPND register, other than through the external interrupt pins, is to set bits in the register directly using an **atmod** instruction. This operation has the same effect as requesting an interrupt through the external interrupt pins.

8.4.5 XINT6 Interrupt Status Register – X6ISR

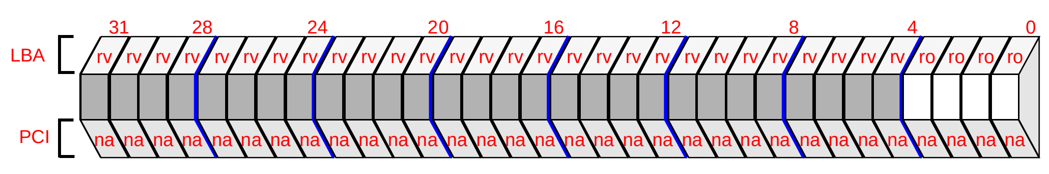
The XINT6 Interrupt Status Register (X6ISR) shows the pending XINT6 interrupts. The source of the XINT6 interrupt can be the internal peripheral devices connected through the XINT6 interrupt latch or the external XINT6# interrupt pin. The interrupts which are connected to the XINT6 input are detailed in Section 8.3.3, Internal Peripheral Interrupt Routing.

The X6ISR register is used to determine the source of an interrupt on the XINT6# input. All bits within this register are defined as read-only. The bits within this register are cleared when the source of the interrupt (status register source shown in Table 8-4) are cleared. X6ISR reflects the current state of the input to the XINT6 interrupt latch.

Due to the asynchronous nature of the 80960Rx internal peripheral units, multiple interrupts can be active when application software reads the X6ISR register. Application software must handle the occurrence of multiple interrupts. In addition, software may subsequently read X6ISR to determine when additional interrupts have occurred while processing the current interrupts. All interrupts from X6ISR will be at the same priority level within the i960 core.

Table 8-16 details the X6ISR register.

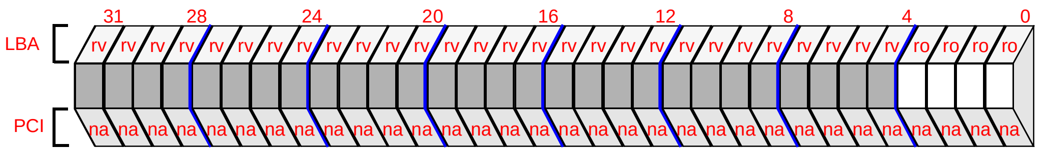
Table 8-16. XINT6 Interrupt Status Register – X6ISR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1708H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:04	0000 000H	Reserved.
03	0_2	External XINT6# Interrupt Pending - when set, an interrupt is pending on the external XINT6# input. When clear, no interrupt exists.

Table 8-16. XINT6 Interrupt Status Register – X6ISR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1708H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
02	0_2	DMA Channel 2 Interrupt Pending - when set, a DMA channel 2 interrupt is pending. When clear, no interrupt condition exists.
01	0_2	DMA Channel 1 Interrupt Pending - when set, a DMA channel 1 interrupt is pending. When clear, no interrupt condition exists.
00	0_2	DMA Channel 0 Interrupt Pending - when set, a DMA channel 0 interrupt is pending. When clear, no interrupt condition exists.

8.4.6 XINT7 Interrupt Status Register – X7ISR

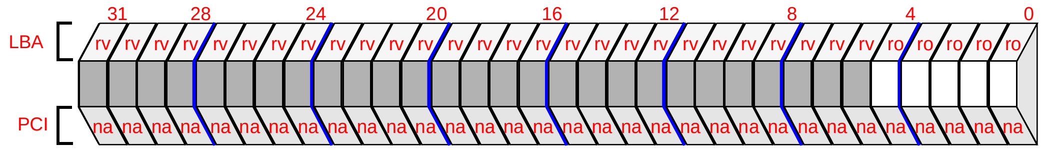
The XINT7 Interrupt Status Register (X7ISR) shows the pending XINT7 interrupts. The source of the XINT7 interrupt can be the internal peripheral devices connected through the XINT7 interrupt latch or the external XINT7# interrupt pin. The interrupts which are connected to the XINT7# input are detailed in Section 8.3.3, Internal Peripheral Interrupt Routing.

The X7ISR register is used to determine the source of an interrupt on the XINT7# input. All bits within this register are defined as read-only. The bits within this register are cleared when the source of the interrupt (status register source shown in Table 8-5) are cleared. X7ISR reflects the current state of the input to the XINT7 interrupt latch.

Due to the asynchronous nature of the 80960Rx internal peripheral units, multiple interrupts can be active when the application software reads the X7ISR register. It is up to the application software to handle the occurrence of multiple interrupts. In addition, software may subsequently read X7ISR to determine when additional interrupts have occurred while processing the current interrupts. All X7ISR interrupts will be at the same priority level within the i960 core.

Table 8-16 details the definition of the X7ISR.

Table 8-17. XINT7 Interrupt Status Register – X7ISR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1704H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 local bus address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
		RO = Read Only
		RW = Read/Write

Bit	Default	Description
31:05	0000 000H	Reserved.
04	0_2	External XINT7# Interrupt Pending - when set, an interrupt is pending on the external XINT7# input. When clear, no interrupt exists.
03	0_2	Primary ATU/Start BIST Interrupt Pending - when set, the host processor has set the start BIST request in the ATUBISTR register. When clear, no start BIST interrupt is pending.
02	0_2	Inbound Doorbell Interrupt Pending - when set, an interrupt from the Inbound Doorbell Unit is pending. When clear, no interrupt is pending.
01	0_2	I^2C Interrupt Pending - when set, an interrupt is from the I^2C Bus Interface Unit is pending. When clear, no interrupt is pending.
00	0_2	APIC Interrupt Pending - when set, an interrupt from the APIC Bus Interface Unit is pending. When clear, no interrupt is pending.

8.4.7 NMI Interrupt Status Register – NISR

The NMI Interrupt Status Register (NISR) shows the pending NMI interrupts. The source of the NMI interrupt can be the internal peripheral devices connected through the NMI Interrupt Latch or the external NMI# interrupt pin. The interrupts which are connected to the NMI# input are detailed in Section 8.3.3, Internal Peripheral Interrupt Routing.

The NMI Interrupt Status Register is used to determine the source of an interrupt on the NMI# input. All of the bits within the NISR are read-only. The bits within this register are cleared when the source of the interrupt (status register source shown in Table 8-6) are cleared. NISR reflects the current state of the input to the NMI Interrupt Latch. Note that although the NMI# input of the i960 core is edge triggered, the external NMI# input of the i960 Rx I/O processor requires a level input and must be latched external to the i960 Rx I/O processor.

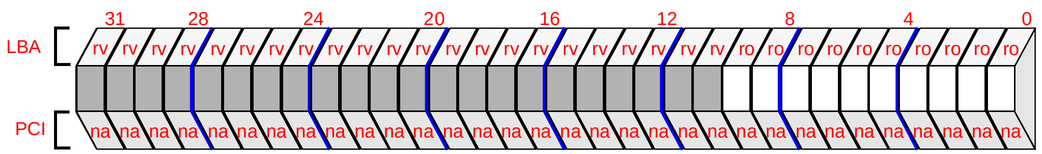
Due to the asynchronous nature of the 80960Rx internal peripheral units, multiple interrupts can be active when the application software reads the NISR register. It is up to the application software to handle the occurrence of multiple interrupts. In addition, software must check the contents of the NISR to ensure all NMI sources are cleared before returning from the NMI interrupt service routine. All NISR interrupts will be at the same priority level within the i960 core.

Example 8-5. Example Code - NMI Interrupt Handler Main Loop

```
/*  NMI Interrupt Handler */
volatile unsigned long int NISR;
do
      {   NISR = *NISR_reg_addr;
if (NISR & 1)
      80960_core_error();
if (NISR & 2)
      primary_atu_error();
if (NISR & 4)
      secondary_atu_error();
if (NISR & 8)
      primary_bridge_interface_error();
if (NISR & 16)
      secondary_bridge_interface_error();
if (NISR & 32)
      dma_channel_0_error();
if (NISR & 64)
      dma_channel_1_error();
if (NISR & 128)
      dma_channel_1_error();
if (NISR & 256)
      messaging_unit_interrupt();
if (NISR & 512)
      extnernal_nmi_interrupt();  }
while( !NISR );
return;
```

Table 8-18 shows the bit definitions for reading the NMI interrupt status register.

Table 8-18. NMI Interrupt Status Register – NISR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
ro ro ro ro ro ro ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1700H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:10	0000 00H	Reserved.
09	0_2	External NMI# Interrupt - when set, an interrupt is pending on the external NMI# input. When clear, no interrupt exists.
08	0_2	Messaging Unit Interrupt - when set, an NMI interrupt or error exists in the Messaging Unit. When clear, no error exists.
07	0_2	DMA Channel 2 Error - when set, a PCI or local bus error condition exists within DMA channel. When clear, no error exists.
06	0_2	DMA Channel 1 Error - when set, a PCI or local bus error condition exists within DMA channel. When clear, no error exists.
05	0_2	DMA Channel 0 Error - when set, a PCI or local bus error condition exists within DMA channel. When clear, no error exists.
04	0_2	Secondary Bridge Error - when set, a PCI error condition exists within the secondary interface of the bridge. When clear, no error exists.
03	0_2	Primary Bridge Interface Error - when set, a PCI error condition exists within the primary interface of the bridge. When clear, no error exists.
02	0_2	Secondary ATU Error - when set, a PCI or local bus error condition exists within the secondary ATU. When clear, no error exists.
01	0_2	Primary ATU Error - when set, a PCI or local bus error condition exists within the primary ATU. When clear, no error exists.
00	0_2	i960 core Error - when set, an error condition caused by the i960 core exists within the internal memory controller. When clear, no error exists.

8.4.8 Interrupt Controller Register Access Requirements

A load instruction that accesses the IPND, IMSK, IMAP2:0 or ICON register has a latency of one internal processor cycle. A store access to an interrupt register is synchronous with respect to the next instruction; that is, the operation completes fully and all state changes take effect before the next instruction begins execution.

Interrupts can be enabled and disabled quickly by the **intdis** and **inten** instructions, which take four cycles each to execute. **intctl** takes a few cycles longer because it returns the previous interrupt enable value. See CHAPTER 6, INSTRUCTION SET REFERENCE for more information on these instructions.

8.4.9 Default and Reset Register Values

The interrupt logic is reset by the primary PCI reset signal or through software. Table 8-19 shows the power-up and reset values. Refer to section 11.4, INITIAL MEMORY IMAGE (IMI) (pg. 11-11) for more information on register values after reset.

Table 8-19. Default Interrupt Routing and Status Values Summary

Register	Default Value	Description
PCI Interrupt Routing Select Register	0000 0000H	S_INTA#/XINT0# routed to the P_INTXA# S_INTB#/XINT1# routed to the P_INTXB# S_INTC#/XINT2# routed to the P_INTXC# S_INTD#/XINT3# routed to the P_INTXD#
SDER Secondary Decode Enable Register	0000H	All NMI# sources are enabled
NMI Interrupt Status Register	0000 0000H	No interrupts set
XINT7 Interrupt Status Register	0000 0000H	No interrupts set
XINT6 Interrupt Status Register	0000 0000H	No interrupts set
IPND	undefined	Software responsible for clearing this register before unmasking any interrupts
IMSK	0000 0000H	All interrupts masked
ICON	Initial Image in Control Table	Set to user’s values
IMAP2:0	Initial Image in Control Table	Set to user’s values

8.5 OPTIMIZING INTERRUPT PERFORMANCE

Figure 8-9 depicts the path from interrupt source to interrupt service routine. This section discusses interrupt performance in general and suggests techniques the application can use to get the best interrupt performance.

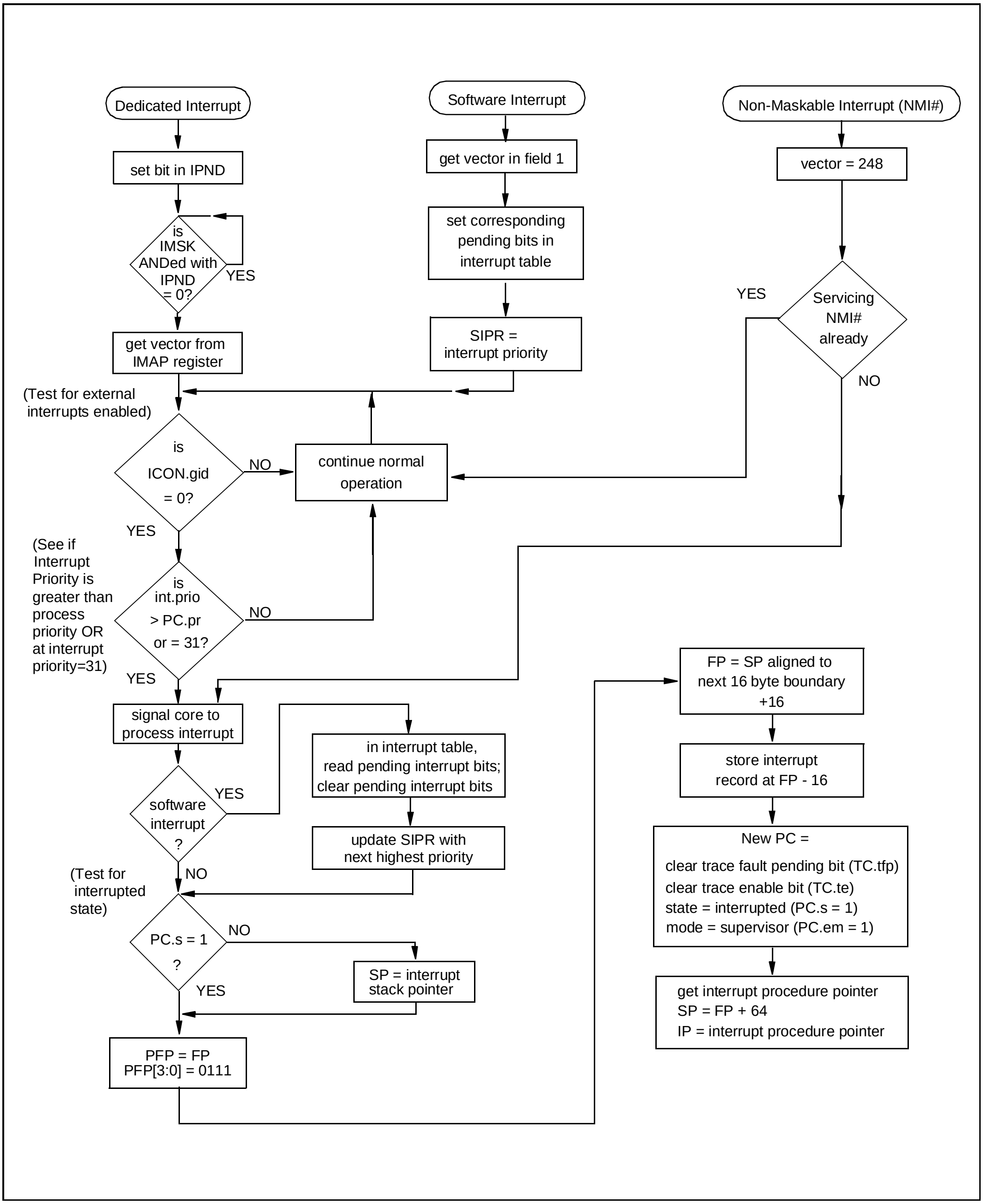

Dedicated Interrupt
set bit in IPND
is IMSK ANDed with IPND = 0?
YES
get vector from IMAP register
(Test for external interrupts enabled)
is ICON.gid = 0?
NO
YES
(See if Interrupt Priority is greater than process priority OR at interrupt priority=31)
is int.prio > PC.pr or = 31?
NO
YES
signal core to process interrupt
software interrupt ?
YES
NO
(Test for interrupted state)
PC.s = 1 ?
NO
YES
PFP = FP
PFP[3:0] = 0111
Software Interrupt
get vector in field 1
set corresponding pending bits in interrupt table
SIPR = interrupt priority
continue normal operation
in interrupt table, read pending interrupt bits; clear pending interrupt bits
update SIPR with next highest priority
SP = interrupt stack pointer
Non-Maskable Interrupt (NMI#)
vector = 248
Servicing NMI# already
YES
NO
FP = SP aligned to next 16 byte boundary +16
store interrupt record at FP - 16
New PC =
clear trace fault pending bit (TC.tfp)
clear trace enable bit (TC.te)
state = interrupted (PC.s = 1)
mode = supervisor (PC.em = 1)
get interrupt procedure pointer
SP = FP + 64
IP = interrupt procedure pointer

Figure 8-9. Interrupt Service Flowchart

8.5.1 Interrupt Service Latency

The established measure of interrupt performance is the time required to perform an interrupt task switch, which is known as *interrupt service latency*. Latency is the time measured between interrupt source activation and execution of the first instruction for the accompanying interrupt-handling procedure.

Interrupt latency depends on interrupt controller configuration and the instruction being executed at the time of the interrupt. The processor also has a number of cache options that reduce interrupt latency. In the discussion that follows, interrupt latency is expressed as a number of bus clock cycles.

8.5.2 Features to Improve Interrupt Performance

The i960 Rx I/O processor employs four methods to reduce interrupt latency:

- Caching interrupt vectors on-chip
- Caching of interrupt handling procedure code
- Reserving register frames in the local register cache
- Caching the interrupt stack in the data cache

8.5.2.1 Vector Caching Option

To reduce interrupt latency, the i960 Rx I/O processors cache some interrupt table vector entries in internal data RAM. When the vector cache option is enabled and an interrupt request has a cached vector to be serviced, the controller fetches the associated vector from internal RAM rather than from the interrupt table in memory.

Interrupts with a vector number with the four least-significant bits equal to 0010_2 can be cached. Vectors that can be cached coincide with the vector numbers selected with the mapping registers and assigned to dedicated-mode inputs. The vector caching option is selected when programming the ICON register; software must explicitly store the vector entries in internal RAM.

Since the internal RAM is mapped to the address space directly, this operation can be performed using the core’s store instructions. Table 8-20 shows the required vector mapping to specific locations in internal RAM. For example, the vector entry for vector number 18 must be stored at RAM location 04H, and so on.

The NMI# vector is also shown in Table 8-20. This vector is always cached in internal data RAM at location 0000H. The processor automatically loads this location at initialization with the value of vector number 248 in the interrupt table.

Table 8-20. Location of Cached Vectors in Internal RAM

Vector Number (Binary)	Vector Number (Decimal)	Internal RAM Address
(NMI#)	248	0000H
$0001\ 0010_2$	18	0004H
$0010\ 0010_2$	34	0008H
$0011\ 0010_2$	50	000CH
$0100\ 0010_2$	66	0010H
$0101\ 0010_2$	82	0014H
$0110\ 0010_2$	98	0018H
$0111\ 0010_2$	114	001CH
$1000\ 0010_2$	130	0020H
$1001\ 0010_2$	146	0024H
$1010\ 0010_2$	162	0028H
$1011\ 0010_2$	178	002CH
$1100\ 0010_2$	194	0030H
$1101\ 0010_2$	210	0034H
$1110\ 0010_2$	226	0038H
$1111\ 0010_2$	242	003CH

8.5.2.2 Caching Interrupt Routines and Reserving Register Frames

The time required to fetch the first instructions of an interrupt-handling procedure affects interrupt response time and throughput. The controller reduces this fetch time by caching interrupt procedures or portions of procedures in the i960 Rx I/O processor's instruction cache.

To decrease interrupt latency for high priority interrupts (priority 28 and above), software can limit the number of frames in the local register cache available to code running at a lower priority (priority 27 and below). This ensures that some number of free frames are available to high-priority interrupt service routines. See section 4.2, LOCAL REGISTER CACHE (pg. 4-2), for more details.

8.5.2.3 Caching the Interrupt Stack

By locating the interrupt stack in memory that can be cached by the data cache, the performance of interrupt returns can be improved. This is because accesses to the interrupt record by the interrupt return can be satisfied by the data cache. See section 12.2, PROGRAMMING THE PHYSICAL MEMORY ATTRIBUTES (PMCON REGISTERS) (pg. 12-3) for details on how to enable data caching for portions of memory.

8.5.3 Base Interrupt Latency

In many applications, the processor's instruction mix and cache configuration are known sufficiently well to use typical interrupt latency in calculations of overall system performance. For example, a timer interrupt may frequently trigger a task switch in a multi-tasking kernel. Base interrupt latency assumes the following:

- Single-cycle RISC instruction is interrupted.
- Frame flush does not occur.
- Bus queue is empty.
- Cached interrupt handler.
- No interaction of faults and interrupts (i.e., a stable system).

Table 8-21 shows the base latencies for all interrupt types, with varying vector caching options.

Table 8-21. Base Interrupt Latency

Interrupt Type	Vector Caching Enabled	Typical 80960Rx Latency (Bus Clocks)
NMI#	Yes	30
XINT5:4#, TINT1:0	Yes	34
	No	40+a
XINT7:6# XINT3:0#	Yes	35
	No	41+a
Software	Yes	68
	No	69+a

NOTES:

1. a = MAX (0,N - 7)

where "N" is the number of bus cycles needed to perform a word load.

8.5.4 Maximum Interrupt Latency

In real-time applications, worst-case interrupt latency must be considered for critical handling of external events. For example, an interrupt from a mechanical subsystem may need service to calculate servo loop parameters to maintain directional control. Determining worst-case latency depends on knowledge of the processor's instruction mix and operating environment as well as the interrupt controller configuration. Excluding certain very long, uninterruptible instructions from critical sections of code reduces worst-case interrupt latency to levels approaching the base latency.

The following tables present worst case interrupt latencies based on possible execution of **divo** (r15 destination), **divo** (r3 destination), **calls** or **flushreg** instructions or software interrupt detection. The assumptions for these tables are the same as for Table 8-21, except for instruction execution.

Table 8-22. Worst-Case Interrupt Latency Controlled by divo to Destination r15

Interrupt Type	Vector Caching Enabled	Worst 80960Rx Latency (Bus Clocks)[1]
NMI#	Yes	43
XINT5:4#, TINT1:0	Yes	45
	No	45+a
XINT7:6# XINT3:0#	Yes	46
	No	46+a

NOTES:

1. a = MAX (0,N - 11), where “N” is the number of bus cycles needed to perform a word load.

Table 8-23. Worst-Case Interrupt Latency Controlled by divo to Destination r3

Interrupt Type	Vector Caching Enabled	Worst 80960Rx Latency (Bus Clocks)[1]
NMI#	Yes	60
XINT5:4#, TINT1:0	Yes	65
	No	72+a
XINT7:6# XINT3:0#	Yes	66
	No	73+a

NOTES:

1. a = MAX (0,N - 7), where “N” is the number of bus cycles needed to perform a word load.

Table 8-24. Worst-Case Interrupt Latency Controlled by calls

Interrupt Type	Vector Caching Enabled	Worst 80960Rx Latency (Bus Clocks)[1]
NMI#	Yes	54+a
XINT5:4#, TINT1:0	Yes	58+a
	No	66+a+b
XINT7:6# XINT3:0#	Yes	59+a
	No	67+a+b

NOTES:

1. a = MAX (0,N - 4)
   b = MAX (0,N - 7)

where “N” is the number of bus cycles needed to perform a word load.

Table 8-25. Worst-Case Interrupt Latency When Delivering a Software Interrupt

Interrupt Type	Vector Caching Enabled	Worst 80960Rx Latency (Bus Clocks)
NMI#	Yes	97
XINT5:4#, TINT1:0	Yes	99
	No	107+a
XINT7:6# XINT3:0#	Yes	100
	No	108+a

NOTES:

1. a = MAX (0,N - 7), where “N” is the number of bus cycles needed to perform a word load.

Table 8-26. Worst-Case Interrupt Latency Controlled by flushreg of One Stack Frame

Interrupt Type	Vector Caching Enabled	Worst 80960Rx Latency (Bus Clocks)
NMI#	Yes	78+a+b
XINT5:4#, TINT1:0	Yes	82+a+b
	No	89+a+b+c
XINT7:6# XINT3:0#	Yes	83+a+b
	No	90+a+b+c

NOTES:

1. a = MAX (0, M - 15)
   b = MAX (0, M - 28)
   c = MAX (0, N - 7)

where “M” is the number of bus cycles needed to perform a quad word store and “N” is the number of bus cycles needed to perform a word load. Interrupt latency increases rapidly as the number of flushed stack frames increases.

8.5.5 Avoiding Certain Destinations for MDU Operations

Typically, when delivering an interrupt, the processor attempts to push the first four local registers (pfp, sp, rip, and r3) onto the local register cache as early as possible. Because of register-interlock, this operation is stalled until previous instructions return their results to these registers. In most cases, this is not a problem; however, in the case of instructions performed by the Multiply/Divide Unit (**divo**, **divi**, **ediv**, **modi**, **remo**, and **remi**), the processor could be stalled for many cycles waiting for the result and unable to proceed to the next step of interrupt delivery.

Interrupt latency can be improved by avoiding the first four local registers as the destination for a Multiply/Divide Unit operation. (Registers pfp, sp, and rip should be avoided anyway for general operations as these are used for procedure linking.)

8.5.6 Secondary PCI to Primary PCI Interrupt Routing Latency

The interrupt routing logic accepts the changes to the routing control value written to the PIRSR register one clock after the write has completed. There is a one clock delay from the time that the interrupt is recognized on the input of the mux until the signal is driven either to the i960 core interrupt controller or the PCI output interrupt pins.

9

FAULTS

CHAPTER 9
FAULTS

This chapter describes the i960® Rx I/O processor's fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanisms. See section 9.10, FAULT REFERENCE (pg. 9-22) for detailed information on each fault type.

9.1 FAULT HANDLING OVERVIEW

The i960 processor architecture defines various conditions in code and/or the processor's internal state that could cause the processor to deliver incorrect or inappropriate results or that could cause it to choose an undesirable control path. These are called *fault conditions*. For example, the architecture defines faults for divide-by-zero and overflow conditions on integer calculations with an inappropriate operand value.

As shown in Figure 9-1, the architecture defines a fault table, a system procedure table, a set of fault handling procedures and stacks (user stack, supervisor stack and interrupt stack) to handle processor-generated faults.

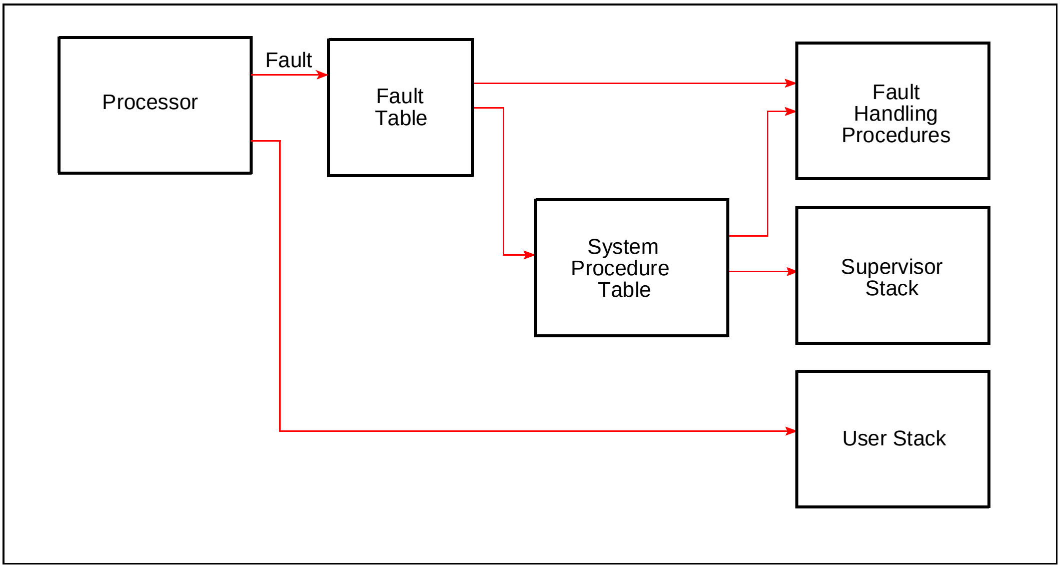


Figure 9-1. Fault-Handling Data Structures

The fault table contains pointers to fault handling procedures. The system procedure table optionally provides an interface to any fault handling procedure and allows faults to be handled in supervisor mode. Stack frames for fault handling procedures are created on either the user or supervisor stack, depending on the mode in which the fault is handled. When the processor is in the interrupted state, the processor uses the interrupt stack.

Once these data structures and the code for the fault procedures are established in memory, the processor handles faults automatically and independently from application software.

The processor can detect a fault at any time while executing instructions, whether from a program, interrupt handling procedure or fault handling procedure. When a fault occurs, the processor determines the fault type and selects a corresponding fault handling procedure from the fault table. It then invokes the fault handling procedure by means of an implicit call. As described later in this chapter, the fault handler call can be:

- A local call (call-extended operation)
- A system-local call (local call through the system procedure table)
- A system-supervisor call (supervisor call through the system procedure table)

A normal fault condition is handled by the processor in the following manner:

- The current local registers are saved and cached on-chip.
- PFP = FP and the value 001 is written to the Return Type Field (Fault Call). Refer to section 7.8, RETURNS (pg. 7-20) for more information.
- When the fault call is a system-supervisor call from user mode, the processor switches to the supervisor stack; otherwise, SP is re-aligned on the current stack.
- The processor writes the fault record on the new stack. This record includes information on the fault and the processor’s state when the fault was generated.
- The Instruction Pointer (IP) of the first instruction of the fault handler is accessed through the fault table or through the system procedure table (for system fault calls).

After the fault record is created, the processor executes the selected fault handling procedure. When a fault is recoverable (i.e., the program can be resumed after handling the fault) the Return Instruction Pointer (RIP) is defined for the fault being serviced (see section 9.10, FAULT REFERENCE (pg. 9-22), and the processor resumes execution at the RIP upon return from the fault handler. When the RIP is undefined, the fault handling procedure can create one by using the **flushreg** instruction followed by a modification of the RIP in the previous frame. The fault handler can also call a debug monitor or reset the processor instead of resuming prior execution.

This procedure call mechanism also handles faults that occur:

- While the processor is servicing an interrupt
- While the processor is servicing another fault

9.2 FAULT TYPES

The i960 architecture defines a basic set of faults that are categorized by type and subtype. Each fault has a unique type and subtype number. When the processor detects a fault, it records the fault type and subtype numbers in the fault record. It then uses the type number to select the fault handling procedure.

The fault handling procedure can optionally use the subtype number to select a specific fault handling action. The i960 Rx I/O processor recognizes i960 architecture-defined faults and a new fault subtype for detecting unaligned memory accesses. Table 9-1 lists all faults that the i960 Rx I/O processor detects, arranged by type and subtype. Text that follows the table gives column definitions.

Table 9-1. i960® Rx I/O Processor Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	**Name**	**Number or Bit Position**	**Name**	
0H	PARALLEL	NA	NA	see section 9.6.4, Parallel Faults (pg. 9-10)
1H	TRACE	Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 Bit 7	INSTRUCTION BRANCH CALL RETURN PRERETURN SUPERVISOR MARK/BREAKPOINT	0001 0002H 0001 0004H 0001 0008H 0001 0010H 0001 0020H 0001 0040H 0001 0080H
2H	OPERATION	1H 2H 3H 4H	INVALID_OPCODE UNIMPLEMENTED UNALIGNED INVALID_OPERAND	0002 0001H 0002 0002H 0002 0003H 0002 0004H
3H	ARITHMETIC	1H 2H	INTEGER_OVERFLOW ZERO-DIVIDE	0003 0001H 0003 0002H
4H	Reserved			
5H	CONSTRAINT	1H	RANGE	0005 0001H
6H	Reserved			
7H	PROTECTION	Bit 1	LENGTH	0007 0002H
8H - 9H	Reserved			
AH	TYPE	1H	MISMATCH	000A 0001H
BH - FH	Reserved			

In Table 9-1:

- The first (left-most) column contains the fault type numbers in hexadecimal.
- The second column shows the fault type name.
- The third column gives the fault subtype number as either: (1) a hexadecimal number or (2) as a bit position in the fault record's 8-bit fault subtype field. The bit position method of indicating a fault subtype is used for certain faults (such as trace faults) in which two or more fault subtypes may occur simultaneously.
- The fourth column gives the fault subtype name. For convenience, individual faults are referenced by their fault-subtype names. Thus an OPERATION.INVALID_OPERAND fault is referred to as an INVALID_OPERAND fault; an ARITHMETIC.INTEGER_OVERFLOW fault is referred to as an INTEGER_OVERFLOW fault.
- The fifth column shows the encoding of the word in the fault record that contains the fault type and fault subtype numbers.

Other i960 processor family members may provide extensions that recognize additional fault conditions. Fault type and subtype encoding allows all faults to be included in the fault table: those that are common to all i960 processors and those that are specific to one or more family members. The fault types are used consistently for all family members. For example, Fault Type 4H is reserved for floating point faults. Any i960 processor with floating point operations uses Entry 4H to store the pointer to the floating point fault handling procedure.

9.3 FAULT TABLE

The fault table (Figure 9-2) is the processor's pathway to the fault handling procedures. It can be located anywhere in the address space. From the Process Control Block, the processor obtains a pointer to the fault table during initialization.

The fault table contains one entry for each fault type. When a fault occurs, the processor uses the fault type to select an entry in the fault table. From this entry, the processor obtains a pointer to the fault handling procedure for the type of fault that occurred. Once called, a fault handling procedure has the option of reading the fault subtype or subtypes from the fault record when determining the appropriate fault recovery action.

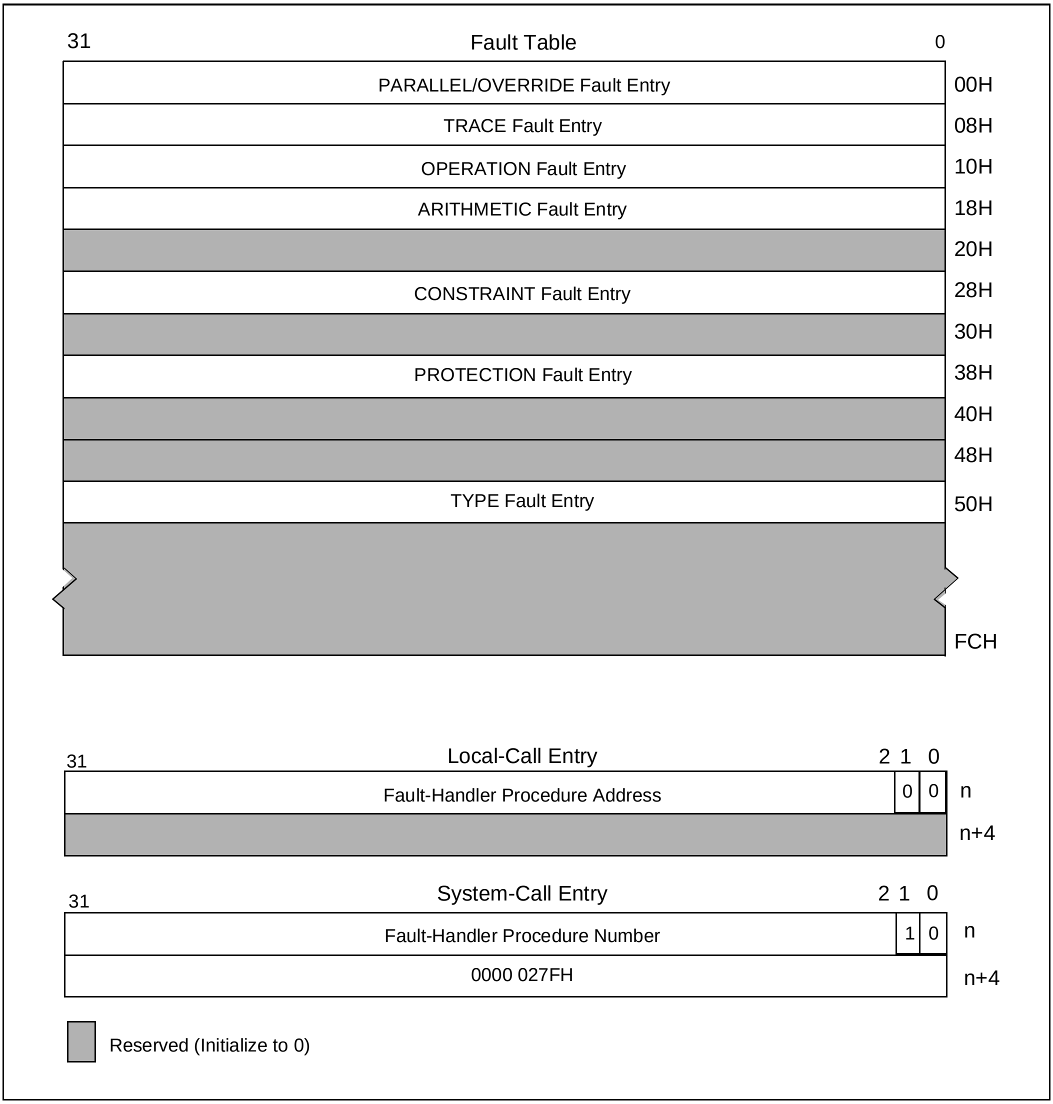


Figure 9-2. Fault Table and Fault Table Entries

As indicated in Figure 9-2, two fault table entry types are allowed: local-call entry and system-call entry. Each is two words in length. The entry type field (bits 0 and 1 of the entry's first word) and the value in the entry's second word determine the entry type.

local-call entry (type 00_2) Provides an instruction pointer for the fault handling procedure. The processor uses this entry to invoke the specified procedure by means of an implicit local-call operation. The second word of a local procedure entry is reserved. It must be set to zero when the fault table is created and not accessed after that.

system-call entry (type 10_2) Provides a procedure number in the system procedure table. This entry must have an entry type of 10_2 and a value in the second word of 0000 027FH. Using this entry, the processor invokes the specified fault handling procedure by means of an implicit call-system operation similar to that performed for the **calls** instruction. A fault handling procedure in the system procedure table can be called with a system-local call or a system-supervisor call, depending on the entry type in the system-procedure table.

Other entry types (01_2 and 11_2) are reserved and have unpredictable behavior.

To summarize, a fault handling procedure can be invoked through the fault table in any of three ways: a local call, a system-local call or a system-supervisor call.

9.4 STACK USED IN FAULT HANDLING

The i960 architecture does not define a dedicated fault handling stack. Instead, to handle a fault, the processor uses either the user, interrupt or supervisor stack, whichever is active when the fault is generated. There is, however, one exception: if the user stack is active when a fault is generated and the fault handling procedure is called with an implicit system supervisor call, the processor switches to the supervisor stack to handle the fault.

9.5 FAULT RECORD

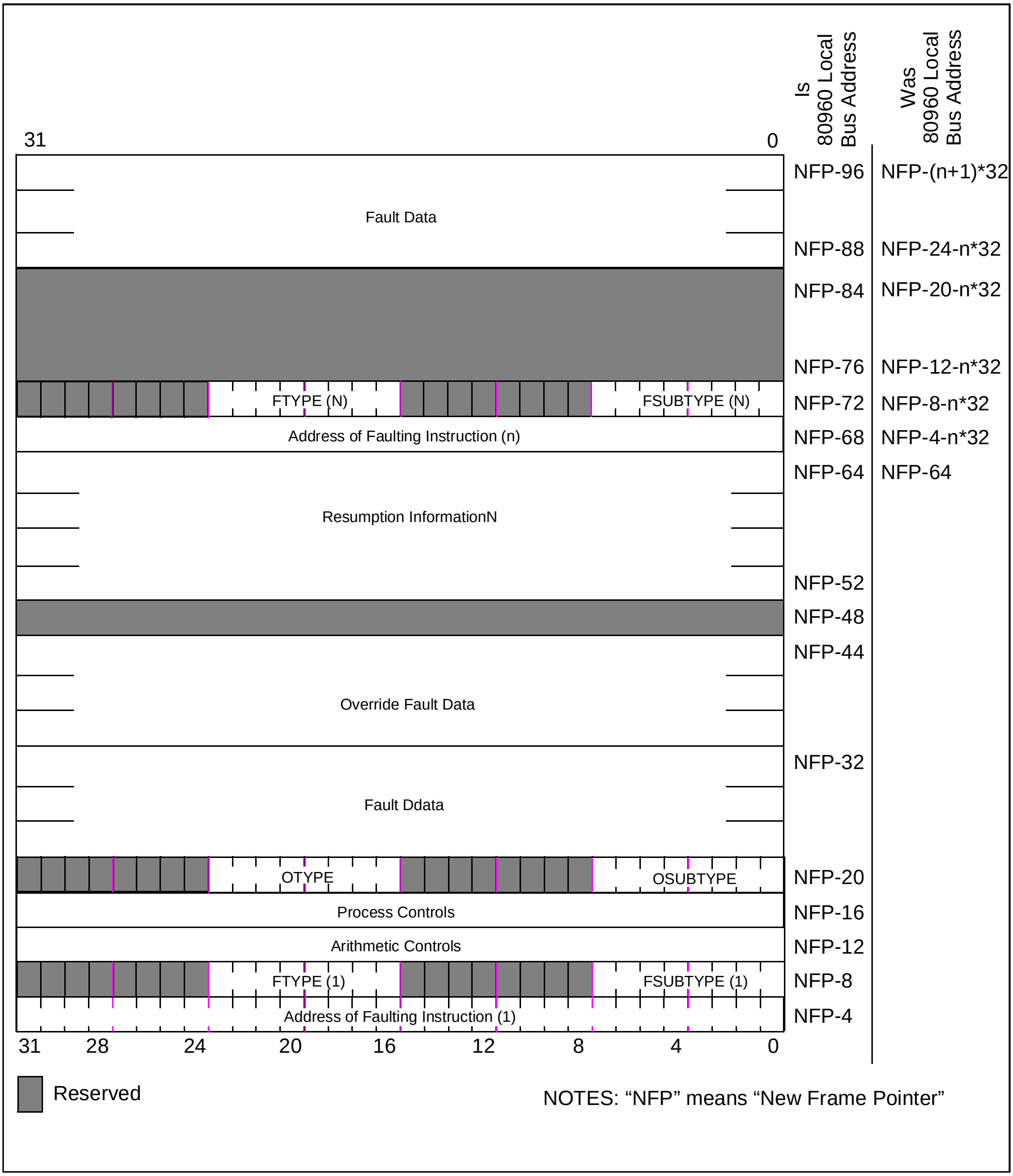
When a fault occurs, the processor records information about the fault in a fault record in memory. The fault handling procedure uses the information in the fault record to correct or recover from the fault condition and, if possible, resume program execution. The fault record is stored on the same stack that the fault handling procedure will use to handle the fault.

9.5.1 Fault Record Description

Figure 9-3 shows the fault record's structure. In this record, the fault's type number and subtype number (or bit positions for multiple subtypes) are stored in the fault type and subtype fields, respectively. The Address of Faulting Instruction Field contains the IP of the instruction that caused the processor to fault.

When a fault is generated, the existing PC and AC register contents are stored in their respective fault record fields. The processor uses this information to resume program execution after the fault is handled.

The Resumption Field is used to store information about a pending trace fault. When a trace fault and a non-trace fault occur simultaneously, the non-trace fault is serviced first and the pending trace may be lost depending on the non-trace fault encountered. The Trace Reporting paragraph for each fault specifies whether the pending trace is kept or lost.



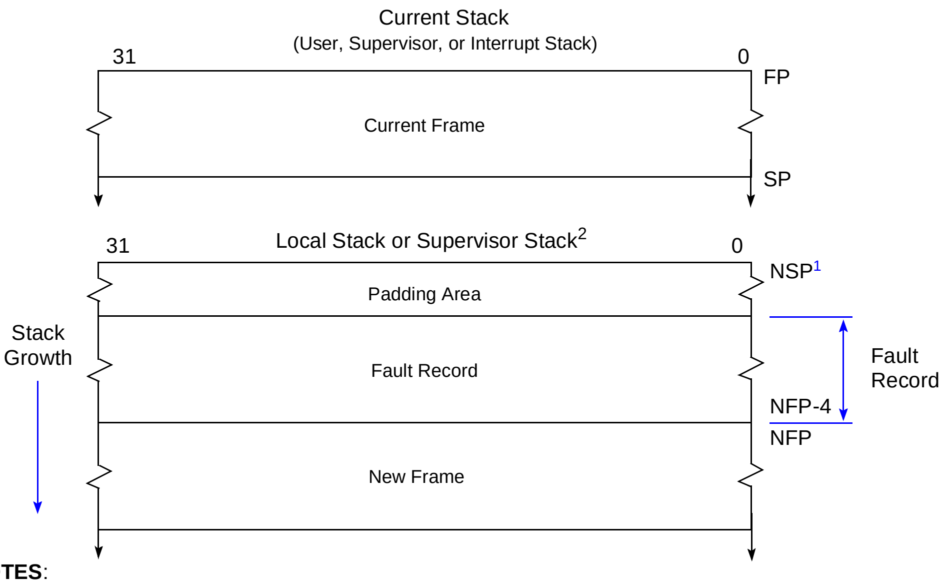
NOTES: “NFP” means “New Frame Pointer”

Figure 9-3. Fault Record

]

9.5.2 Fault Record Location

The fault record is stored on the stack that the processor uses to execute the fault handling procedure. As shown in Figure 9-4, this stack can be the user stack, supervisor stack or interrupt stack. The fault record begins at byte address NFP-1. NFP refers to the new frame pointer that is computed by adding the memory size allocated for padding and the fault record to the new stack pointer (NSP). The processor rounds the FP to the next 16-byte boundary and then allocates 80 bytes for the fault record.



NOTES:

1. If the call to the fault handler procedure does not require a stack switch, the new stack pointer (NSP) is the same as SP.
2. If the processor is in user mode and the fault handler procedure is called with a system supervisor call, the processor switches to the supervisor stack.

Figure 9-4. Storage of the Fault Record on the Stack

9.6 MULTIPLE AND PARALLEL FAULTS

Multiple fault conditions can occur during a single instruction execution and during multiple instruction execution when the instructions are executed by different units within the processor. The following sections describe how faults are handled under these conditions.

9.6.1 Multiple Non-Trace Faults on the Same Instruction

Multiple fault conditions can occur during a single instruction execution. For example, an instruction can have an invalid operand and unaligned address. When this situation occurs, the processor is required to recognize and generate at least one of the fault conditions. The processor may not detect all fault conditions and will report only one detected non-trace fault on a single instruction.

In a multiple fault situation, the reported fault condition is left to the implementation.

9.6.2 Multiple Trace Fault Conditions on the Same Instruction

Trace faults on different instructions cannot happen concurrently, because trace faults are precise (see section 9.9, PRECISE AND IMPRECISE FAULTS (pg. 9-20)). Multiple trace fault conditions on the same instruction are reported in a single trace fault record (with the exception of prereturn trace, which always happens alone). To support multiple fault reporting, the trace fault uses bit positions in the fault-subtype field to indicate occurrences of multiple faults of the same type (see Table 9-1).

9.6.3 Multiple Trace and Non-Trace Fault Conditions on the Same Instruction

The execution of a single instruction can create one or more trace fault conditions in addition to multiple non-trace fault conditions. When this occurs:

- The pending trace is dismissed if any of the non trace faults dismisses it, as mentioned in the “Trace Reporting” paragraph for that fault in section 9.10, FAULT REFERENCE (pg. 9-22).
- The processor services one of the non trace faults.
- Finally, the trace is serviced upon return from the non-trace fault handler if it was not dismissed in step 1.

9.6.4 Parallel Faults

The i960 Rx I/O processor exploits the architecture’s tolerance of out-of-order instruction execution by issuing instructions to independent execution units on the chip. The following subsections describe how the processor handles faults in this environment.

9.6.4.1 Faults on Multiple Instructions Executed in Parallel

When AC.nif=0, imprecise faults relative to different instructions executing in parallel may be reported in a single parallel fault record. For these conditions, the processor calls a unique fault handler, the PARALLEL fault handler (see section 9.9.4, No Imprecise Faults (AC.nif) Bit (pg. 9-21)). This mechanism allows instructions that can fault to be executed in parallel with other instructions or out of order.

In parallel fault situations, the processor saves the fault type and subtype of the second and subsequent faults detected in the optional section of the fault record. The optional section is the area below NFP-64 where the fault records for each of the parallel faults that occurred are stored. The fault handling procedure for parallel faults can then analyze the fault record and handle the faults. The fault record for parallel faults is described in the next section.

When the RIP is undefined for at least one of the faults found in the parallel fault record, then the RIP of the parallel fault handler is undefined. In this case, the parallel fault handling procedure can either create a RIP and return or call a debug monitor to analyze the faults.

When the RIP is defined for all faults found in the fault record, then it will point to the next instruction not yet executed. The parallel fault handler can simply return to the next instruction not yet executed with a **ret** instruction.

Consider the following code example, where the **muli** and the **addi** instructions both have overflow conditions. AC.om=0, AC.nif = 0, and both instructions are in the instruction cache at the time of their execution. The **addi** and **muli** are allowed to execute in parallel because AC.nif = 0 and the faults that these instructions can generate (ARITHMETIC) are imprecise.

```
muli g2, g4, g6;
addi g8, g9, g10;    # results in integer overflow
```

The fault on the **addi** is detected before the fault on the **muli** because the **muli** takes longer to execute. The fault call synchronizes faults on the way to the overflow fault handler for the **addi** instruction (see section 9.9.5, Controlling Fault Precision (pg. 9-21)), which is when the **muli** fault is detected. The processor builds a parallel fault record with information relative to both faults and calls the parallel fault handler. In the fault handler, ARITHMETIC faults may be recovered by storing the desired result of the instruction in the proper destination register and setting the AC.of flag (optional) to indicate that an overflow occurred. A **ret** at the end of the parallel fault handler routine will then return to the next instruction not yet executed in the program flow.

On the i960 Rx I/O processor, the **muli** overflow fault is the only fault that can happen with a delay. Therefore, parallel fault records can report a maximum of 2 faults, one of which must be a **muli** ARITHMETIC.INTEGER_OVERFLOW fault.

A parallel fault handler must be accessed through a system-supervisor call. Local and system-local parallel fault handlers are not supported by the architecture and have unpredictable behavior. Tracing is disabled upon entry into the parallel fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the parallel fault handler should not set PC.te.

9.6.4.2 Fault Record for Parallel Faults

When parallel faults occur, the processor selects one of the faults and records it in the first 16 bytes of the fault record as described in section 9.5.1, Fault Record Description (pg. 9-7). The remaining parallel faults are written to the fault record's optional section, and the fault handling procedure for parallel faults is invoked. Figure 9-3 shows the structure of the fault record for parallel faults.

The OType/OSubtype word at NFP - 20 contains the number of parallel faults. The optional section also contains a 32-byte parallel fault record for each additional parallel fault. These parallel fault records are stored incrementally in the fault record starting at byte offset NFP-68. The fault record for each additional fault contains only the fault type, fault subtype, address-of-faulting-instruction and the optional fault section. (For example, when two parallel faults occur, the fault record for the second fault is located from NFP-96 to NFP-65.)

For the second fault recorded (n=2), the relationship (NFP-8-(n * 32)) reduces to NFP-72. For the i960 Rx I/O processor, a maximum of two faults are reported in the parallel fault record, and one of them must be the ARITHMETIC.INTEGER_OVERFLOW fault on a **muli** instruction.

9.6.5 Override Faults

The i960 Rx I/O processor can detect a fault condition while the processor is preparing to service a previously detected fault. When this occurs, it is called an *override condition*. This section describes this condition and how the processor handles it.

A normal fault condition is handled by the processor in the following manner:

- The current local registers are saved and cached on-chip.
- PFP = FP and the value 001 is written to the Return Type Field (Fault Call). Refer to section 7.8, RETURNS (pg. 7-20) for more information.
- When the fault call is a system-supervisor call from user mode, the processor switches to the supervisor stack; otherwise, SP is re-aligned on the current stack.
- The processor writes the fault record on the new stack.
- The IP of the first instruction of the fault handler is accessed through the fault table or through the system procedure table (for system fault calls).

A fault that occurs during any of the above actions is called an override fault. In response to this condition, the processor does the following:

- Switches the execution mode to supervisor.
- Selects the override condition that shows that the writing of the fault record was unsuccessful. If no such fault exists, the processor selects one of the other fault conditions. This method ensures that the fault handler has information regarding the fault record write.
- Saves information pertaining to the override condition selected. The fault record describes the first fault as described previously. Field OType contains the fault type of the second fault, field OSubtype contains the fault subtype of the second fault and field override-fault-data contains what would normally be the fault data field for the second fault type.
- Attempts to access the IP of the first instruction in the override fault handler through the system procedure table.

It should be noted that a fault that occurs while the processor is actually executing a fault handling procedure is not an override fault.

The override fault entry is entry 0. When the override fault entry in the fault table points to a location beyond the system procedure table, the processor enters system error mode. Override fault conditions include: PROTECTION and OPERATION.UNIMPLEMENTED faults.

An override fault handler must be accessed through a system-supervisor call. Local and system-local override fault handlers are not supported by the architecture and have an unpredictable behavior. Tracing is disabled upon entry into the override fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the override fault handler should not set PC.te.

9.6.6 System Error

When a fault is detected while the processor is in the process of servicing an override or parallel fault, the processor enters the system error state. Note that “servicing” indicates that the processor has detected the override or parallel fault, but has not begun executing the fault handling procedure. This type of error causes the processor to enter a system error state. In this state, the processor uses only one read bus transaction to signal the fail code message; the address of the bus transaction is the fail code itself. See section 11.3.1.5, FAIL# Code (pg. 11-11).

9.7 FAULT HANDLING PROCEDURES

The fault handling procedures can be located anywhere in the address space except within the on-chip data RAM or MMR space. Each procedure must begin on a word boundary. The processor can execute the procedure in user or supervisor mode, depending on the fault table entry type.

9.7.1 Possible Fault Handling Procedure Actions

The processor allows easy recovery from many faults that occur. When fault recovery is possible, the processor's fault handling mechanism allows the processor to automatically resume work on the program or pending interrupt when the fault occurred. Resumption is initiated with a **ret** instruction in the fault handling procedure.

When recovery from the fault is not possible or not desirable, the fault handling procedure can take one of the following actions, depending on the nature and severity of the fault condition (or conditions, in the case of multiple faults):

- Return to a point in the program or interrupt code other than the point of the fault.
- Call a debug monitor.
- Perform processor or system shutdown with or without explicitly saving the processor state and fault information.

When working with the processor at the development level, a common fault handling strategy is to save the fault and processor state information and call a debugging tool such as a monitor.

9.7.2 Program Resumption Following a Fault

Because of the wide variety of faults, they can occur at different times with respect to the faulting instruction:

- Before execution of the faulting instruction (e.g., fetch from on-chip RAM)
- During instruction execution (e.g., integer overflow)
- Immediately following execution (e.g., trace)

9.7.2.1 Faults Happening Before Instruction Execution

The following fault types occur before instruction execution:

- ARITHMETIC.ZERO_DIVIDE
- TYPE.MISMATCH
- PROTECTION.LENGTH
- All OPERATION subtypes except UNALIGNED

For these faults, the contents of a destination register are lost, and memory is not updated. The RIP is defined for the ARITHMETIC.ZERO_DIVIDE fault only. In some cases the fault occurs before the faulting instruction is executed, the faulting instruction may be fixed and re-executed upon return from the fault handling procedure.

9.7.2.2 Faults Happening During Instruction Execution

The following fault types occur during instruction execution:

- CONSTRAINT.RANGE
- OPERATION.UNALIGNED
- ARITHMETIC.INTEGER_OVERFLOW

For these faults, the fault handler must explicitly modify the RIP to return to the faulting application (except for ARITHMETIC.INTEGER_OVERFLOW).

When a fault occurs during or after execution of the faulting instruction, the fault may be accompanied by a program state change such that program execution cannot be resumed after the fault is handled. For example, when an integer overflow fault occurs, the overflow value is stored in the destination. When the destination register is the same as one of the source registers, the source value is lost, making it impossible to re-execute the faulting instruction.

9.7.2.3 Faults Happening After Instruction Execution

For these faults, the Return Instruction Pointer (RIP) is defined and the fault handler can return to the next instruction in the flow:

- TRACE
- ARITHMETIC.INTEGER_OVERFLOW

In general, resumption of program execution with no changes in the program's control flow is possible with the following fault types or subtypes:

- All TRACE Subtypes

The effect of specific fault types on a program is defined in section 9.10, FAULT REFERENCE (pg. 9-22) under the heading Program State Changes.

9.7.3 Return Instruction Pointer (RIP)

When a fault handling procedure is called, a Return Instruction Pointer (RIP) is saved in the image of the RIP in the faulting frame. The RIP can be accessed at address PFP+8 while executing the fault handler after a **flushreg**. The RIP in the previous frame points to an instruction where program execution can be resumed with no break in the program's control flow. It generally points to the faulting instruction or to the next instruction to be executed. In some instances, however, the RIP is undefined. RIP content for each fault is described in section 9.10, FAULT REFERENCE (pg. 9-22).

9.7.4 Returning to the Point in the Program Where the Fault Occurred

As described in section 9.7.2, Program Resumption Following a Fault (pg. 9-14), most faults can be handled such that program control flow is not affected. In this case, the processor allows a program to be resumed at the point where the fault occurred, following a return from a fault handling procedure (initiated with a **ret** instruction). The resumption mechanism used here is similar to that provided for returning from an interrupt handler.

Also, to restore the PC register from the fault record upon return from the fault handler, the fault handling procedure must be executed in supervisor mode either by using a supervisor call or by running the program in supervisor mode. See the pseudocode in section 6.2.54, ret (pg. 6-92).

9.7.5 Returning to a Point in the Program Other Than Where the Fault Occurred

A fault handling procedure can also return to a point in the program other than where the fault occurred. To do this, the fault procedure must alter the RIP. To do this reliably, the fault handling procedure should perform the following steps:

1. Flush the local register sets to the stack with a **flushreg** instruction.
2. Modify the RIP in the previous frame.
3. Clear the trace-fault-pending flag in the fault record’s process controls field before the return (optional).
4. Execute a return with the **ret** instruction.

Use this technique carefully and only in situations where the fault handling procedure is closely coupled with the application program.

9.7.6 Fault Controls

For certain fault types and subtypes, the processor employs register mask bits or flags that determine whether or not a fault is generated when a fault condition occurs. Table 9-2 summarizes these flags and masks, the data structures in which they are located, and the fault subtypes they affect.

The integer overflow mask bit inhibits the generation of integer overflow faults. The use of this mask is discussed in section 9.10, FAULT REFERENCE (pg. 9-22).

The Arithmetic Controls no imprecise faults (AC.nif) bit controls the synchronizing of faults for a category of faults called imprecise faults. The function of this bit is described in section 9.9, PRECISE AND IMPRECISE FAULTS (pg. 9-20).

TC register trace mode bits and the PC register trace enable bit support trace faults. Trace mode bits enable trace modes; the trace enable bit (PC.te) enables trace fault generation. The use of these bits is described in the trace faults description in section 9.10, FAULT REFERENCE (pg. 9-22). Further discussion of these flags is provided in CHAPTER 10, TRACING AND DEBUGGING.

Table 9-2. Fault Control Bits and Masks

Flag or Mask Name	Location	Faults Affected
Integer Overflow Mask Bit	Arithmetic Controls (AC) Register	INTEGER_OVERFLOW
No Imprecise Faults Bit	Arithmetic Controls (AC) Register	All Imprecise Faults
Trace Enable Bit	Process Controls (PC) Register	All TRACE Faults
Trace Mode	Trace Controls (TC) Register	All TRACE Faults except hardware breakpoint traces and **fmark**
Unaligned Fault Mask	Process Control Block (PRCB)	UNALIGNED Fault

The unaligned fault mask bit is located in the process control block (PRCB), which is read from the fault configuration word (located at address PRCB pointer + 0CH) during initialization. It controls whether unaligned memory accesses generate a fault. See section 12.4.2, Bus Transactions Across Region Boundaries (pg. 12-7).

9.8 FAULT HANDLING ACTION

Once a fault occurs, the processor saves the program state, calls the fault handling procedure and, if possible, restores the program state when the fault recovery action completes. No software other than the fault handling procedures is required to support this activity.

Three types of implicit procedure calls can be used to invoke the fault handling procedure: a local call, a system-local call and a system-supervisor call.

The following subsections describe actions the processor takes while handling faults. It is not necessary to read these sections to use the fault handling mechanism or to write a fault handling procedure. This discussion is provided for those readers who wish to know the details of the fault handling mechanism.

9.8.1 Local Fault Call

When the selected fault handler entry in the fault table is an entry type 000_2 (a local procedure), the processor operates as described in section 7.1.3.1, Call Operation (pg. 7-6), with the following exceptions:

- A new frame is created on the stack that the processor is currently using. The stack can be the user stack, supervisor stack or interrupt stack.
- The fault record is copied into the area allocated for it in the stack, beginning at NFP-1. (See Figure 9-4.)
- The processor gets the IP for the first instruction in the called fault handling procedure from the fault table.
- The processor stores the fault return code (001_2) in the PFP return type field.

When the fault handling procedure is not able to perform a recovery action, it performs one of the actions described in section 9.7.2, Program Resumption Following a Fault (pg. 9-14).

When the handler action results in recovery from the fault, a **ret** instruction in the fault handling procedure allows processor control to return to the program that was executing when the fault occurred. Upon return, the processor performs the action described in section 7.1.3.2, Return Operation (pg. 7-7), except that the arithmetic controls field from the fault record is copied into the AC register. When the processor is in user mode before execution of the return, the process controls field from the fault record is not copied back to the PC register.

9.8.2 System-Local Fault Call

When the fault handler selects an entry for a local procedure in the system procedure table (entry type 10_2), the processor performs the same action as is described in the previous section for a local fault call or return. The only difference is that the processor gets the fault handling procedure's address from the system procedure table rather than from the fault table.

9.8.3 System-Supervisor Fault Call

When the fault handler selects an entry for a supervisor procedure in the system procedure table, the processor performs the same action described in section 7.1.3.1, Call Operation (pg. 7-6), with the following exceptions:

- When the fault occurs while in user mode, the processor switches to supervisor mode, reads the supervisor stack pointer from the system procedure table and switches to the supervisor stack. A new frame is then created on the supervisor stack.

- When the fault occurs while in supervisor mode, the processor creates a new frame on the current stack. When the processor is executing a supervisor procedure when the fault occurred, the current stack is the supervisor stack; when it is executing an interrupt handler procedure, the current stack is the interrupt stack. (The processor switches to supervisor mode when handling interrupts.)
- The fault record is copied into the area allocated for it in the new stack frame, beginning at NFP-1. (See Figure 9-4.)
- The processor gets the IP for the first instruction of the fault handling procedure from the system procedure table (using the index provided in the fault table entry).
- The processor stores the fault return code (001_2) in the PFP register return type field. When the fault is not a trace, parallel or override fault, it copies the state of the system procedure table trace control flag (byte 12, bit 0) into the PC register trace enable bit. When the fault is a trace, parallel or override fault, the trace enable bit is cleared.

On a return from the fault handling procedure, the processor performs the action described in section 7.1.3.2, Return Operation (pg. 7-7) with the addition of the following:

- The fault record arithmetic controls field is copied into the AC register.
- When the processor is in supervisor mode prior to the return from the fault handling procedure (which it should be), the fault record process controls field is copied into the PC register. The mode is then switched back to user, if it was in user mode before the call.
- The processor switches back to the stack it was using when the fault occurred. (When the processor was in user mode when the fault occurred, this operation causes a switch from the supervisor stack to the user stack.)
- When the trace-fault-pending flag and trace enable bits are set in the PC field of the fault record, the trace fault on the instruction at the origin of the supervisor fault call is handled at this time.

The user should note that PC register restoration causes any changes to the process controls done by the fault handling procedure to be lost.

9.8.4 Faults and Interrupts

When an interrupt occurs during an instruction that will fault, an instruction that has already faulted, or fault handling procedure selection, the processor:

1. Completes the selection of the fault handling procedure.

2. Creates the fault record.

3. Services the interrupt just prior to executing the first instruction of the fault handling procedure.

4. Handles the fault upon return from the interrupt.

Handling the interrupt before the fault reduces interrupt latency.

9.9 PRECISE AND IMPRECISE FAULTS

As described in section 9.10.5, PARALLEL Faults (pg. 9-29), the i960 architecture — to support parallel and out-of-order instruction execution — allows some faults to be generated together.

The processor provides two mechanisms for controlling the circumstances under which faults are generated: the AC register no-imprecise-faults bit (AC.nif) and the instructions that synchronize faults. See section 9.9.5, Controlling Fault Precision (pg. 9-21) for more information. Faults are categorized as precise, imprecise and asynchronous. The following subsections describe each.

9.9.1 Precise Faults

A fault is precise if it meets all of the following conditions:

- The faulting instruction is the earliest instruction in the instruction issue order to generate a fault.
- All instructions after the faulting instruction, in instruction issue order, are guaranteed not to have executed.

TRACE and PROTECTION.LENGTH faults are always precise. Precise faults cannot be found in parallel records with other precise or imprecise faults.

9.9.2 Imprecise Faults

Faults that do not meet all of the requirements for precise faults are considered imprecise. For imprecise faults, the state of execution of instructions surrounding the faulting instruction may be unpredictable. When instructions are executed out of order and an imprecise fault occurs, it may not be possible to access the source operands of the instruction. This is because they may have been modified by subsequent instructions executed out of order. However, the RIP of some imprecise faults (e.g., ARITHMETIC) points to the next instruction that has not yet executed and guarantees the return from the fault handler to the original flow of execution. Faults that the architecture allows to be imprecise are OPERATION, CONSTRAINT, ARITHMETIC and TYPE.

9.9.3 Asynchronous Faults

Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. This group includes MACHINE faults, which are not implemented on the 80960Rx.

9.9.4 No Imprecise Faults (AC.nif) Bit

The Arithmetic Controls no imprecise faults (AC.nif) bit controls imprecise fault generation. When AC.nif is set, out of order instruction execution is disabled and all faults generated are precise. Therefore, setting this bit will reduce processor performance. When AC.nif is clear, several imprecise faults may be reported together in a parallel fault record. Precise faults can never be found in parallel fault records, thus only more than one imprecise fault occurring concurrently with AC.nif = 0 can produce a parallel fault.

Compiled code should execute with the AC.nif bit clear, using **syncf** where necessary to ensure that faults occur in order. In this mode, imprecise faults are considered to be catastrophic errors from which recovery is not needed. This also allows the processor to take advantage of internal pipelining, which can speed up processing time. When only precise faults are allowed, the processor must restrict the use of pipelining to prevent imprecise faults.

The AC.nif bit should be set if recovery from one or more imprecise faults is required. For example, the AC.nif bit should be set if a program needs to handle and recover from unmasked integer-overflow faults and the fault handling procedure cannot be closely coupled with the application to perform imprecise fault recovery.

9.9.5 Controlling Fault Precision

The **syncf** instruction forces the processor to complete execution of all instructions that occur prior to **syncf** and to generate all faults before it begins work on instructions that occur after **syncf**. This instruction has two uses:

- It forces faults to be precise when the AC.nif bit is clear.

- It ensures that all instructions are complete and all faults are generated in one block of code before executing another block of code.

The implicit fault call operation synchronizes all faults. In addition, the following instructions or operations perform synchronization of all faults except MACHINE.PARITY:

- Call and return operations including **call**, **callx**, **calls** and **ret** instructions, plus the implicit interrupt and fault call operations.
- Atomic operations including **atadd** and **atmod**.

9.10 FAULT REFERENCE

This section describes each fault type and subtype and gives detailed information about what is stored in the various fields of the fault record. The section is organized alphabetically by fault type. The following paragraphs describe the information that is provided for each fault type.

Fault Type: Gives the number that appears in the fault record fault-type field when the fault is generated.

Fault Subtype: Lists the fault subtypes and the number associated with each fault subtype.

Function: Describes the purpose and handling of the fault type and each subtype.

RIP: Describes the value saved in the image of the RIP register in the stack frame that the processor was using when the fault occurred. In the RIP definitions, “next instruction” refers to the instruction directly after the faulting instruction or to an instruction to which the processor can logically return when resuming program execution.

Note that the discussions of many fault types specify that the RIP contains the address of the instruction that would have executed next had the fault not occurred.

Fault IP: Describes the contents of the fault record’s fault instruction pointer field, typically the faulting instruction’s IP.

Fault Data: Describes any values stored in the fault record’s fault data field.

Class: Indicates if a fault is precise or imprecise.

Program State Changes: Describes the process state changes that would prevent re-executing the faulting instruction if applicable.

Trace Reporting: Relates whether a trace fault (other than PRERET) can be detected on the faulting instruction, also if and when the fault is serviced.

Notes: Additional information specific to particular implementations of the i960 architecture.

9.10.1 ARITHMETIC Faults

Fault Type: 3H

Fault Subtype:

Number	**Name**
0H	Reserved
1H	INTEGER_OVERFLOW
2H	ZERO_DIVIDE
3H-FH	Reserved

Function: Indicates a problem with an operand or the result of an arithmetic instruction. An INTEGER_OVERFLOW fault is generated when the result of an integer instruction overflows its destination and the AC register integer overflow mask is cleared. Here, the result's *n* least significant bits are stored in the destination, where *n* is destination size. Instructions that generate this fault are:

addi	**subi**	**stis**
stib	**shli**	**ADDI<cc>**
muli	**divi**	**SUBI<cc>**

An ARITHMETIC.ZERO_DIVIDE fault is generated when the divisor operand of an ordinal- or integer-divide instruction is zero. Instructions that generate this fault are:

divo	**divi**
ediv	**remi**
remo	**modi**

RIP: IP of the instruction that would have executed next if the fault had not occurred.

Fault IP: IP of the faulting instruction.

Class: Imprecise.

Program State Changes: Faults may be imprecise when executing with the AC.nif bit cleared. INTEGER_OVERFLOW and ZERO_DIVIDE faults may not be recoverable because the result is stored in the destination before the fault is generated (e.g., the faulting instruction cannot be re-executed if the destination register was also a source register for the instruction).

Trace Reporting: The trace is reported upon return from the arithmetic fault handler.

9.10.2 CONSTRAINT Faults

Fault Type: 5H

Fault Subtype:

Number	Name
0H	Reserved
1H	RANGE
2H-FH	Reserved

Function: Indicates the program or procedure violated an architectural constraint.

A CONSTRAINT.RANGE fault is generated when a **FAULT<cc>** instruction is executed and the AC register condition code field matches the condition required by the instruction.

RIP: No defined value.

Fault IP: Faulting instruction.

Class: Imprecise.

Program State Changes: These faults may be imprecise when executing with the AC.nif bit cleared. No changes in the program’s control flow accompany these faults. A CONSTRAINT.RANGE fault is generated after the **FAULT<cc>** instruction executes. The program state is not affected.

Trace Reporting: Serviced upon return from the Constraint fault handler.

9.10.3 OPERATION Faults

Fault Type: 2H

Fault Subtype:

Number	Name
0H	Reserved
1H	INVALID_OPCODE
2H	UNIMPLEMENTED
3H	UNALIGNED
4H	INVALID_OPERAND
5H - FH	Reserved

Function: Indicates the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics.

An INVALID_OPCODE fault is generated when the processor attempts to execute an instruction containing an undefined opcode or addressing mode.

An UNIMPLEMENTED fault is generated when the processor attempts to execute an instruction fetched from on-chip data RAM, or when a non-word or unaligned access to a memory-mapped region is performed, or when attempting to write memory-mapped region 0xFF0084XX when rights have not been granted.

An UNALIGNED fault is generated when the following conditions are present: (1) the processor attempts to access an unaligned word or group of words in non-MMR memory; and (2) the fault is enabled by the unaligned-fault mask bit in the PRCB fault configuration word.

An INVALID_OPERAND fault is generated when the processor attempts to execute an instruction that has one or more operands having special requirements that are not satisfied. This fault is generated when specifying a non-defined **sysctl, icctl, dcctl or intctl** command, or referencing an unaligned long-, triple- or quad-register group, or by referencing an undefined register, or by writing to the RIP register (r2).

RIP: No defined value.

Fault IP: Address of the faulting instruction.

Fault Data: When an UNALIGNED fault is signaled, the effective address of the unaligned access is placed in the fault record’s optional data section, beginning at address NFP-24. This address is useful to debug a program that is making unintentional unaligned accesses.

Class: Imprecise.

Program State Changes: For the INVALID_OPCODE and UNIMPLEMENTED faults (case: store to MMR), the destination of the faulting instruction is not modified. (For the UNALIGNED fault, the memory operation completes correctly before the fault is reported.) In all other cases, the destination is undefined.

Trace Reporting: OPERATION.UNALIGNED fault: the trace is reported upon return from the OPERATION fault handler.
All other subtypes: the trace event is lost.

Note: OPERATION.UNALIGNED fault is not implemented on i960 Kx and Sx CPUs.

9.10.4 OVERRIDE Faults

Fault Type: Fault table entry = 10H

The fault type in the fault record on the stack equals the fault type of the initial fault. The fault type in the internal registers equals the fault type of the additional fault detected while attempting to service the initial fault.

Fault Subtype: The fault subtype in the fault record on the stack equals the fault subtype of the initial fault. The fault subtype in the internal registers equals the fault subtype of the additional fault detected while attempting to service the initial fault.

Fault OType: The fault type of the additional fault detected while attempting to deliver the program fault.

Fault OSubtype: The fault subtype of the additional fault detected while attempting to deliver the program fault.

Function: The override fault handler must be accessed through a system-supervisor call. Local and system-local override fault handlers are not supported and have an unpredictable behavior. Tracing is disabled upon entry into the override fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the override fault handler should not set PC.te.

Trace Reporting: Same behavior as if the override condition had not existed. Refer to the description of the original program fault.

9.10.5 PARALLEL Faults

Fault Type: Fault table entry = 0H
Fault type in fault record = fault type of one of the parallel faults.

Fault Subtype: Fault subtype of one of the parallel faults.

Fault OType: 0H

Fault OSubtype: Number of parallel faults.

Function: See section 9.6.4, Parallel Faults (pg. 9-10) for a complete description of parallel faults. When the AC.nif=0, the architecture permits the processor to execute instructions in parallel and out-of-order by different execution units. When an imprecise fault occurs in any of these units, it is not possible to stop the execution of those instructions after the faulting instruction. It is also possible that more than one fault is detected from different instructions almost at the same time.

When there is more than one outstanding fault at the point when all execution units terminate, a parallel fault situation arises. The fault record of parallel faults contains the fault information of all faults that occurred in parallel. The number of parallel faults is indicated in the Parallel Faults Field (NFP-20). See Figure 9-3. The maximum size of the fault record is implementation dependent and depends on the number of parallel and pipeline execution units in the specific implementation.

The parallel fault handler must be accessed through a system-supervisor call. Local and system-local parallel fault handlers are not supported by the i960 processor and have an unpredictable behavior. Tracing is disabled upon entry into the parallel fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the parallel fault handler should not set PC.te.

RIP: When all parallel fault types allow a RIP to be defined, the RIP is the next instruction in the flow of execution, otherwise it is undefined.

Fault IP: IP of one of the faulting instructions.

Class: Imprecise.

Program State Changes: State changes associated with all the parallel faults.

Trace Reporting: If all parallel fault types allow for a resumption trace, then a trace is reported upon return from the parallel fault handler, or else it is lost.

9.10.6 PROTECTION Faults

Fault Type: 7H

Fault Subtype:

Number	Name
Bit 0	Reserved
Bit 1	LENGTH
Bit 2-7	Reserved

Function: Indicates that a program or procedure is attempting to perform an illegal operation that the architecture protects against.

A PROTECTION.LENGTH fault is generated when the index operand, used in a **calls** instruction, points to an entry beyond the extent of the system procedure table.

RIP: IP of the faulting instruction.

IP of the faulting instruction.

Fault IP: LENGTH: IP of the faulting instruction.

Class: Imprecise. (PROTECTION.LENGTH is precise even though the PROTECTION fault class is imprecise.)

Program State Changes: LENGTH: The instruction does not execute.

Trace Reporting: PROTECTION.LENGTH: The trace event is lost.

9.10.7 TRACE Faults

Fault Type: 1H

Fault Subtype:

Number	Name
Bit 0	Reserved
Bit 1	INSTRUCTION
Bit 2	BRANCH
Bit 3	CALL
Bit 4	RETURN
Bit 5	PRERETURN
Bit 6	SUPERVISOR
Bit 7	MARK/BREAKPOINT

Function: Indicates the processor detected one or more trace events. The event tracing mechanism is described in CHAPTER 10, TRACING AND DEBUGGING.

A trace event is the occurrence of a particular instruction or instruction type in the instruction stream. The processor recognizes seven different trace events: instruction, branch, call, return, prereturn, supervisor, mark. It detects these events only if the TC register mode bit is set for the event. If the PC register trace enable bit is also set, the processor generates a fault when a trace event is detected.

A TRACE fault is generated following the instruction that causes a trace event (or prior to the instruction for the prereturn trace event). The following trace modes are available:

INSTRUCTION — Generates a trace event following every instruction.

BRANCH — Generates a trace event following any branch instruction when the branch is taken (a branch trace event does not occur on branch-and-link or call instructions).

CALL — Generates a trace event following any call or branch-and-link instruction or an implicit fault call.

RETURN — Generates a trace event following a **ret**.

PRERETURN — Generates a trace event prior to any **ret** instruction, provided the PFP register prereturn trace flag is set (the processor sets the flag automatically when a call trace is serviced). A prereturn trace fault is always generated alone.

SUPERVISOR — Generates a trace event following any **calls** instruction that references a supervisor procedure entry in the system procedure table and on a return from a supervisor procedure where the return status type in the PFP register is 010_2 or 011_2.

MARK/BREAKPOINT — Generates a trace event following the **mark** instruction. The MARK fault subtype bit, however, is used to indicate a match of the instruction-address breakpoint register or the data-address breakpoint register as well as the **fmark** and **mark** instructions.

A TRACE fault subtype bit is associated with each mode. Multiple fault subtypes can occur simultaneously; all trace fault conditions detected on one instruction (except prereturn) are reported in one single trace fault, with the fault subtype bit set for each subtype that occurs. The prereturn trace is always reported alone.

When a fault type other than a TRACE fault is generated during execution of an instruction that causes a trace event, the non-trace fault is handled before the trace fault. An exception is the prereturn-trace fault, which occurs before the processor detects a non-trace fault and is handled first.

Similarly, if an interrupt occurs during an instruction that causes a trace event, the interrupt is serviced before the TRACE fault is handled. Again, the TRACE.PRERETURN fault is different. Since it is generated before the instruction, it is handled before any interrupt that occurs during instruction execution.

A trace fault handler must be accessed through a system-supervisor call (it must be a supervisor procedure in the system procedure table). Local and system-local trace fault handlers are not supported by the architecture and may have unpredictable behavior. Tracing is automatically disabled when entering the trace fault handler and is restored upon return from the trace fault handler. The trace fault handler should not modify PC.te.

RIP: Instruction immediately following the instruction traced, in instruction issue order, except for PRERETURN. For PRERETURN, the RIP is the return instruction traced.

Fault IP: IP of the faulting instruction for all except prereturn trace and call trace (on implicit fault calls), for which the fault IP field is undefined.

Class: Precise.

Program State Changes: All trace faults except PRERETURN are serviced after the execution of the faulting instruction. The processor returns to the instruction immediately following the instruction traced, in instruction issue order. For PRERETURN, the return is traced before it executes. The processor re-executes the return instruction after completion of the PRERETURN trace fault handler.

9.10.8 TYPE Faults

Fault Type: AH

Fault Subtype:

Number	Name
0H	Reserved
1H	MISMATCH
2H-FH	Reserved

Function: Indicates a program or procedure attempted to perform an illegal operation on an architecture-defined data type or a typed data structure.

A TYPE.MISMATCH fault is generated when attempts are made to:

- Execute a privileged (supervisor-mode only) instruction while the processor is in user mode. Privileged instructions on the i960 Rx I/O processor are:

  modpc **intctl**
  sysctl **inten**
  icctl **intdis**
  dcctl

- Write to on-chip data RAM while the processor is in supervisor-only write mode and BCON.irp is set.
- Write to the first 64 bytes of on-chip data RAM while the processor is in either user or supervisor mode and BCON.sirp is set.
- Write to memory-mapped registers in supervisor space from user mode.
- Write to timer registers while in user mode, when timer registers are protected against user-mode writes.

RIP: No defined value.

Fault IP: IP of the faulting instruction.

Class: Imprecise.

Program State Changes: The fault happens before execution of the instruction. Machine state is not changed.

Trace Reporting: The trace event is lost.

10

TRACING AND DEBUGGING

CHAPTER 10
TRACING AND DEBUGGING

This chapter describes the i960® Rx I/O processor's facilities for runtime activity monitoring. The i960 architecture provides facilities for monitoring processor activity through trace event generation. A trace event indicates a condition where the processor has just completed executing a particular instruction or a type of instruction or where the processor is about to execute a particular instruction. When the processor detects a trace event, it generates a trace fault and makes an implicit call to the fault handling procedure for trace faults. This procedure can, in turn, call debugging software to display or analyze the processor state when the trace event occurred. This analysis can be used to locate software or hardware bugs or for general system monitoring during program development.

Tracing is enabled by the process controls (PC) register trace enable bit and a set of trace mode bits in the trace controls (TC) register. Alternatively, the **mark** and **fmark** instructions can be used to generate trace events explicitly in the instruction stream.

The i960 Rx I/O processor also provides four hardware breakpoint registers that generate trace events and trace faults. Two registers are dedicated to trapping on instruction execution addresses, while the remaining two registers can trap on the addresses of various types of data accesses.

10.1 TRACE CONTROLS

To use the architecture's tracing facilities, software must provide trace fault handling procedures, perhaps interfaced with a debugging monitor. Software must also manipulate the following registers and control bits to enable the various tracing modes and enable or disable tracing in general.

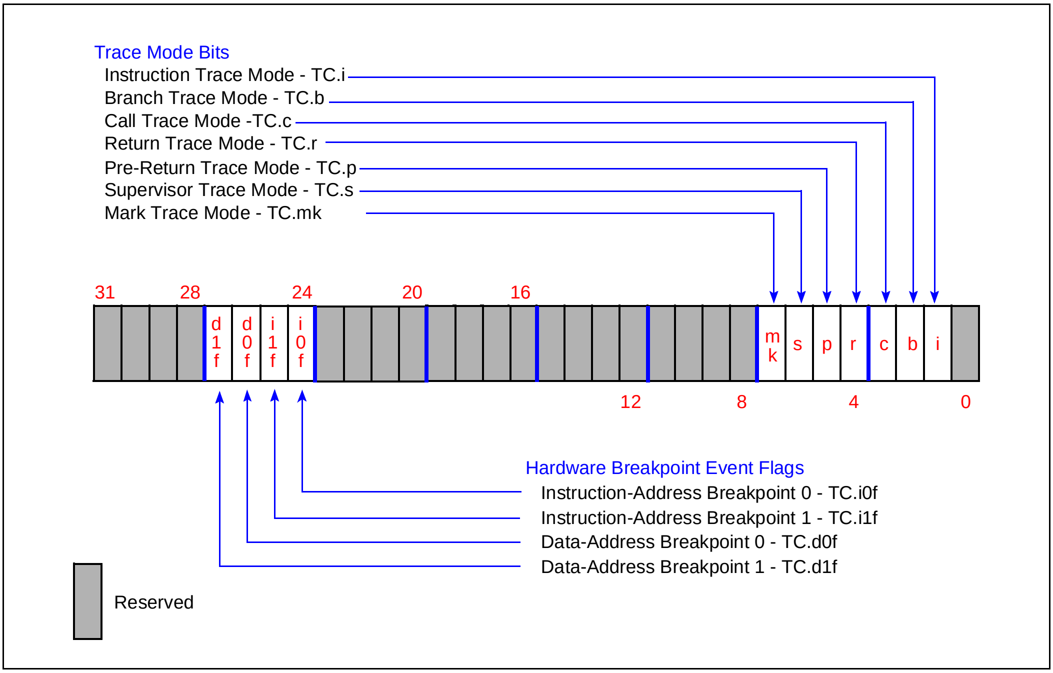
- TC register mode bits
- DAB0-DAB1 registers' address field and enable bit (in the control table)
- System procedure table supervisor-stack-pointer field trace control bit
- IPB0-IPB1 registers' address field (in the control table)
- PC register trace enable bit
- PFP register return status field prereturn trace flag (bit 3)
- BPCON register breakpoint mode bits and enable bits (in the control table)

These controls are described in the following subsections.

10.1.1 Trace Controls Register – TC

The TC register (Table 10-1) allows software to define conditions that generate trace events.

Table 10-1. 80960Rx Trace Controls Register – TC


Trace Mode Bits
Instruction Trace Mode - TC.i
Branch Trace Mode - TC.b
Call Trace Mode -TC.c
Return Trace Mode - TC.r
Pre-Return Trace Mode - TC.p
Supervisor Trace Mode - TC.s
Mark Trace Mode - TC.mk
31
28
24
20
16
12
8
4
0
d1f
d0f
i1f
i0f
mk
s
p
r
c
b
i
Hardware Breakpoint Event Flags
Instruction-Address Breakpoint 0 - TC.i0f
Instruction-Address Breakpoint 1 - TC.i1f
Data-Address Breakpoint 0 - TC.d0f
Data-Address Breakpoint 1 - TC.d1f
Reserved

The TC register contains mode bits and event flags. Mode bits define a set of tracing conditions that the processor can detect. For example, when the call-trace mode bit is set, the processor generates a trace event when a call or branch-and-link operation executes. See section 10.2 (pg. 10-3). The processor uses event flags to monitor which breakpoint trace events are generated.

A special instruction, modify-trace-controls (**modtc**), allows software to modify the TC register. On initialization, the TC register is read from the Control Table. **modtc** can then be used to set or clear trace mode bits as required. Updating TC mode bits may take up to four non-branching instructions to take effect. Software can access the breakpoint event flags using **modtc**. The processor automatically sets and clears these flags as part of its trace handling mechanism: the breakpoint event flag corresponding to the trace being serviced is set in the TC while servicing a breakpoint trace fault; the TC event flags are cleared upon return from the trace fault handler. When the program is not in a trace fault handler, or when the trace is not for breakpoints, the TC event bits are clear. On the i960 Rx I/O processor, TC register bits 0, 8 through 23 and 28 through 31 are reserved. Software must initialize these bits to zero and cannot modify them afterwards.

10.1.2 PC Trace Enable Bit and Trace-Fault-Pending Flag

The Process Controls (PC) register trace enable bit and the trace-fault-pending flag in the PC field of the fault record control tracing (see section 3.6.3, Process Controls Register – PC (pg. 3-19)). The trace enable bit enables the processor's tracing facilities; when set, the processor generates trace faults on all trace events.

Typically, software selects the trace modes to be used through the TC register. It then sets the trace enable bit to begin tracing. This bit is also altered as part of some call and return operations that the processor performs as described in section 10.5.2, Tracing on Calls and Returns (pg. 10-13).

The update of PC.te through **modpc** may take up to four non-branching instructions to take effect. The update of PC.te through call and return operations is immediate.

The trace-fault-pending flag, in the PC field of the fault record, allows the processor to remember to service a trace fault when a trace event is detected at the same time as another event (e.g., non-trace fault, interrupt). The non-trace fault event is serviced before the trace fault, and depending on the event type and execution mode, the trace-fault-pending flag in the PC field of the fault record may be used to generate a fault upon return from the non-trace fault event (see section 10.5.2.4, Tracing on Return from Implicit Call: Fault Case (pg. 10-15)).

10.2 TRACE MODES

This section defines trace modes enabled through the TC register. These modes can be enabled individually or several modes can be enabled at once. Some modes overlap, such as call-trace mode and supervisor-trace mode.

- Instruction trace
- Call trace
- Branch trace
- Return trace
- Mark trace
- Supervisor trace
- Prereturn trace

See section 10.4, HANDLING MULTIPLE TRACE EVENTS (pg. 10-12) for a description of processor function when multiple trace events occur.

10.2.1 Instruction Trace

When the instruction-trace mode is enabled in TC (TC.i = 1) and tracing is enabled in PC (PC.te = 1), the processor generates an instruction-trace fault immediately after an instruction is executed. A debug monitor can use this mode (TC.i = 1, PC.te = 1) to single-step the processor.

10.2.2 Branch Trace

When the branch-trace mode is enabled in TC (TC.b = 1) and PC.te is set, the processor generates a branch-trace fault immediately after a branch instruction executes, if the branch is taken. A branch-trace event is not generated for conditional-branch instructions that do not branch, branch-and-link instructions, and call-and-return instructions.

10.2.3 Call Trace

When the call-trace mode is enabled in TC (TC.c = 1) and PC.te is set after the call operation, the processor generates a call-trace fault when a call instruction (**call**, **callx** or **calls**) or a branch-and-link instruction (**bal** or **balx**) executes. See section 10.5.2.1, Tracing on Explicit Call (pg. 10-13) for a detailed description of call tracing on explicit instructions. Interrupt calls are never traced.

An implicit call to a fault handler also generates a call trace if TC.c and PC.te are set after the call. Refer to section 10.5.2.2, Tracing on Implicit Call (pg. 10-14) for a complete description of this case.

When the processor services a trace fault, it sets the prereturn-trace flag (PFP register bit 3) in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. The processor uses this flag to determine whether or not to signal a prereturn-trace event on a **ret** instruction.

10.2.4 Return Trace

When the return-trace mode is enabled in TC and PC.te is set after the return instruction, the processor generates a return-trace fault for a return from explicit call (PFP.rrr = 000 or PFP.rrr = 01x). See section 10.5.2.3, Tracing on Return from Explicit Call (pg. 10-15).

A return from fault may be traced and a return from interrupt cannot. See section 10.5.2.4, Tracing on Return from Implicit Call: Fault Case (pg. 10-15) and section 10.5.2.5, Tracing on Return from Implicit Call: Interrupt Case (pg. 10-15) for details.

10.2.5 Prereturn Trace

When the TC prereturn-trace mode, the PC.te, and the PFP prereturn-trace flag (PFP.p) are set, the processor generates a prereturn-trace fault prior to executing a **ret** execution. The dependence on PFP.p implies that prereturn tracing cannot be used without enabling call tracing. The processor sets PFP.p whenever it services a call-trace fault (as described above) for call-trace mode.

If another trace event occurs at the same time as the prereturn-trace event, the processor generates a fault on the non-prereturn-trace event first. Then, on a return from that fault handler, it generates a fault on the prereturn-trace event. The prereturn trace is the only trace event that can cause two successive trace faults to be generated between instruction boundaries.

10.2.6 Supervisor Trace

When supervisor-trace mode is enabled in TC and PC.te is set, the processor generates a supervisor-trace fault after either of the following:

- A call-system instruction (**calls**) executes from user mode and the procedure table entry is for a system-supervisor call.
- A **ret** instruction executes from supervisor mode and the return-type field is set to 010_2 or 011_2 (i.e., return from **calls**).

This trace mode allows a debugging program to determine kernel-procedure call boundaries within the instruction stream.

10.2.7 Mark Trace

Mark trace mode allows trace faults to be generated at places other than those specified with the other trace modes, using the **mark** instruction. It should be noted that the MARK fault subtype bit in the fault record is used to indicate a match of the instruction-address breakpoint registers or the data-address breakpoint registers as well as the **fmark** and **mark** instructions.

10.2.7.1 Software Breakpoints

mark and **fmark** allow breakpoint trace faults to be generated at specific points in the instruction stream. When mark trace mode is enabled and PC.te is set, the processor generates a mark trace fault any time it encounters a **mark** instruction. **fmark** causes the processor to generate a mark trace fault regardless of whether or not mark trace mode is enabled, provided PC.te is set. If PC.te is clear, **mark** and **fmark** behave like no-ops.

10.2.7.2 Hardware Breakpoints

The hardware breakpoint registers are provided to enable generation of trace faults on instruction execution and data access.

The i960 Rx I/O processor implements two instruction and two data address breakpoint registers, denoted IPB0, IPB1, DAB0 and DAB1. The instruction and data address breakpoint registers are 32-bit registers. The instruction breakpoint registers cause a break *after* execution of the target instruction. The DABx registers cause a break *after* the memory access has been issued to the bus controller.

Hardware breakpoint registers may be armed or disarmed. When the registers are armed, hardware breakpoints can generate an architectural trace fault. When the registers are disarmed, no action occurs, and execution continues normally. Since instructions are always word aligned, the two low-order bits of the IPBx registers act as control bits. Control bits for the DABx registers reside in the Breakpoint Control (BPCON) register. BPCON enables the data address breakpoint registers, and sets the specific modes of these registers. Hardware breakpoints are globally enabled by the process controls trace enable bit (PC.te).

The IPBx, DABx, and BPCON registers may be accessed using normal load and store instructions (except for loads from IPBx register). The application must be in supervisor mode for a legal access to occur. See section 3.3, MEMORY-MAPPED CONTROL REGISTERS (MMRs) (pg. 3-6) for more information on the address for each register.

Applications must request modification rights to the hardware breakpoint resources, before attempting to modify these resources. Rights are requested by executing the **sysctl** instruction, as described in the following section.

10.2.7.3 Requesting Modification Rights to Hardware Breakpoint Resources

Application code must always first request and acquire modification rights to the hardware breakpoint resources before any attempt is made to modify them. This mechanism is employed to eliminate simultaneous usage of breakpoint resources by emulation tools and application code. An emulation tool exercises supervisor control over breakpoint resource allocation. If the emulator retains control of breakpoint resources, none are available for application code. If an emulation tool is not being used in conjunction with the device, modification rights to breakpoint resources will be granted to the application. The emulation tool may relinquish control of breakpoint resources to the application.

If the application attempts to modify the breakpoint or breakpoint control (BPCON) registers without first obtaining rights, an OPERATION.UNIMPLEMENTED fault will be generated. In this case, the breakpoint resource will not be modified, whether accessed through a **sysctl** instruction or as a memory-mapped register.

Application code requests modification rights by executing the **sysctl** instruction and issuing the Breakpoint Resource Request message (*src1*.Message_Type = 06H). In response, the current available breakpoint resources will be returned as the *src/dst* parameter (*src/dst* must be a register). The *src2* parameter is not used. Results returned in the *src/dst* parameter must be interpreted as shown in Table 10-2.

Table 10-2. *src/dst* Encoding

***src/dst* 7:4**	***src/dst* 3:0**
Number of Available Data Address Breakpoints	Number of Available Instruction Breakpoints

NOTE: src/dst 31:8 are reserved and will always return zeroes.

The following code sample illustrates the execution of the breakpoint resource request.

```
ldconst 0x600, r4    # Load the Breakpoint Resource
                     # Request message type into r4.
sysctl r4, r4, r4    # Issue the request.
```

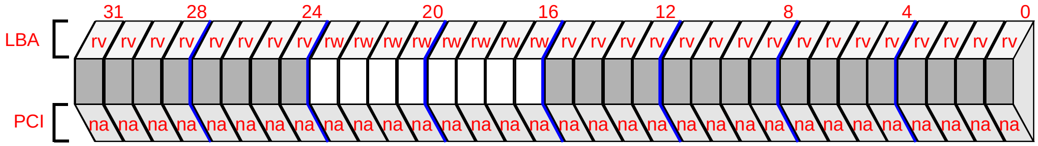
Assume in this example that after execution of the **sysctl** instruction, the value of r4 is 0000 0022H. This indicates that the application has gained modification rights to both instruction and both data address breakpoint registers. If the value returned is zero, the application has not gained the rights to the breakpoint resources.

Because the i960 Rx I/O processor does not initialize the breakpoint registers from the control table during initialization (as i960 Cx processors do), the application must explicitly initialize the breakpoint registers in order to use them once modification rights have been granted by the **sysctl** instruction.

10.2.7.4 Breakpoint Control Register – BPCON

The format of the BPCON registers are shown in Table 10-3 and Table 10-6. Each breakpoint has four control bits associated with it: two mode and two enable bits. The enable bits (DABx.e0, DABx.e1) in BPCON act to enable or disable the data address breakpoints, while the mode bits (DABx.m0, DABx.m1) dictate which type of access will generate a break event.

Table 10-3. Breakpoint Control Register – BPCON


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	8440H
PCI:	NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:24	00H	Reserved. Initialize to 0.
23	0_2	DAB1 Breakpoint Mode Control Bit: DAB1.m1
22	0_2	DAB1 Breakpoint Mode Control Bit: DAB1.m0
21	0_2	DAB1 Breakpoint Enable Control Bit: DAB1.e1
20	0_2	DAB1 Breakpoint Enable Control Bit: DAB1.e0
19	0_2	DAB0 Breakpoint Mode Control Bit: DAB0.m1
18	0_2	DAB0 Breakpoint Mode Control Bit: DAB0.m0
17	0_2	DAB0 Breakpoint Enable Control Bit: DAB0.e1
16	0_2	DAB0 Breakpoint Enable Control Bit: DAB0.e0
15:00	0000H	Reserved. Initialize to 0.

Programming the BPCON register is summarized in Table 10-4 and Table 10-5.

Table 10-4. Configuring the Data Address Breakpoint Registers – DABx

PC.te	DABx.e1	DABx.e0	Description
0	X	X	No action. With PC.te clear, breakpoints are globally disabled.
X	0	0	No action. DABx is disabled.
1	0	1	Reserved.
1	1	0	Reserved.
1	1	1	Generate a Trace Fault.

NOTE:“X” = don’t care. Reserved combinations must not be used.

The mode bits of BPCON control the type of access that generates a fault, trace message, or break event, as summarized in Table 10-5.

Table 10-5. Programming the Data Address Breakpoint Modes – DABx

DABx.m1	DABx.m0	Mode
0	0	Break on Data Write Access Only.
0	1	Break on Data Read or Data Write Access.
1	0	Break on Data Read Access.
1	1	Break on Data Read or Data Write Access.

10.2.7.5 Data Address Breakpoint Registers – DABx

The format for the Data Address Breakpoint (DAB) registers is shown in Table 10-6. Each breakpoint register contains a 32-bit address of a byte to match on.

A breakpoint is triggered when both a data access's type and address matches that specified by BPCON and the appropriate DAB register. The mode bits for each DAB register, which are contained in BPCON (see section 10.2.7.4), qualify the access types that DAB will match. An access-type match selects that DAB register to perform address checking. An address match occurs when the byte address of any of the bytes referenced by the data access matches the byte address contained within a selected DAB.

Consider the following example. DAB0 is enabled to break on any data read access and has a value of 100FH. Any of the following instructions will cause the DAB0 breakpoint to be triggered:

```
ldob  0x100f,r8
ldos  0x100e,r8
ld    0x100c,r8
ld    0x100d,r8  /* even unaligned accesses */
ldl   0x1008,r8
ldq   0x1000,r8
```

Note that the instruction:

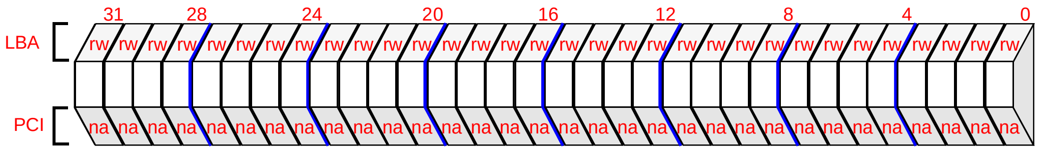
```
ldt 0x1000,r8
```

does not cause the breakpoint to be triggered because byte 100FH is not referenced by the triple word access.

Data address breakpoints can be set to break on any data read, any data write, or any data read or data write access. All accesses qualify for checking. These include explicit load and store instructions, and implicit data accesses performed by other instructions and normal processor operations.

For data accesses to the memory-mapped control register space, it is unpredictable whether breakpoint traces are generated when the access matches the breakpoints and also results in an OPERATION fault or TYPE.MISMATCH fault. The OPERATION or TYPE.MISMATCH fault will always be reported in this case.

Table 10-6. Data Address Breakpoint Register – DABx


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

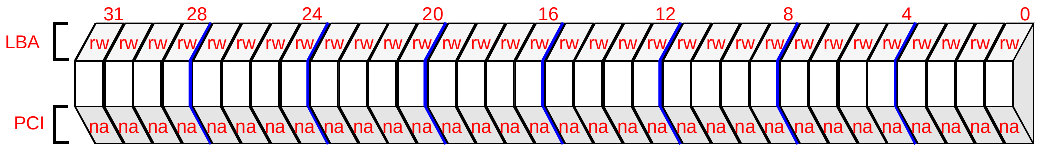
LBA:	Ch 0-8420H
	Ch 1-8424H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

PCI:	NA	
Bit	**Default**	**Description**
31:00	0000 0000H	Data Address.

10.2.7.6 Instruction Breakpoint Registers – IPBx

The format for the instruction breakpoint registers is given in Table 10-7. The upper thirty bits of the IPBx register contain the word-aligned instruction address on which to break. The two low-order bits indicate the action to take upon an address match.

Table 10-7. Instruction Breakpoint Register – IPBx


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	Ch 0-8400H
	Ch 1-8404H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

PCI:	NA	
Bit	**Default**	**Description**
31:02	0000 0000H	Instruction Address.
01	0_2	IPBX Mode: IPB1
00	0_2	IPBX Mode: IPB0

Programming the instruction breakpoint register modes is shown in Table 10-8

On the i960 Rx I/O processor, the instruction breakpoint memory-mapped registers can be read by using the **sysctl** instruction only. They can be modified by **sysctl** or by a word-length store instruction.

Storing directly to an IP breakpoint register may cause unexpected results if tracing is enabled. Any instructions in the superscalar template of a store operation that updates an IPB and any instructions in the subsequent superscalar template may trigger on the new or old value of the breakpoint register. The IP in the fault record may be that of the instruction that caused the breakpoint or may be the new value of the IPB register. The return IP in the fault record will always be correct.

If it is necessary to avoid this condition, use the modify memory-mapped control register operation of the **sysctl** instruction to update the IPB registers.

Table 10-8. Instruction Breakpoint Modes

PC.te	IPBx.m1	IPBx.m0	Action
0	X	X	No action. Globally disabled.
X	0	0	No action. IPBx disabled.
1	0	1	Reserved.
1	1	0	Reserved.
1	1	1	Generate a Trace Fault.

NOTE: “X” = don’t care. Reserved combinations must not be used.

10.3 GENERATING A TRACE FAULT

To summarize the information presented in the previous sections, the processor services a trace fault when PC.te is set and the processor detects any of the following conditions:

- An instruction included in a trace mode group executes or is about to execute (in the case of a prereturn trace event) and the trace mode for that instruction is enabled.
- A fault call operation executes and the call-trace mode is enabled.
- A **mark** instruction executes and the breakpoint-trace mode is enabled.
- An **fmark** instruction executes.
- The processor executes an instruction at an IP matching an enabled instruction address breakpoint (IPB) register.
- The processor issues a memory access matching the conditions of an enabled data address breakpoint (DAB) register.

10.4 HANDLING MULTIPLE TRACE EVENTS

With the exception of a prereturn trace event, which is always reported alone, it is possible for a combination of trace events to be reported in the same fault record. The processor may not report all events; however, it will always report a supervisor event and it will always signal at least one event.

If the processor reports prereturn trace and other trace types at the same time, it reports the other trace types in a single trace fault record first, and then services the prereturn trace fault upon return from the other trace fault.

10.5 TRACE FAULT HANDLING PROCEDURE

The processor calls the trace fault handling procedure when it detects a trace event. See section 9.7, FAULT HANDLING PROCEDURES (pg. 9-13) for general requirements for fault handling procedures. A trace fault handler must be invoked with an implicit system-supervisor call, this differs from other fault handling procedures. When the call is made, the processor clears the PC register trace enable bit (PC.te), disabling trace faults in the trace fault handler. Recall that for all other implicit or explicit system-supervisor calls, the processor replaces the trace enable bit with the system procedure table trace control bit. Clearing PC.te ensures that tracing is turned off when a trace fault handling procedure is being executed, thus preventing an endless loop of trace fault handling calls.

The processor calls the trace fault handling procedure when it detects a trace event. See section 9.7, FAULT HANDLING PROCEDURES (pg. 9-13) for general requirements for fault handling procedures.

The trace fault handling procedure is involved in a specific way and is handled differently than other faults. A trace fault handler must be invoked with an implicit system-supervisor call. When the call is made, the PC register trace enable bit is cleared. This disables trace faults in the trace fault handler. Recall that for all other implicit or explicit system-supervisor calls the trace enable bit is replaced with the system procedure table trace control bit. The exception handling of trace enable for trace faults ensures that tracing is turned off when a trace fault handling procedure is being executed. This is necessary to prevent an endless loop of trace fault handling calls.

10.5.1 Tracing and Interrupt Procedures

When the processor invokes an interrupt handling procedure to service an interrupt, it disables tracing. It does this by saving the PC register’s current state in the interrupt record, then clearing the PC register trace enable bit.

On returning from the interrupt handling procedure, the processor restores the PC register to the state it was in prior to handling the interrupt, which restores the trace enable bit. See section 10.5.2.2, Tracing on Implicit Call (pg. 10-14) and section 10.5.2.5, Tracing on Return from Implicit Call: Interrupt Case (pg. 10-15) for detailed descriptions of tracing on calls and returns from interrupts.

10.5.2 Tracing on Calls and Returns

During call and return operations, the trace enable flag (PC.te) may be altered. This section discusses how tracing is handled on explicit and implicit calls and returns.

Since all trace faults (except prereturn) are serviced after execution of the traced instruction, tracing on calls and returns is controlled by the PC.te in effect after the call or the return.

10.5.2.1 Tracing on Explicit Call

Tracing an explicit call happens before execution of the first instruction of the procedure called.

Tracing is not modified by using a **call** or **callx** instruction. Further, tracing is not modified by using a **calls** instruction from supervisor mode. When **calls** is issued from user mode, PC.te is read from the supervisor stack pointer trace enable bit (SSP.te) of the system procedure table, which is cached on chip during initialization. The trace enable bit in effect before the **calls** is stored in the new PFP[0] bit and is restored upon return from the routine (see section 10.5.2.3, Tracing on Return from Explicit Call (pg. 10-15)). The **calls** instruction and all instructions of the procedure called are traced according to the new PC.te.

Table 10-9. Tracing on Explicit Call

Call Type	Calling Procedure Trace Enable	Calling Procedure Mode	Saved PFP.rt2:0	Called Procedure Trace Enable Bit
call, callx	PC.te	user or supervisor	000_2	PC.te
calls	PC.te	supervisor	000_2	PC.te
calls	PC.te	user	$01t_2$ Stores PC.te into bit 0 of PFP.rt2:0	SSP.te

Refer to Table 7-3., Encoding of Return Status Field (pg. 7-21).

10.5.2.2 Tracing on Implicit Call

Tracing on an implicit call happens before execution of the first instruction of the non-trace fault handler called. Table 10-10 summarizes all cases of tracing on implicit call. In the table, "a" is a bit variable that symbolizes the trace enable bit in PC.

Table 10-10 summarizes all cases.

Table 10-10. Tracing on Implicit Call

Call Type	System Procedure Table Entry	Previous Frame Pointer Return Status (PFP.rt2:0)	Source PC.te	Target PC.te	PC.te Value Used for Traces on Implicit Call
00-Fault[1]	N.A.	001	a[2]	a	a
10-Fault[1]	00	001	a	a	a
10-Fault[1]	10	001	a	SSP.te	SSP.te
00-Parallel/Override Fault 00-Trace Fault	x[2]	Type of trace fault not supported			
10-Parallel/Override Fault 10-Trace Fault	00	Type of trace fault not supported			
10-Parallel/Override Fault 10-Trace Fault	10	001	a	0	0
Interrupt	N.A.	111	a	0	0

[1] * On i960® Rx I/O processor, all faults except parallel/override and trace faults.

[2] "a" and "x" are bit variables.

Tracing is not altered on the way to a local or a system-local fault handler, so the call is traced if PC.te and TC.c are set before the call. For an implicit system-supervisor call, PC.te is read from the Supervisor Stack Pointer enable bit (SSP.te). The trace on the call is serviced before execution of the first instruction of the non-trace fault handler (tracing is disabled on the way to a trace fault handler).

On the i960 Rx I/O processor, the parallel/override fault handler must be accessed through a system-supervisor call. Tracing is disabled on the way to the parallel/override fault handler.

The only type of trace fault handler supported is the system-supervisor type. Tracing is disabled on the way to the trace fault handler.

Tracing is disabled by the processor on the way to an interrupt handler, so an interrupt call is never traced.

Note that the Fault IP field of the fault record is not defined when tracing a fault call, because there is no instruction pointer associated with an implicit call.

10.5.2.3 Tracing on Return from Explicit Call

Table 10-11 shows all cases.

Table 10-11. Tracing on Return from Explicit Call

PFP.rt2:0	Execution Mode PC.em	Trace Enable Used for Trace on Return
000_2	user or supervisor	PC.te
$01a_2$	user	PC.te
$01a_2$	supervisor	t_2 (from PFP.r2:0)

Refer to Table 7-3., Encoding of Return Status Field (pg. 7-21).

For a return from local call (return type 000), tracing is not modified. For a return from system call (return type 01a, with PC.te equal to “a” before the call), tracing of the return and subsequent instructions is controlled by “a”, which is restored in the PC.te during execution of the return.

10.5.2.4 Tracing on Return from Implicit Call: Fault Case

When the processor detects several fault conditions on the same instruction (referred to as the “target”), the non-trace fault is serviced first. Upon return from the non-trace fault handler, the processor services a trace fault on the target if in supervisor mode before the return and if the trace enable and trace-fault-pending flags are set in the PC field of the non-trace fault record (at FP-16).

If the processor is in user mode before the return, tracing is not altered. The pending trace on the target instruction is lost, and the return is traced according to the current PC.te.

10.5.2.5 Tracing on Return from Implicit Call: Interrupt Case

When an interrupt and a trace fault are reported on the same instruction, the instruction completes and then the interrupt is serviced. Upon return from the interrupt, the trace fault is serviced if the interrupt handler did not switch to user mode. On the i960 Rx I/O processor, the interrupt handler returns directly to the trace fault handler.

If the interrupt return is executed from user mode, the PC register is not restored and tracing of the return occurs according to the PC.te and TC.modes bit fields.

11

INITIALIZATION AND SYSTEM REQUIREMENTS

CHAPTER 11
INITIALIZATION AND SYSTEM REQUIREMENTS

This chapter describes the steps that the i960® Rx I/O processor performs during initialization. Discussed are the reset modes, the reset state and built-in self test (BIST) features. This chapter also describes the processor's basic system requirements — including power, ground and clock — and concludes with some general guidelines for high-speed circuit board design.

11.1 OVERVIEW

The i960 Rx I/O processor initialization can basically be separated into two steps: initialization of the i960 core processor and initialization of all of the other units. Four initialization modes are available; the selected mode is determined by the values of the D/C#/RST_MODE# (hereafter called RST_MODE#) and RETRY signals when P_RST# is asserted. These modes dictate when the i960 core processor initializes and when the primary PCI interface accepts transactions.

Many of the i960 Rx I/O processor's functional units require initialization before system operation. The order in which they are initialized is important and is dependent on the system design. There is no one single initialization process for the i960 Rx I/O processor. Instead, there are several options that may be considered.

NOTE: Sample initialization code, technical notes and other developer resources are available on the Intel World Wide Web site at: http://www.intel.com.

11.1.1 Core Initialization

When the i960 core processor initialization begins, the processor uses an Initial Memory Image (IMI) to establish its state. The IMI includes:

- Initialization Boot Record (IBR) – contains the addresses of the first instruction of the user's code and the PRCB.
- Process Control Block (PRCB) – contains pointers to system data structures; also contains information used to configure the processor at initialization.
- System data structures – the processor caches several data structure pointers internally at initialization.

Software can reinitialize the processor. When a reinitialization takes place, a new PRCB and reinitialization instruction pointer are specified. Reinitialization is useful for relocating data structures from ROM to RAM after initialization.

11.1.2 General Initialization

The i960 Rx I/O processor supports several facilities to assist in system testing and start-up diagnostics. ONCE mode electrically removes the processor from a system. This feature is useful for system-level testing where a remote tester exercises the processor system. The i960 Rx I/O processor also supports JTAG boundary scan (see CHAPTER 23, TEST FEATURES). During initialization, the processor performs an internal functional self test and local bus self test. These features are useful for system diagnostics to ensure basic CPU and system bus functionality.

The processor is designed to minimize the requirements of its external system. It requires an input clock (S_CLK) and clean power and ground connections (V_{SS} and V_{CC}). Since the processor can operate at a high frequency, the external system must be designed with considerations to reduce induced noise on signals, power and ground.

11.2 80960Rx INITIALIZATION

Several functional units within the i960 Rx I/O processor must be initialized before system operation. These are the PCI-to-PCI Bridge, Address Translation Unit (ATU), i960 core processor, Memory Controller, and Secondary PCI Bus Arbiter. The order in which they are initialized is dependent on how the 80960Rx is used in the system. The initialization process begins when the Primary PCI Bus Reset signal (P_RST#) is asserted.

11.2.1 Initialization Modes

The initialization process is generally controlled through either an external host processor or the i960 core processor. Based on this assumption, there are four initialization modes.

The mode is determined by the value of the RST_MODE# and RETRY signals, described in the next sections. Table 11-1 describes the relationship between the RST_MODE# and RETRY signal values and the initialization mode.

Table 11-1. Initialization Modes

RST_MODE#	RETRY	Initialization Mode	Primary PCI Interface	i960 Core Processor
0	0	Mode 0	Accepts Transactions	Held in Reset
0	1	Mode 1	Retries All Configuration Transactions	Held in Reset
1	0	Mode 2	Accepts Transactions	Initializes
1	1	Mode 3 (default)	Retries All Configuration Transactions	Initializes

The RST_MODE# signal is sampled on the rising edge of P_RST#. The inverse value of this signal is then written to the Core Processor Reset bit in the Extended Bridge Control Register (EBCR). See CHAPTER 15, PCI-TO-PCI BRIDGE UNIT. When RST_MODE# is active and P_RST# is asserted, the i960 core processor is held in reset until P_RST# is deasserted. The i960 core processor reset is released when the reset bit in EBCR is cleared. When RST_MODE# is inactive and P_RST# is asserted, the i960 core processor is reset. The i960 core processor then begins its normal initialization sequence when P_RST# is deasserted.

The RETRY signal is sampled on the rising edge of P_RST#. The value of this signal is written to the Configuration Cycle Disable bit in the EBCR. When RETRY is active and P_RST# is de-asserted, the 80960Rx signals a Retry on all PCI configuration cycles it receives on the primary PCI bus. When RETRY is inactive and P_RST# is de-asserted, the 80960Rx accepts PCI configuration cycles on the primary PCI bus.

Figure 11-1 shows a flow chart of the initialization process.

11.2.2 Mode 0 Initialization

Mode 0 allows a host processor to configure the 80960Rx peripherals while the i960 core processor is held in reset. The host processor configures the PCI-to-PCI Bridge by assigning bus numbers, allocating PCI address space, and assigning IRQ numbers. The memory controller and ATU can also be initialized by the host processor. Program code for the i960 core processor may be downloaded into local memory by the host processor.

The host processor clears the 80960 reset signal by clearing the Core Processor Reset bit in the EBCR. This deasserts the internal reset signal on the i960 core processor and the processor begins its initialization process.

11.2.3 Mode 1 Initialization

Mode 1 allows an external agent to initialize the i960 Rx I/O processor. When P_RST# is asserted, the i960 core processor is held in reset and the host processor is prevented from initializing the PCI-to-PCI Bridge.

In this mode, an external agent has the opportunity to initialize the 80960Rx peripherals. The external agent may either clear the reset condition on the i960 core processor by clearing the Core Processor Reset bit in the EBCR or enable the PCI-to-PCI Bridge to be begin receiving configuration cycles by clearing the Configuration Cycle Disable bit in the EBCR.

11.2.4 Mode 2 Initialization

Mode 2 allows configuration cycles on the Bridge at any time and allows the i960 core processor to initialize after reset. Mode 2 allows each unit of the 80960Rx to be initialized in its own manner. All units are reset when the P_RST# signal is asserted. Each unit returns to its default state. Be aware that race conditions may exist between 80960 operation after reset and PCI configuration.

11.2.5 Mode 3 (Default Mode)

Mode 3 allows the i960 core processor to initialize and control the initialization process before the host processor is allowed to configure the 80960Rx peripherals. During this time, the primary PCI interface signals a Retry on all configuration cycles it receives until the i960 core processor clears the Configuration Cycle Disable bit in the EBCR. This option is only available when an initialization ROM is used.

By allowing the i960 core processor to control the initialization process, it is possible to initialize the PCI configuration registers to values other than the default power-up values. Certain PCI configuration registers that are read only through PCI configuration cycles are read/write from the i960 core processor. This allows the programmer to customize the way the 80960Rx appears to the PCI configuration software.

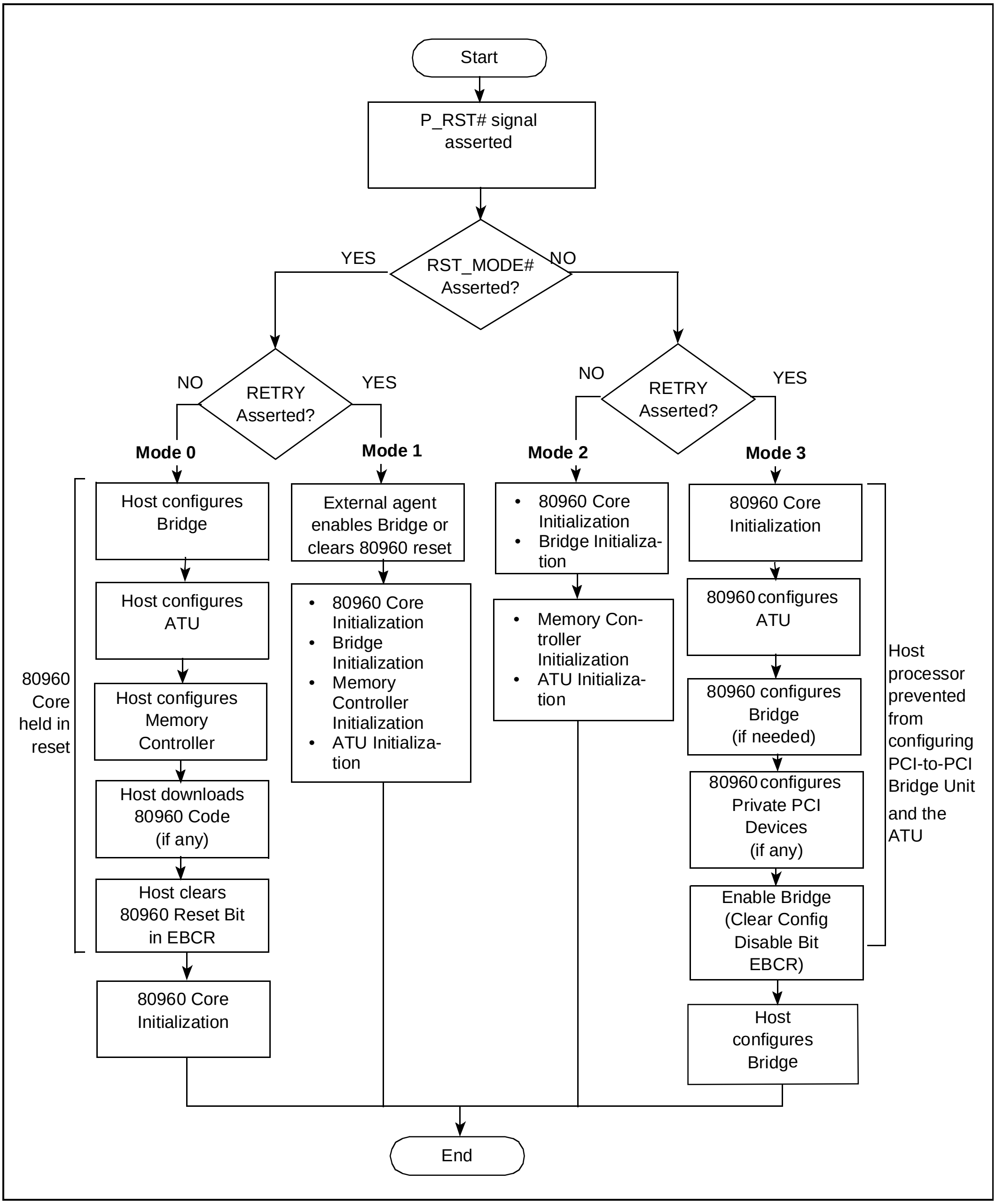

Start
P_RST# signal asserted
RST_MODE# Asserted?
YES
NO
RETRY Asserted?
NO
YES
RETRY Asserted?
NO
YES
Mode 0
Mode 1
Mode 2
Mode 3
Host configures Bridge
Host configures ATU
Host configures Memory Controller
Host downloads 80960 Code (if any)
Host clears 80960 Reset Bit in EBCR
80960 Core Initialization
80960 Core held in reset
External agent enables Bridge or clears 80960 reset
• 80960 Core Initialization
• Bridge Initialization
• Memory Controller Initialization
• ATU Initialization
• 80960 Core Initialization
• Bridge Initialization
• Memory Controller Initialization
• ATU Initialization
80960 Core Initialization
80960 configures ATU
80960 configures Bridge (if needed)
80960 configures Private PCI Devices (if any)
Enable Bridge (Clear Config Disable Bit EBCR)
Host configures Bridge
Host processor prevented from configuring PCI-to-PCI Bridge Unit and the ATU
End

Figure 11-1. Initialization Examples Flow Chart

11.2.6 Secondary PCI Bus Arbitration Unit

The Secondary PCI Bus Arbiter is enabled or disabled after reset, depending on the value of the S_REQ5#/S_ARB_EN signal. When S_REQ5#/S_ARB_EN is low on the rising edge of P_RST#, the secondary arbiter is disabled. When S_REQ5#/S_ARB_EN is high on the rising edge of P_RST#, the secondary arbiter is enabled. After reset, all devices are set to low priority, except for the secondary PCI interface of the PCI-to-PCI bridge, which is set to high priority.

The secondary bus arbiter is reset by the S_RST# signal on the secondary interface. Whenever the secondary bus is reset, the secondary arbiter is reset moving all devices to their programmed priority levels and starting the round robin arbitration sequence on the lowest number device at each priority level.

The Secondary Arbitration Control Register is reset to a value of 0000 2AA8H. This assigns the lowest priority to all external devices and assigns the secondary interface to the highest priority level.

11.2.7 Local Bus Arbitration Unit

The internal local bus arbitration logic is reset by the P_RST# signal. The reset values of the registers are shown in Table 11-2. All of the bus masters are initialized to the highest priority. None of the devices are disabled at powerup.

Table 11-2. Reset Values

Local Arbitration Register	Reset Value	Note
Local Bus Arbitration Control Register (LBACR)	0000 0000H	All Bus Masters Enabled
Local Bus Arbitration Latency Count Register (LBALCR)	0000 0FFFH	Maximum Count Value

11.2.8 Reset State Operation

The 80960Rx has two reset conditions:

- P_RST#
- L_RST#

each is described in detail in the following sections.

11.2.8.1 i960® Rx I/O Processor Reset State Operation

The P_RST# signal, when asserted, causes the i960 Rx I/O processor to enter the reset state. All external signals go to a defined state, internal logic is initialized, and certain registers are set to defined values. P_RST# is a level-sensitive, asynchronous input.

P_RST# must be asserted when power is applied to the processor. The processor then stabilizes in the reset state. This power-up reset is referred to as *cold reset*. To ensure that all internal logic has stabilized in the reset state, a valid input clock (S_CLK) and V_{CC} must be present and stable for a specified time before P_RST# can be deasserted.

The processor may also be cycled through the reset state after execution has started. This is referred to as *warm reset*. For a warm reset, P_RST# must be asserted for a minimum number of clock cycles. Specifications for a cold and warm reset can be found in the *80960RP/RD Intelligent I/O Microprocessor* Data Sheets.

While the processor's P_RST# signal is asserted, output signals are driven to the states as indicated in Table 11-2. User software cannot reset the entire i960 Rx I/O processor; however, the **sysctl** instruction can reset the i960 core processor. The P_RST# signal must be asserted to enter the reset state. See section 11.6, Reinitializing and Relocating Data Structures (pg. 11-24).

11.2.8.2 i960® Jx Core Processor Reset State Operation

The L_RST# signal, when asserted, causes the i960 core processor to enter the reset state. All core signals go to a defined state, internal core logic is initialized, and certain registers are set to defined values.

L_RST# is asserted in the EBCR when the ATU and DMA have indicated that they are off the PCI bus. L_RST# also asserts when P_RST# asserts.

L_RST# asserts after P_RST# is asserted. L_RST# deasserts after P_RST# deasserts.

11.3 i960® CORE PROCESSOR INITIALIZATION

Initialization describes the mechanism that the processor uses to establish its initial state and begin instruction execution. When i960 core processor initialization begins, the processor automatically configures itself with information specified in the IMI and performs its built-in self test based on the sampling of the STEST signal. The processor then branches to the first instruction of user code. See Figure 11-2 for a flow chart of i960 core processor initialization.

The objective of the initialization sequence is to provide a complete, working initial state when the first user instruction executes. The user's startup code needs only to perform several basic functions to place the processor in a configuration for executing application code.

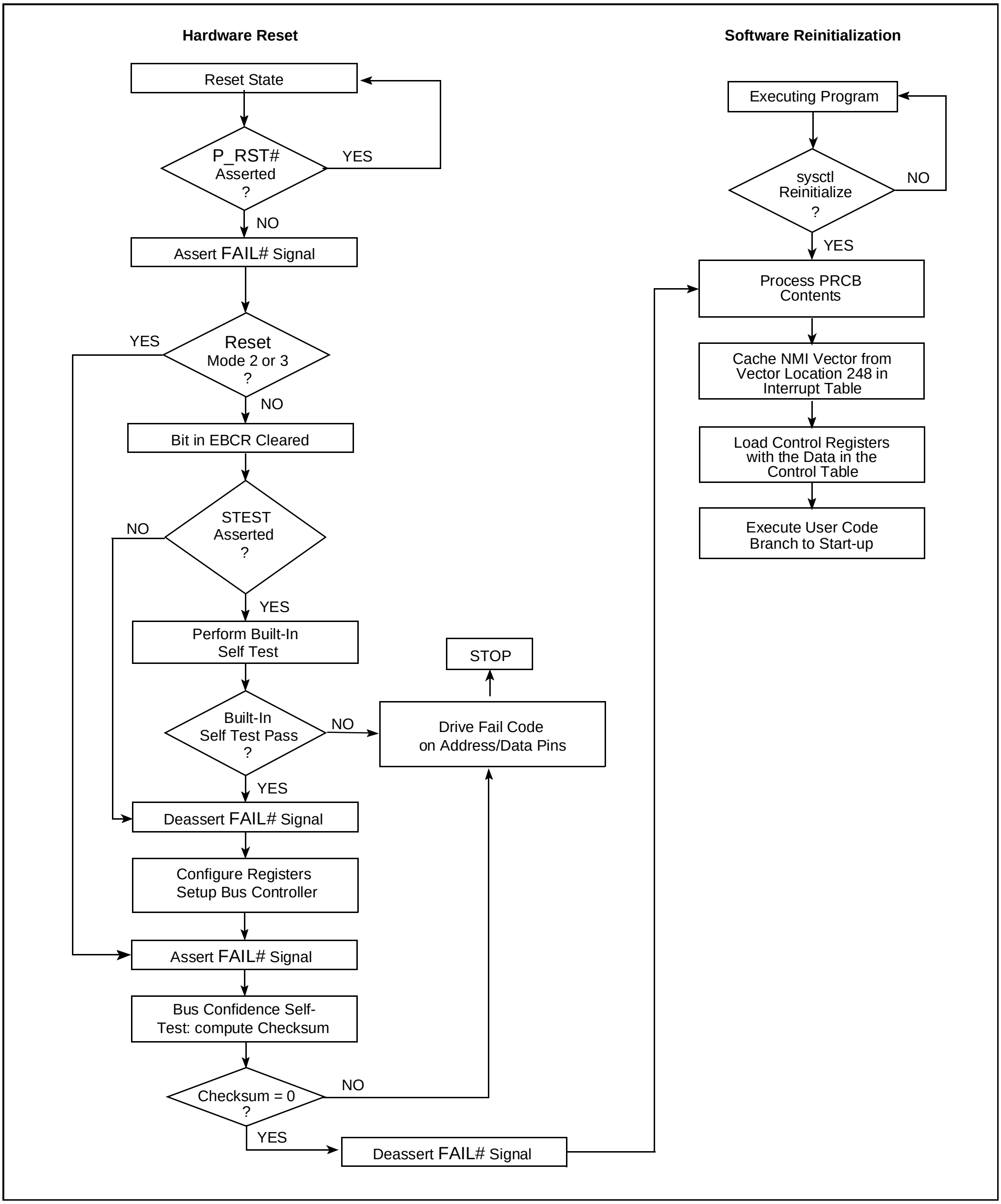

Hardware Reset
Software Reinitialization
Reset State
P_RST# Asserted ?
YES
NO
Assert FAIL# Signal
Reset Mode 2 or 3 ?
YES
NO
Bit in EBCR Cleared
STEST Asserted ?
NO
YES
Perform Built-In Self Test
Built-In Self Test Pass ?
NO
YES
STOP
Drive Fail Code on Address/Data Pins
Deassert FAIL# Signal
Configure Registers Setup Bus Controller
Assert FAIL# Signal
Bus Confidence Self-Test: compute Checksum
Checksum = 0 ?
NO
YES
Deassert FAIL# Signal
Executing Program
sysctl Reinitialize ?
NO
YES
Process PRCB Contents
Cache NMI Vector from Vector Location 248 in Interrupt Table
Load Control Registers with the Data in the Control Table
Execute User Code Branch to Start-up

Figure 11-2. Processor Initialization Flow

11.3.1 Self Test Function (STEST, FAIL#)

As part of initialization, the i960 Rx I/O processor executes a local bus confidence self test, an alignment check for data structures within the initial memory image (IMI), and optionally, a built-in self test program. The self test (STEST) signal enables or disables built-in self test. The FAIL# signal indicates that the self tests failed by asserting FAIL#. During normal operations the FAIL# signal can be asserted when a core processor error is detected. The following subsections further describe these signal functions.

Built-in self test checks basic functionality of internal data paths, registers and memory arrays on-chip. Built-in self test is not intended to be a full validation of processor functionality; it is intended to detect catastrophic internal failures and complement a user's system diagnostics by ensuring a confidence level in the processor before any system diagnostics are executed.

11.3.1.1 The STEST Signal

The STEST signal enables and disables Built-In Self Test (BIST). BIST can be disabled when the initialization time needs to be minimized or when diagnostics are simply not necessary. The STEST signal is sampled under the following conditions:

- On the rising edge P_RST#
- On the rising edge of reset mode (RST_MODE#), if used.
- On the rising edge of a local bus reset (initiated after the Reset Local Bus bit in the Extended Bridge Control Register (EBCR) is set).

When STEST is asserted, the i960 core processor executes the built-in self test. When STEST is deasserted, the i960 core processor bypasses built-in self test.

11.3.1.2 Local Bus Confidence Test

The local bus confidence test is always performed regardless of STEST signal value. The local bus confidence test reads eight words from the Initialization Boot Record (IBR) and performs a checksum on the words and the constant FFF FFFFH. The test passes only when the processor calculates a sum of zero (0). The test can detect catastrophic bus failures such as external address, data or control lines that are stuck, shorted or open.

11.3.1.3 The Fail Signal (FAIL#)

The FAIL# signal signals errors in either the built-in self test or the bus confidence self test. FAIL# is asserted (low) for each self test (Figure 11-3):

- When any test fails, the FAIL# signal remains asserted, a fail code message is driven onto the address bus, and the processor stops execution at the point of failure.

- When a core processor error occurs, FAIL# is also asserted. See section 11.3.1.4, IMI Alignment Check and Core Processor Error (pg. 11-10) for details.
- When the test passes, FAIL# is deasserted.

When FAIL# stays asserted, the only way to resume normal operation is to perform a reset operation. When the STEST signal is used to disable the built-in self test, the test does not execute; however, FAIL# still asserts at the point where the built-in self test would occur. FAIL# is deasserted after the bus confidence test passes. In Figure 11-3, all transitions on the FAIL# signal are relative to S_CLK as described in the *80960RP/RD Intelligent I/O Microprocessor* Data Sheets.

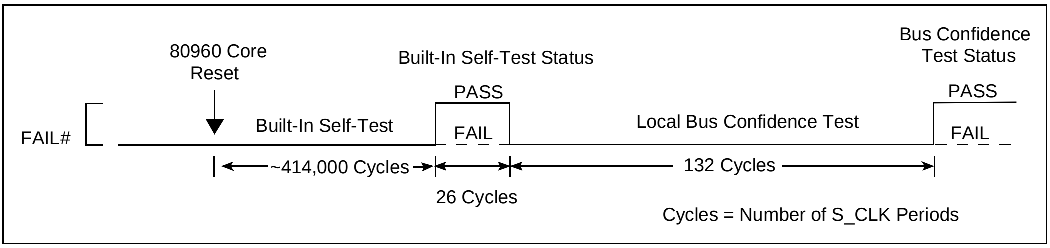


Figure 11-3. FAIL# Timing

11.3.1.4 IMI Alignment Check and Core Processor Error

The alignment check during initialization for data structures within the IMI ensures that the PRCB, control table, interrupt table, system-procedure table, and fault table are aligned to word boundaries. Normal processor operation is not possible without the alignment of these key data structures. The alignment check is one case where a core processor error could occur.

The other case of core processor error can occur during regular operation when generation of an override fault incurs a fault. The sequence of events leading up to this case is quite uncommon.

When a core processor error is detected, the FAIL# signal is asserted, a fail code message is driven onto the address bus, and the processor stops execution at the point of failure. The only way to resume normal operation of the processor is to perform a reset operation. Because core processor error generation can occur sometime after the BUS confidence test and even after initialization during normal processor operation, the FAIL# signal is a logic one before the detection of a Core PROCESSOR Error.

11.3.1.5 FAIL# Code

The processor uses only one read bus transaction to signal the fail code message; the address of the bus transaction is the fail code itself. The fail code is of the form: 0xFEFFFF*nn*; bits 6 to 0 contain a mask recording the possible failures. Bit 7, when one, indicates the mask contains failures from Built-In Self-Test (BIST); when zero, the mask indicates other failures. The fail codes are shown in Table 11-3 and Table 11-4.

Table 11-3. BIST Failure Codes

Bit	When Set
7	Set to one for BIST failure
6	On-chip Data-RAM failure detected by BIST
5	Internal Microcode ROM failure detected by BIST
4	I-cache failure detected by BIST
3	D-cache failure detected by BIST
2	Local-register cache or processor core failure detected by BIST
1	Always Zero
0	Always Zero

Table 11-4. Non-BIST Failure Codes

Bit	When Set
7	Set to zero for non-BIST failure
6	Always One; this bit does not indicate a failure
5	Always One; this bit does not indicate a failure
4	A data structure within the IMI is not aligned to a word boundary
3	A core processor error during normal operation has occurred
2	The Bus Confidence test has failed
1	Always Zero
0	Always Zero

11.4 INITIAL MEMORY IMAGE (IMI)

The IMI comprises the minimum set of data structures that the processor needs to initialize. As shown in Figure 11-4, these structures are: the initialization boot record (IBR), process control block (PRCB) and system data structures. The IBR is located at a fixed address in memory. The other components are referenced directly or indirectly by pointers in the IBR and the PRCB. The IMI performs three functions for the processor:

- Provides initial configuration information for the core and integrated peripherals.
- Provides pointers to the system data structures and the first instruction to be executed after processor initialization.
- Provides checksum words that the processor uses in its self test routine at startup.

Several data structures are typically included as part of the IMI because values in these data structures are accessed by the processor during initialization. These data structures are usually programmed in the systems's boot ROM, located in memory region 14_15 of the address space. The required data structures are:

- PRCB
- IBR
- System procedure table
- Control table
- Interrupt table
- Fault table

To ensure proper processor operation, the PRCB, system procedure table, control table, interrupt table, and fault table must not be located in architecturally reserved memory – addresses reserved for on-chip Data RAM and addresses at and above FEFF FF60H. In addition, each of these structures must start at a word-aligned address; a core processor error occurs when any of these structures are not word-aligned. See section 11.3.1.3, The Fail Signal (FAIL#) (pg. 11-9).

At initialization, the processor loads the Supervisor Stack Pointer (SSP) from the system procedure table, aligns it to a 16-byte boundary, and caches the pointer in the SSP memory-mapped control register — see section 3.3, MEMORY-MAPPED CONTROL REGISTERS (MMRs) (pg. 3-6). Recall that the supervisor stack pointer is located in the preamble of the system procedure table at byte offset 12 from the base address. The system procedure table base address is programmed in the PRCB. Consult section 7.5.1, System Procedure Table (pg. 7-15) for the format of the system procedure table.

At initialization, the NMI vector is loaded from the interrupt table and saved at location 0000 0000H of the internal data RAM. The interrupt table is typically programmed in the boot ROM and then relocated to internal RAM by reinitializing the processor.

The fault table is typically located in boot ROM. When it is necessary to locate the fault table in RAM, the processor must be reinitialized.

The remaining data structures that an application may need are the user stack, supervisor stack and interrupt stack. These stacks must be located in the 80960Rx's local bus RAM.

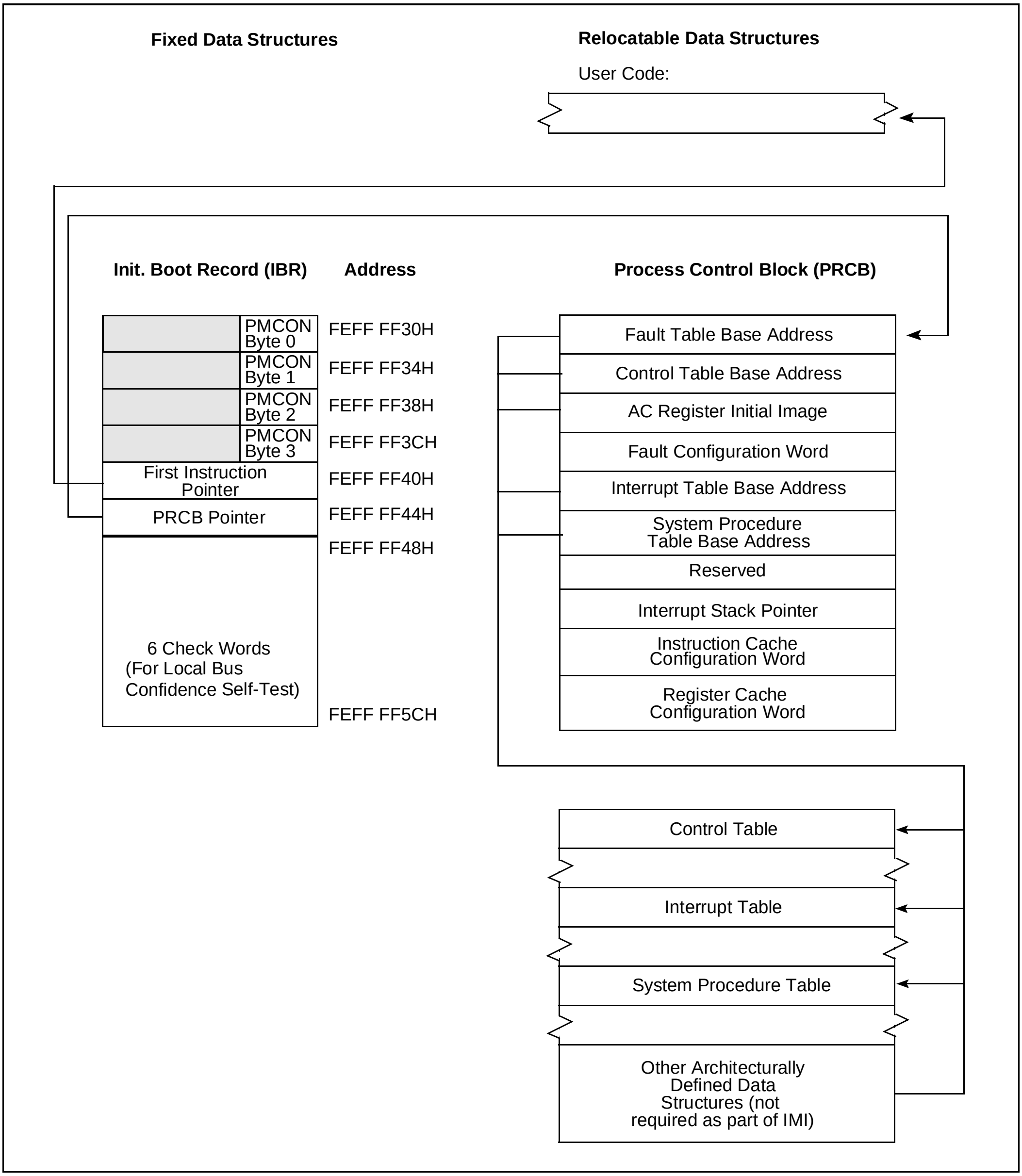

Fixed Data Structures
Relocatable Data Structures
User Code:
Init. Boot Record (IBR)
Address
Process Control Block (PRCB)
PMCON Byte 0
PMCON Byte 1
PMCON Byte 2
PMCON Byte 3
First Instruction Pointer
PRCB Pointer
6 Check Words (For Local Bus Confidence Self-Test)
FEFF FF30H
FEFF FF34H
FEFF FF38H
FEFF FF3CH
FEFF FF40H
FEFF FF44H
FEFF FF48H
FEFF FF5CH
Fault Table Base Address
Control Table Base Address
AC Register Initial Image
Fault Configuration Word
Interrupt Table Base Address
System Procedure Table Base Address
Reserved
Interrupt Stack Pointer
Instruction Cache Configuration Word
Register Cache Configuration Word
Control Table
Interrupt Table
System Procedure Table
Other Architecturally Defined Data Structures (not required as part of IMI)

Figure 11-4. Initial Memory Image (IMI) and Process Control Block (PRCB)

11.4.1 Initialization Boot Record (IBR)

The initialization boot record (IBR) is the primary data structure required to initialize the 80960Rx processor. The IBR is a 12-word structure which must be located at address FEFF FF30H (see Table 11-5). The IBR is made up of four components: the initial bus configuration data, the first instruction pointer, the PRCB pointer and the bus confidence test checksum data.

Table 11-5. Initialization Boot Record

Byte Physical Address	Description
FEFF FF30H	PMCON14_15, byte 0
FEFF FF31H to FEFF FF33H	Reserved
FEFF FF34H	PMCON14_15, byte 1
FEFF FF35H to FEFF FF37H	Reserved
FEFF FF38H	PMCON14_15, byte 2
FEFF FF39H to FEFF FF3BH	Reserved
FEFF FF3CH	PMCON14_15, byte 3
FEFF FF3DH to FEFF FF3FH	Reserved
FEFF FF40H to FEFF FF43H	First Instruction Pointer
FEFF FF44H to FEFF FF47H	PRCB Pointer
FEFF FF48H to FEFF FF4BH	Local Bus Confidence Self-Test Check Word 0
FEFF FF4CH to FEFF FF4FH	Local Bus Confidence Self-Test Check Word 1
FEFF FF50H to FEFF FF53H	Local Bus Confidence Self-Test Check Word 2
FEFF FF54H to FEFF FF57H	Local Bus Confidence Self-Test Check Word 3
FEFF FF58H to FEFF FF5BH	Local Bus Confidence Self-Test Check Word 4
FEFF FF5CH to FEFF FF5FH	Local Bus Confidence Self-Test Check Word 5

When the processor reads the IMI during initialization, it must know the bus characteristics of external memory where the IMI is located. Specifically, it must know the bus width and endianism for the remainder of the IMI. At initialization, the processor sets the PMCON register to an 8-bit bus width. The processor then needs to form the initial DLMCON and PMCON14_15 registers so that the memory containing the IBR can be accessed correctly. The lowest-order byte of each of the IBR’s first 4 words are used to form the register values. On the i960 Rx I/O processor, the bytes at FEFF FF30H and FEFF FF34H are not needed, so the processor starts fetching at address FEFF FF38. The loading of these registers is shown in the pseudo-code flow in Example 11-1.

Example 11-1. Processor Initialization Pseudocode Flow

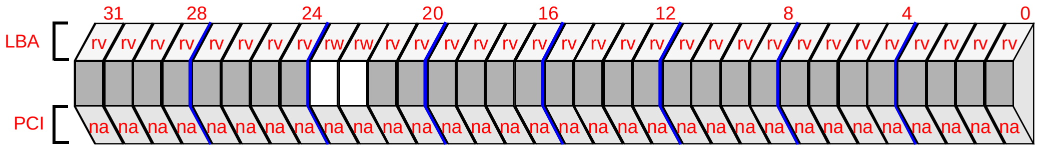
```
Processor_Initialization_flow()

{       FAIL_pin = true;
        restore_full_cache_mode; disable(I_cache); invalidate(I_cache);
        disable(D_cache); invalidate(D_cache);
        BCON.ctv = 0; /* Selects PMCON14_15 to control all accesses */
        PMCON14_15 = 0; /* Selects 8-bit bus width */

/** Exit Reset State & Start_Init **/
        if (STEST_ON_RISING_EDGE_OF_RESET)
                status = BIST();  /* BIST does not return if it fails */
        FAIL_pin = false;
        PC = 0x001f2002;     /* PC.Priority = 31, PC.em = Supervisor,*/
                             /* PC.te = 0; PC.State = Interrupted    */
        ibr_ptr = 0xfeffff30;    /* ibr_ptr used to fetch IBR words  */

/* Read PMCON14_15 image in IBR */
FAIL_pin = true;                IMSK = 0;
DLMCON.dcen = 0;        LMMR0.lmte = 0; LMMR1.lmte = 0;
PMCON14_15[byte2] = 0xc0 & memory[ibr_ptr +8];

/*Compute CheckSum on Boot Record */
carry = 0;      CheckSum = 0xffffffff;
for( i = 6; i>0; i--)        /* carry is carry out from previous add*/
        CheckSum = memory[ibr_ptr + 24 + i*4] + CheckSum + carry;
prcb_ptr = memory[ibr_ptr + 0x14];
IP = memory[prcb_ptr + 4];
CheckSum = prcb_ptr + IP + CheckSum + carry;
if(CheckSum != 0)
        {fail_msg = 0xfeffff64;         /* Fail BUS Confidence Test */
        dummy = memory[fail_msg];  /* Do load with address = fail_msg */
        for(;;);                  /* loop forever with FAIL pin true */
        }
else    FAIL_pin = false;

/* Process PRCB and Control Table */
prcb_ptr = memory[ibr_ptr + 0x14];
Process_PRCB(prcb_ptr);      /* See Process PRCB Section for Details */

Destroy_Global_&_Local_Register_Values();  /*Previous values of Global
                                               and Local Registers are
                                               Destroyed during
                                        initialization and software re-
                                                initialization*/
g0 = 80960core_device_ID;
return;                 /* Execute First Instruction */

}
```

The processor initializes the DLMCON.dcen bit to 0 to disable data caching. The remainder of the assembled word is used to initialize PMCON14_15. In conjunction with this step, the processor clears the bus control table valid bit (BCON.ctv), to ensure for the remainder of initialization that every bus request issued takes configuration information from the PMCON14_15 register, regardless of the memory region associated with the request. At a later point in initialization, the processor loads the remainder of the memory region configuration table from the external control table. The Bus Configuration (BCON) register is also loaded at this time. The control table valid (BCON.ctv) bit is then set in the control table to validate the PMCON registers after they are loaded. In this way, the bus controller is completely configured during initialization. (See CHAPTER 13, LOCAL BUS for a complete discussion of memory regions and configuring the bus controller.)

After the local bus configuration data is loaded and the new bus configuration is in place, the processor loads the remainder of the IBR which consists of the first instruction pointer, the PRCB pointer and six checksum words. The PRCB pointer and the first instruction pointer are internally cached. The six checksum words — along with the PRCB pointer and the first instruction pointer — are used in a checksum calculation which implements a confidence test of the local bus. The checksum calculation is shown in the pseudo-code flow in Example 11-2. When the checksum calculation equals zero, then the confidence test of the local bus passes.

Table 11-6 further describes the IBR organization.

Table 11-6. PMCON14_15 Register Bit Description in IBR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 8638H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:24	00H	Reserved. Initialize to 0.
23:22	00_2	Local Bus Width (BW) (00) 8-bit (01) 16-bit (10) 32-bit (11) Reserved
21:00	00 0000H	Reserved. Initialize to 0.

11.4.2 Process Control Block – PRCB

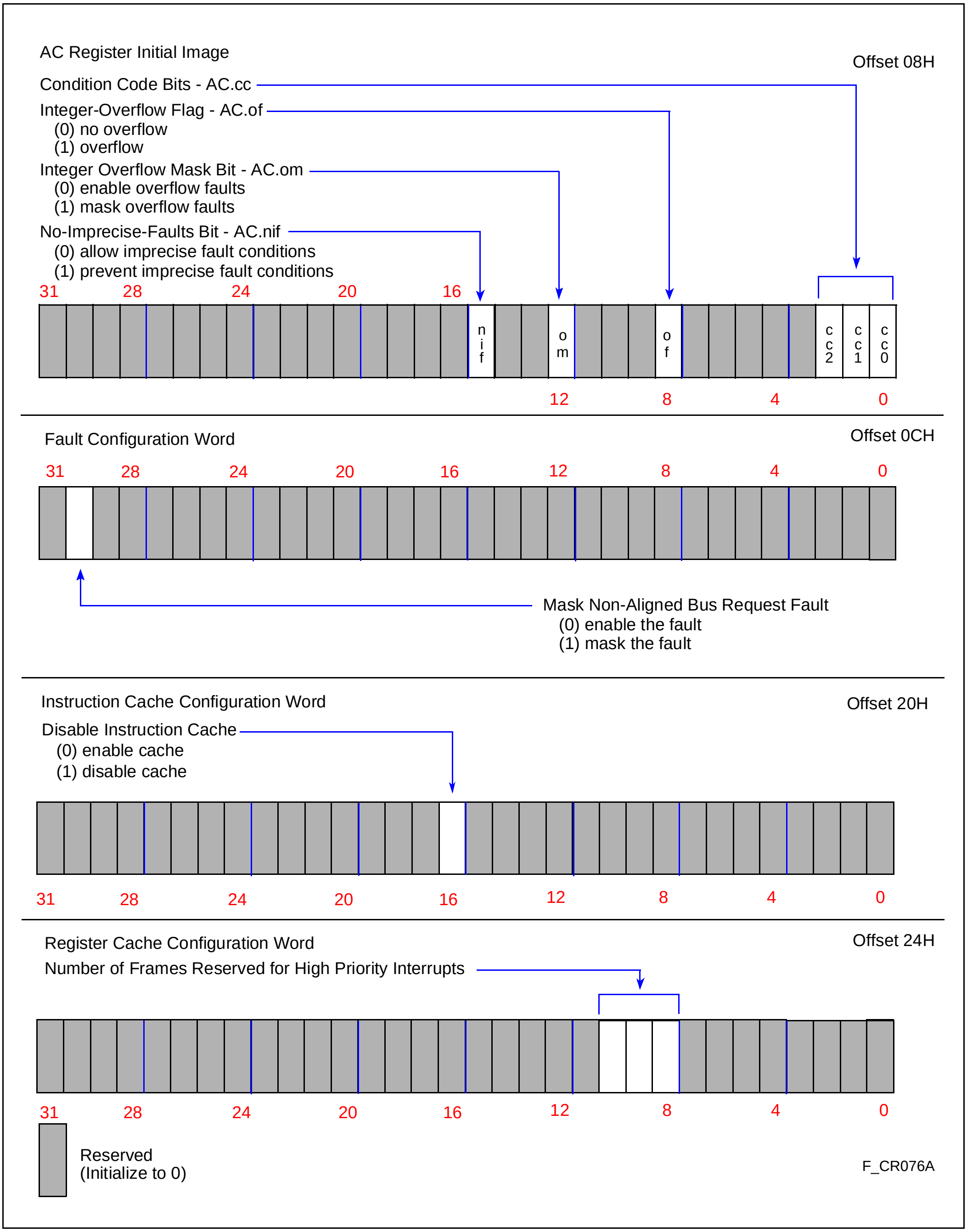
The PRCB contains base addresses for system data structures and initial configuration information for the i960 core processor. The base addresses are accessed from these internal registers. The registers are accessible to the users through the memory mapped interface. Upon reset or reinitialization, the registers are initialized. The PRCB format is shown in Table 11-7.

Table 11-7. PRCB Configuration

Physical Address	Description
PRCB POINTER + 00H	Fault Table Base Address
PRCB POINTER + 04H	Control Table Base Address
PRCB POINTER + 08H	AC Register Initial Image
PRCB POINTER + 0CH	Fault Configuration Word
PRCB POINTER + 10H	Interrupt Table Base Address
PRCB POINTER + 14H	System Procedure Table Base Address
PRCB POINTER + 18H	Reserved
PRCB POINTER + 1CH	Interrupt Stack Pointer
PRCB POINTER + 20H	Instruction Cache Configuration Word
PRCB POINTER + 24H	Register Cache Configuration Word

The initial configuration information is programmed in the arithmetic controls register (AC) initial image, the fault configuration word, the instruction cache configuration word, and the register cache configuration word. Table 11-8 show these configuration words.

Table 11-8. Process Control Block Configuration Words


AC Register Initial Image
Offset 08H
Condition Code Bits - AC.cc
Integer-Overflow Flag - AC.of
(0) no overflow
(1) overflow
Integer Overflow Mask Bit - AC.om
(0) enable overflow faults
(1) mask overflow faults
No-Imprecise-Faults Bit - AC.nif
(0) allow imprecise fault conditions
(1) prevent imprecise fault conditions
31
28
24
20
16
nif
om
of
cc2
cc1
cc0
12
8
4
0
Fault Configuration Word
Offset 0CH
31
28
24
20
16
12
8
4
0
Mask Non-Aligned Bus Request Fault
(0) enable the fault
(1) mask the fault
Instruction Cache Configuration Word
Offset 20H
Disable Instruction Cache
(0) enable cache
(1) disable cache
31
28
24
20
16
12
8
4
0
Register Cache Configuration Word
Offset 24H
Number of Frames Reserved for High Priority Interrupts
31
28
24
20
16
12
8
4
0
Reserved
(Initialize to 0)
F_CR076A

11.4.3 Process PRCB Flow

The following pseudo-code flow illustrates the processing of the PRCB. Note that this flow is used for both initialization and reinitialization (through **sysctl**).

Example 11-2. PRCB Processing Pseudo-code Flow

```
Process_PRCB(prcb_ptr)
{       PRCB_mmr = prcb_ptr;
        reset_state(data_ram);  /* It is unpredictable whether the   */
                                /* Data RAM keeps its prior contents */
        fault_table  =  memory[PRCB_mmr];
        ctrl_table   =  memory[PRCB_mmr+0x4];
        AC           =  memory[PRCB_mmr+0x8];
        fault_config =  memory[PRCB_mmr+0xc];
        if (1 & (fault_config >> 30))
generate_fault_on_unaligned_access = false;
        else    generate_fault_on_unaligned_access = true;

/** Load Interrupt Table Pointer **/
        Reset_block_NMI;
        interrupt_table =  memory[PRCB_mmr+0x10];

/** Load System Procedure Table Pointer **/
        sysproc = memory[PRCB_mmr+0x14];

/** Initialize ISP, FP, SP, and PFP **/
        ISP_mmr =  memory[PRCB_mmr+0x1c];
        FP      = ISP_mmr;
        SP      = FP + 64;
        PFP     = FP;

/** Initialize Instruction Cache **/
        ICCW = memory[PRCB_mmr+0x20];
        if (1 & (ICCW >> 16) ) enable(I_cache);

/** Cache NMI Vector Entry in Data RAM**/
        memory[0] = memory[interrupt_table + (248*4) + 4];

/** Process System Procedure Table **/
        temp    = memory[sysproc+0xc];
        SSP_mmr = (~0x3) & temp;
        SSP.te  = 1 & temp;

/** Configure Local Register Cache **/
        programmed_limit = (7 & (memory[PRCB_mmr+0x24] >> 8) );
        config_reg_cache( programmed_limit );

/** Load_control_table. Note breakpoints and BPCON are excluded here **/
        load_control_table(ctrl_table+0x10 , ctrl_table+0x58);
                  /* Load ctrl_table+0x10  through ctrl_table+0x58 */
        load_control_table(ctrl_table+0x68 , ctrl_table+0x6c);
                /* Load ctrl_table+0x68  through ctrl_table+0x6C */
        IBP0 = 0x0; IBP1 = 0x0; DAB0 = 0x0; DAB1 = 0x0;

/** Initialize Timers **/
        TMR0.tc   = 0; TMR1.tc   = 0; TMR0.enable = 0; TMR1.enable = 0;
        TMR0.sup  = 0; TMR1.sup  = 0; TMR0.reload = 0; TMR1.reload = 0;
        TMR0.csel = 0; TMR1.csel = 0;

        return;
```

11.4.3.1 AC Initial Image

The AC initial image is loaded into the on-chip AC register during initialization. The AC initial image allows the initial value of the overflow mask, no imprecise faults bit and condition code bits to be selected at initialization.

The AC initial image condition code bits can be used to specify the source of an initialization or reinitialization when a single instruction entry point to the user start-up code is desirable. This is accomplished by programming the condition code in the AC initial image to a different value for each different entry point. The user start-up code can detect the condition code values — and thus the source of the reinitialization — by using the compare or compare-and-branch instructions.

11.4.3.2 Fault Configuration Word

The fault configuration word allows the operation-unaligned fault to be masked when an unaligned memory request is issued. When an unaligned access is encountered, the processor *always* performs the access. After performing the access, the processor determines whether it should generate a fault. When bit 30 in the fault configuration word is set, a fault is not generated after an unaligned memory request is performed. When bit 30 is clear, a fault is generated after an unaligned memory request is performed.

11.4.3.3 Instruction Cache Configuration Word

The instruction cache configuration word allows the instruction cache to be enabled or disabled at initialization. When bit 16 in the instruction cache configuration word is set, the instruction cache is disabled and all instruction fetches are directed to external memory. Disabling the instruction cache is useful for tracing execution in a software debug environment.

The instruction cache remains disabled until the following operations:

- The processor is reinitialized with a new value in the instruction cache configuration word
- **icctl** is issued with the enable instruction cache operation
- **sysctl** is issued with the configure instruction cache message type and a cache configuration mode other than disable cache.

11.4.3.4 Register Cache Configuration Word

The register cache configuration word specifies the number of free frames in the local register cache that can be used by critical code (i.e., code that is in the interrupted state and has a process priority greater than or equal to 28).

The register cache and the configuration word are explained further in section 4.2, LOCAL REGISTER CACHE (pg. 4-2).

11.4.4 Control Table

The control table is the data structure that contains the on-chip control registers values. It is automatically loaded during initialization and must be completely constructed in the IMI. Figure 11-5 shows the Control Table format.

For register bit definitions of the on-chip control table registers, see the following:

- IMAP — Table 8-11 through Table 8-13, Interrupt Map Register 2 – IMAP2 (pg. 8-36)
- ICON — Table 8-10. Interrupt Control Register – ICON (pg. 8-34)
- PMCON — Table 12-2. Physical Memory Control Registers – PMCON0:15 (pg. 12-5)
- TC — Table 10-1. 80960Rx Trace Controls Register – TC (pg. 10-2)
- BCON — Table 12-3. Bus Control Register Bit Definitions – BCON (pg. 12-6)

31 0	
Reserved (Initialize to 0)	00H
Reserved (Initialize to 0)	04H
Reserved (Initialize to 0)	08H
Reserved (Initialize to 0)	0CH
Interrupt Map 0 (IMAP0)	10H
Interrupt Map 1 (IMAP1)	14H
Interrupt Map 2 (IMAP2)	18H
Interrupt Configuration (ICON)	1CH
Physical Memory Region 0:1 Configuration (PMCON0_1)	20H
Reserved (Initialize to 0)	24H
Physical Memory Region 2:3 Configuration (PMCON2_3)	28H
Reserved (Initialize to 0)	2CH
Physical Memory Region 4:5 Configuration (PMCON4_5)	30H
Reserved (Initialize to 0)	34H
Physical Memory Region 6:7 Configuration (PMCON6_7)	38H
Reserved (Initialize to 0)	3CH
Physical Memory Region 8:9 Configuration (PMCON8_9)	40H
Reserved (Initialize to 0)	44H
Physical Memory Region 10:11 Configuration (PMCON10_11	48H
Reserved (Initialize to 0)	4CH
Physical Memory Region 12:13 Configuration (PMCON12_13)	50H
Reserved (Initialize to 0)	54H
Physical Memory Region 14:15 Configuration (PMCON14_15)	58H
Reserved (Initialize to 0)	5CH
Reserved (Initialize to 0)	60H
Reserved (Initialize to 0)	64H
Trace Controls (TC)	68H
Bus Configuration Control (BCON)	6CH

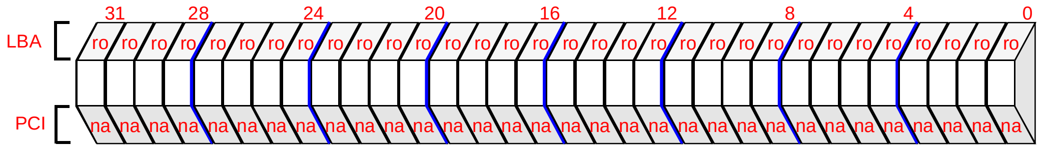
Figure 11-5. Control Table

11.5 DEVICE IDENTIFICATION ON RESET

During the manufacturing process, values characterizing the i960 Rx I/O processor type and stepping are programmed into the memory-mapped registers. The i960 Rx I/O processor contains two read-only device ID MMRs. One holds the Processor Device ID (PDIDR) and the other holds the i960 Core Processor Device ID (DEVICEID).

The device identification values are compliant with the IEEE 1149.1 specification and Intel standards. Table 11-9 and Table 11-10 describe the fields of the two Device IDs.

Table 11-9. Processor Device ID Register - PDIDR

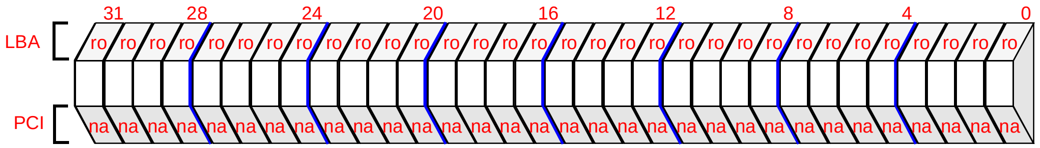

31
28
24
20
16
12
8
4
0
LBA
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1710H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write
			
			PCI = PCI Configuration Address Offset

Bit	Default	Description
31:28	X	Version - Indicates stepping changes.
27	X	V_{CC} - Indicates device voltage type. 0=5.0V 1=3.3V
26:21	X	Product Type - Indicates the generation or “family member”.
20:17	X	Generation Type - Indicates the generation of the device.
16:12	X	Model Type - Indicates member within a series and specific model information.
11:01	X	Manufacturer ID - Indicates manufacturer ID assigned by IEEE. 0000 0001 001=Intel Corporation
0	1	Constant

NOTE: The values programmed into this registers varies with stepping. Refer to the *i960® Rx I/O Processor Specification Update (272918)* for the correct value

Table 11-10. i960® Core Processor Device ID Register - DEVICEID


31
28
24
20
16
12
8
4
0
LBA
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: FF00 8710H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:28	X	Version - Indicates stepping changes.
27	X	V_{CC} - Indicates device voltage type. 0=5.0V 1=3.3V
26:21	X	Product Type - Indicates the generation or “family member”.
20:17	X	Generation Type - Indicates the generation of the device.
16:12	X	Model Type - Indicates member within a series and specific model information.
11:01	X	Manufacturer ID - Indicates manufacturer ID assigned by IEEE. 0000 0001 001=Intel Corporation
0	1	Constant

NOTE: The values programmed into this registers varies with stepping. Refer to the *i960® Rx I/O Processor Specification Update (272918)* for the correct value

11.6 Reinitializing and Relocating Data Structures

Reinitialization can reconfigure the processor and change pointers to data structures. The processor is reinitialized by issuing the **sysctl** instruction with the reinitialize processor message type. See 6.2.67, sysctl (pg. 6-114) for a description of **sysctl**.) The reinitialization instruction pointer and a new PRCB pointer are specified as operands to the **sysctl** instruction. When the processor is reinitialized, the fields in the newly specified PRCB are loaded as described in section 11.4.2, Process Control Block – PRCB (pg. 11-17).

Reinitialization is useful for relocating data structures to RAM after initialization. The interrupt table must be located in RAM: to post software-generated interrupts, the processor writes to the pending priorities and pending interrupts fields in this table. It may also be necessary to relocate the control table to RAM: it must be in RAM when the control register values are to be changed by user code. In some systems, it is necessary to relocate other data structures (fault table and system procedure table) to RAM because of unsatisfactory load performance from ROM.

After initialization, the software is responsible for copying data structures from ROM into RAM. The processor is then reinitialized with a new PRCB which contains the base addresses of the new data structures in RAM.

The processor caches the following pointers during its initialization. To modify these data structures, a software re-initialization is needed.

- Interrupt Table Address
- Fault Table Address
- System Procedure Table Address
- Control Table Address

11.7 SYSTEM REQUIREMENTS

The following sections discuss generic hardware requirements for a system built around the i960 Rx I/O processor. This section describes electrical characteristics of the processor's interface to the external circuit, including the S_CLK, P_RST#, STEST, FAIL#, ONCE#, V_{SS} and V_{CC} signals. Specific signal functions for the external bus signals and interrupt inputs are discussed in their respective sections in this manual.

11.7.1 Clocking

The i960 Rx I/O processor has a single clock input (S_CLK) for control. All input/output timings are relative to S_CLK.

The range of operation for all PCI clocks is 0 to 33 MHz. The i960 Rx I/O processor has an internal PLL that limits the range of processor clock operation from 16 MHz to 33 MHz. When the minimum frequency is not met, the internal status of the processor is not guaranteed.

The clock input is designed to be driven by most common TTL crystal clock oscillators. The clock input must be free of noise and conform with the specifications listed in the 80960Rx Data Sheets. S_CLK input capacitance is minimal; for this reason, it may be necessary to terminate the S_CLK circuit board traces at the processor to reduce overshoot and undershoot.

11.7.2 Output Clocks

The i960 Rx I/O processor supports an I^2C bus interface. The output clock frequency for I^2C operation is 100 KHz or 400 KHz. This clock is generated from the i960 core processor clock. To use the I^2C interface, a clock divider value must be written into the I^2C Clock Count Register. See section 21.10.5, I^2C Clock Count Register – ICCR (pg. 21-27).

11.7.3 Reset

There are multiple ways to reset the i960 Rx I/O processor. Reset is controlled either through external signals or control registers.

When the primary PCI bus reset signal P_RST# is asserted, the i960 Rx I/O processor:

- Asserts the secondary PCI bus reset signal S_RST#**.**
- Resets the i960 core processor and the local bus.
- Resets all internal units, including the PCI-to-PCI bridge unit.
- Asserts local bus reset.

Reset is also available through the Bridge Control registers in the PCI to PCI Bridge Unit:

- The Secondary Bus Reset bit in the Bridge Control Register resets the secondary PCI bus by asserting the secondary PCI reset signal S_RST#**.**
    - The PCI to PCI Bridge Unit resets its posting buffers and address queues and the secondary PCI bus interface, but not the PCI configuration registers or its primary PCI interface.
    - DMA Channel 2 immediately halts any PCI transactions and gracefully completes any local bus transactions. It then returns to an idle state. DMA Channel 2 does not begin any new transfers until the Secondary Bus Reset bit is cleared.
    - Secondary ATU immediately halts any PCI transactions and gracefully completes any local bus transactions. The i960 core processor is released from back-off, when necessary. The Secondary ATU does not accept any new i960 core processor requests until the Secondary Bus Reset bit is cleared.
    - The software must clear this bit.
- The Reset Local Bus bit in the Extended Bridge Control Register (EBCR) resets the i960 core processor and all units on the local bus. Before reset, the DMA channels and the ATUs halt all PCI bus transactions. Software must ensure that the I^2C bus and the APIC bus are idle before the reset occurs. The i960 core processor may or may not be held in reset when the reset local bus bit is cleared by software. This depends on the default value of the Core Processor Reset bit in the EBCR. The local bus reset does not reset the PCI-to-PCI bridge unit or its configuration registers. All other configuration registers are reset.

See CHAPTER 15, PCI-TO-PCI BRIDGE UNIT for a full description of the Bridge Control Register and the Extended Bridge Control Register.

11.7.4 Power and Ground Requirements (V_{CC}, V_{SS})

The large number of V_{SS} and V_{CC} signal effectively reduces the impedance of power and ground connections to the chip and reduces transient noise induced by current surges. The i960 Rx I/O processor is implemented in CHMOS IV technology. Unlike NMOS processes, power dissipation in the CHMOS process is due to capacitive charging and discharging on-chip and in the processor's output buffers; there is almost no DC power component. The nature of this power consumption results in current surges when capacitors charge and discharge. The processor's power consumption depends mostly on frequency. It also depends on voltage and capacitive bus load (see the 80960Rx Data Sheets).

To reduce clock skew internal to the i960 Rx I/O processor, the V_{CCPLL} pins for the Phase Lock Loop (PLL) circuits are isolated on the pinout. The lowpass filter, as shown in Figure 11-6, reduces noise induced clock jitter and its effects on timing relationships in system designs. The 0.01 µF capacitor must be of the type X7R and the node connecting V_{CCPLL} must be as short as possible.

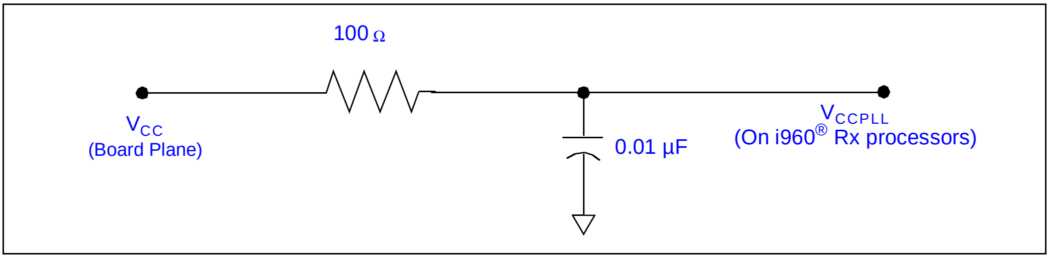


Figure 11-6. V_{CCPLL} Lowpass Filter

11.7.5 Power and Ground Planes

Power and ground planes must be used in i960 Rx I/O processor systems to minimize noise. Justification for these power and ground planes is the same as for multiple V_{SS} and V_{CC} pins. Power and ground lines have inherent inductance and capacitance; therefore, an impedance $Z=(L/C)^{1/2}$.

Total characteristic impedance for the power supply can be reduced by adding more lines. This effect is illustrated in Figure 11-7, which shows that two lines in parallel have half the impedance of one. Ideally, a plane, an infinite number of parallel lines, results in the lowest impedance. Fabricate power and ground planes with a 1 oz. copper for outer layers and 0.5 oz. copper for inner layers.

All power and ground pins must be connected to the planes. Ideally, the i960 Rx I/O processor should be located at the center of the board to take full advantage of these planes, simplify layout and reduce noise.

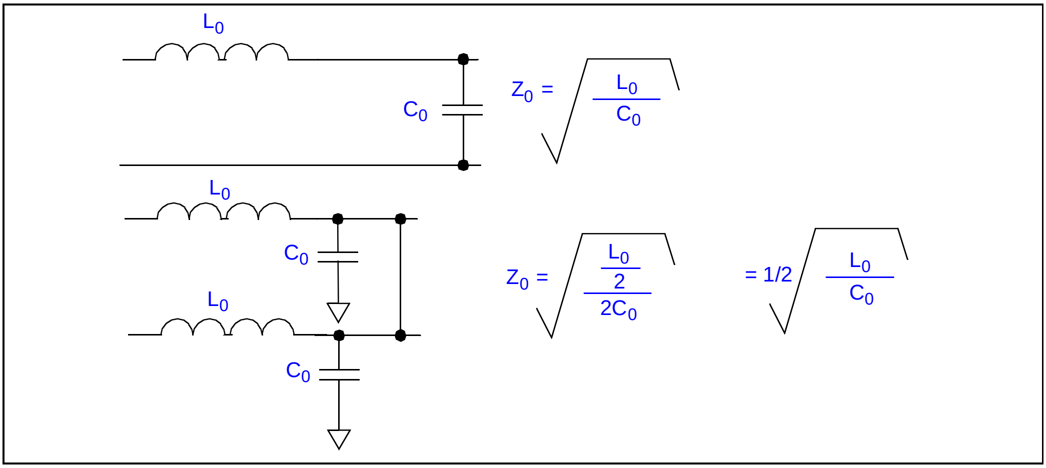

L_0
C_0
$Z_0 = \sqrt{\frac{L_0}{C_0}}$
L_0
C_0
L_0
C_0
$Z_0 = \sqrt{\frac{\frac{L_0}{2}}{2C_0}}$
$= 1/2 \sqrt{\frac{L_0}{C_0}}$

Figure 11-7. Reducing Characteristic Impedance

11.7.6 Decoupling Capacitors

Decoupling capacitors placed across the processor between V_{CC} and V_{SS} reduce voltage spikes by supplying the extra current needed during switching. Place these capacitors close to the device because connection line inductance negates their effect. Also, for this reason, the capacitors should be low inductance. Chip capacitors (surface mount) exhibit lower inductance.

11.7.7 High Frequency Design Considerations

At high signal frequencies and/or with fast edge rates, the transmission line properties of signal paths in a circuit must be considered. Transmission line effects and crosstalk become significant in comparison to the signals. These errors can be transient and therefore difficult to debug. In this section, some high-frequency design issues are discussed; for more information, consult a reference on high-frequency design.

11.7.8 Line Termination

Input voltage level violations are usually due to voltage spikes that raise input voltage levels above the maximum limit (overshoot) and below the minimum limit (undershoot). These voltage levels can cause excess current on input gates, resulting in permanent damage to the device. Even when no damage occurs, many devices are not guaranteed to function as specified when input voltage levels are exceeded.

Signal lines are terminated to minimize signal reflections and prevent overshoot and undershoot. Terminate the line when the round-trip signal path delay is greater than signal rise or fall time. When the line is not terminated, the signal reaches its high or low level before reflections have time to dissipate and overshoot or undershoot occurs.

For the i960 Rx I/O processor, two termination methods are recommended: AC and series. An AC termination matches the impedance of the trace, there by eliminating reflections due to the impedance mismatch.

Series termination decreases current flow in the signal path by adding a series resistor as shown in Figure 11-8. The resistor increases signal rise and fall times so that the change in current occurs over a longer period of time. Because the amount of voltage overshoot and undershoot depends on the change in current over time (V = L di/dt), the increased time reduces overshoot and undershoot. Place the series resistor as close as possible to the signal source. AC termination is effective in reducing signal reflection (ringing). This termination is accomplished by adding an RC combination at the signal's farthest destination (Figure 11-9). While the termination provides no DC load, the RC combination damps signal transients.

Selection of termination methods and values is dependent upon many variables, such as output buffer impedance, board trace impedance and input impedance.

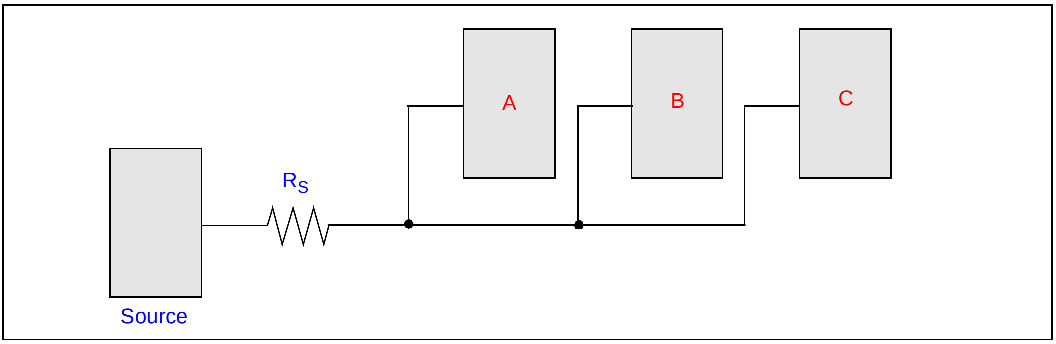


Figure 11-8. Series Termination

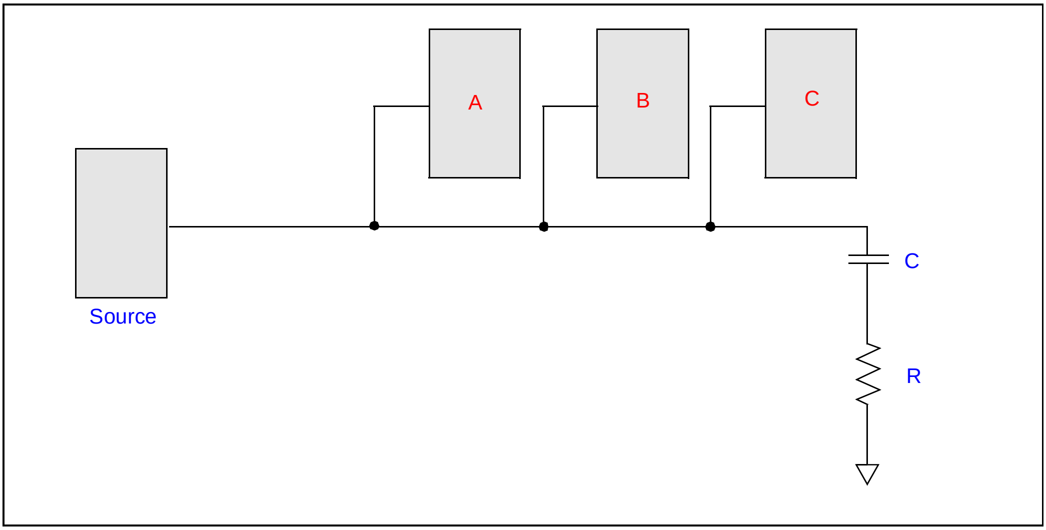


Figure 11-9. AC Termination

11.7.9 Latchup

Latchup is a condition in a CMOS circuit in which V_{CC} becomes shorted to V_{SS}. Intel's CMOS IV processes are immune to latchup under normal operation conditions. Latchup can be triggered when the voltage limits on I/O pins are exceeded, causing internal PN junctions to become forward biased.

The following guidelines help prevent latchup:

- Observe the maximum rating for input voltage on I/O pins.
- Never apply power to an i960 Rx I/O processor signal or a device connected to an i960 Rx I/O processor signal before applying power to the i960 Rx I/O processor itself.
- Prevent overshoot and undershoot on I/O pins by adding line termination and by designing to reduce noise and reflection on signal lines.

11.7.10 Interference

Interference is the result of electrical activity in one conductor that causes transient voltages to appear in another conductor. Interference increases with the following factors:

- Frequency Interference is the result of changing currents and voltages. The more frequent the changes, the greater the interference.

- Closeness-of-conductors Interference is due to electromagnetic and electrostatic fields whose effects are weaker further from the source.

Two types of interference must be considered in high frequency circuits: electromagnetic interference (EMI) and electrostatic interference (ESI).

EMI is caused by the magnetic field that exists around any current-carrying conductor. The magnetic flux from one conductor can induce current in another conductor, resulting in transient voltage. Several precautions can minimize EMI:

- Run ground lines between two adjacent lines wherever they traverse a long section of the circuit board. The ground line should be grounded at both ends.
- Run ground lines between the lines of an address bus or a data bus when either of the following conditions exist:
    - The bus is on an external layer of the board.
    - The bus is on an internal layer but not sandwiched between power and ground planes that are at most 10 mils away.

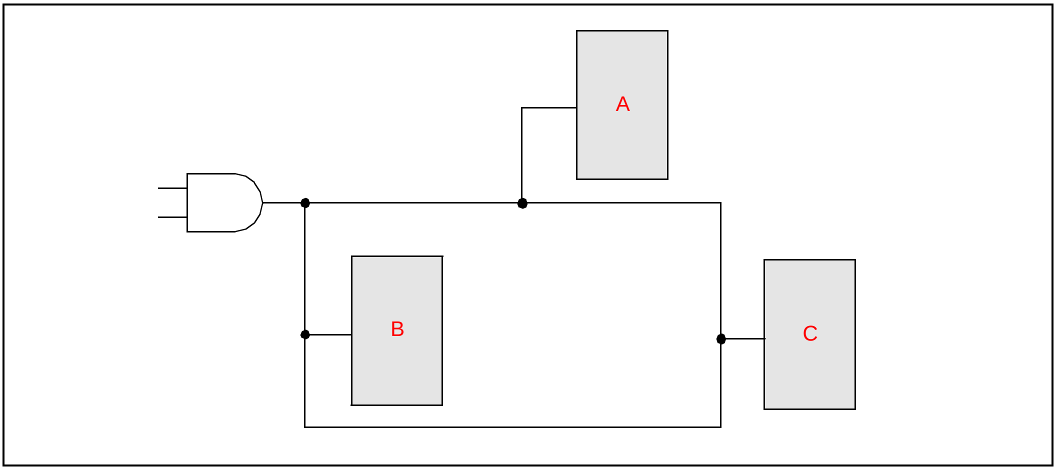


Figure 11-10. Avoid Closed-Loop Signal Paths

ESI is caused by the capacitive coupling of two adjacent conductors. The conductors act as the plates of a capacitor; a charge built up on one induces the opposite charge on the other.

The following steps reduce ESI:

- Separate signal lines so that capacitive coupling becomes negligible.
- Run a ground line between two lines to cancel the electrostatic fields.

12

CORE PROCESSOR LOCAL BUS CONFIGURATION

CHAPTER 12
CORE PROCESSOR
LOCAL BUS CONFIGURATION

This chapter provides information on setting the memory-mapped registers that configure the local memory bus. Topics include setting address ranges for different types of memory and configuring the bus width. This chapter also details enabling/disabling data caching for a memory region.

12.1 MEMORY ATTRIBUTES

Every location in memory has associated physical and logical attributes. For example, a specific location may have the following attributes:

- ***Physical:*** Memory is an 8-bit wide ROM
- ***Logical:*** Data is non-cacheable

In the example above, physical attributes correspond to those parameters that indicate *how to physically access the data*. The BCU uses physical attributes to determine the local bus protocol and signal pins to use when controlling the memory subsystem. The logical attributes tell the BCU how to interpret, format and control interaction of on-chip data caches. The physical and logical attributes for an individual location are independently programmable.

12.1.1 Physical Memory Attributes

The only programmable physical memory attribute for the i960® Rx I/O processor is the local bus width, which can be 8-, 16- or 32-bits wide.

For the purposes of assigning memory attributes, the physical address space is partitioned into 8 fixed 512 Mbyte regions determined by the upper three address bits. The regions are numbered as 8 paired sections for consistency with other i960 processor implementations. Region 0_1 maps to addresses 0000 0000H to 1FFF FFFFH and region 14_15 maps to addresses E000 0000H to FFFF FFFFH. The physical memory attributes for each region are programmable through the PMCON registers. The PMCON registers are loaded from the Control Table. The i960 Rx I/O processor provides one PMCON register for each region. The descriptions of the PMCON registers and instructions on programming them are found in section 12.2, PROGRAMMING THE PHYSICAL MEMORY ATTRIBUTES (PMCON REGISTERS) (pg. 12-3).

12.1.2 Logical Memory Attributes

The i960 Rx I/O processor provides a mechanism for defining two *Logical Memory Templates (LMTs)*. An LMT may be used to specify whether a section (or subset) of a physical memory subsystem connected to the BCU (e.g., DRAM, SRAM) is cacheable or non-cacheable in the on-chip data cache.

There are typically several different LMTs defined within a single memory subsystem. For example, data within one area of DRAM may be non-cacheable while data in another area is cacheable. Figure 12-1 shows the use of the Control Table (PMCON registers) with logical memory templates for a single DRAM region in a typical application.

Each logical memory template is defined by programming *Logical Memory Configuration (LMCON) registers*. An LMCON register pair defines a data template for areas of memory that have common logical attributes. The i960 Rx I/O processor has two pairs of LMCON registers — defining two separate templates. The extent of each data template is described by an address (on 4 Kbyte boundaries) and an address mask. The address is programmed in the Logical Memory Address register (LMADR). The mask is programmed in the Logical Memory Mask register (LMMSK). These two registers constitute the LMCON register pair.

The *Default Logical Memory Configuration (DLMCON)* register provides configuration data for areas of memory that do not fall within one of the two logical data templates.

The LMCON registers and their programming are described in section 12.5, PROGRAMMING THE LOGICAL MEMORY ATTRIBUTES (pg. 12-8).

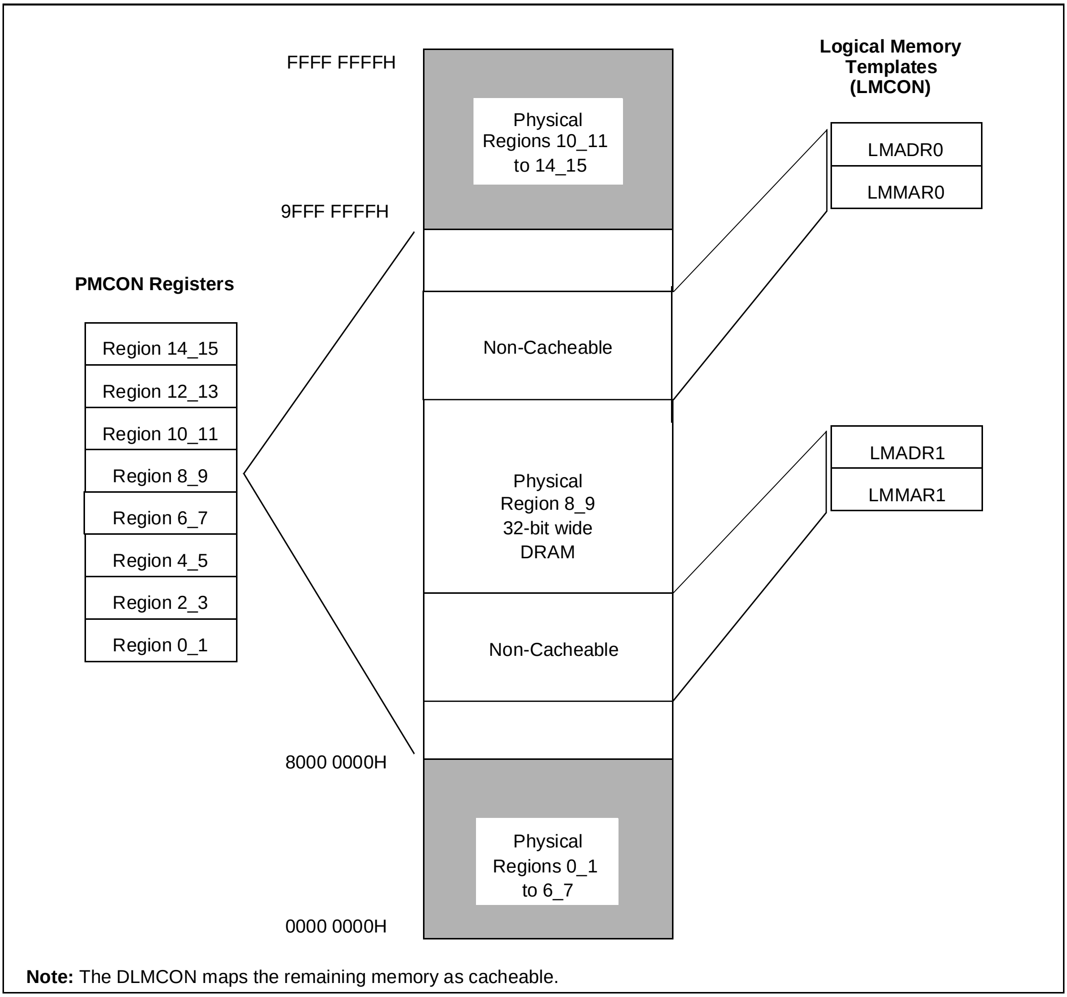


Figure 12-1. PMCON and LMCON Example

12.2 PROGRAMMING THE PHYSICAL MEMORY ATTRIBUTES (PMCON REGISTERS)

The Physical Memory Configuration registers, PMCON0_1 to PMCON14_15, are shown in Table 12-2. The PMCON registers reside within memory-mapped control register space. Each PMCON register controls one 512-Mbyte region of memory according to the mapping shown in Table 12-1.

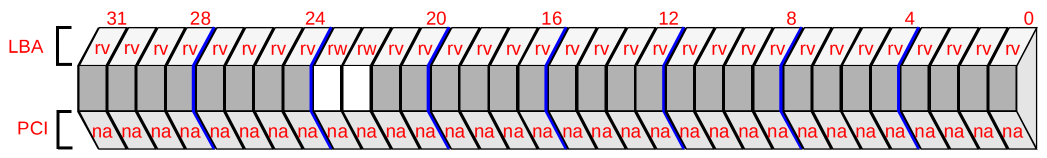
Table 12-1. PMCON Address Mapping

Register (Control Table Entry)	Region Controlled	Required Bus Width
Physical Memory Control Register 0 – PMCON0_1	0000 0000H to 0FFF FFFFH and 1000 0000H to 1FFF FFFFH	32 bits - 80960Rx Peripheral Memory-Mapped Registers
Physical Memory Control Register 1 – PMCON2_3	2000 0000H to 2FFF FFFFH and 3000 0000H to 3FFF FFFFH	Application dependent[1]
Physical Memory Control Register 2 – PMCON4_5	4000 0000H to 4FFF FFFFH and 5000 0000H to 5FFF FFFFH	Application dependent[1]
Physical Memory Control Register 3 – PMCON6_7	6000 0000H to 6FFF FFFFH and 7000 0000H to 7FFF FFFFH	Application dependent[1]
Physical Memory Control Register 4 – PMCON8_9	8000 0000H to 8FFF FFFFH and 9000 0000H to 9FFF FFFFH	32 bits - 80960Rx outbound ATU translation windows[2] (See Figure 16-5., 80960 Local Bus Memory Map - Outbound Translation Window (pg. 16-13))
Physical Memory Control Register 5 – PMCON10_11	A000 0000H to AFFF FFFFH and B000 0000H to BFFF FFFFH	Application dependent[2]
Physical Memory Control Register 6 – PMCON12_13	C000 0000H to CFFF FFFFH and D000 0000H to DFFF FFFFH	Application dependent[2]
Physical Memory Control Register 7 – PMCON14_15	E000 0000H to EFFF FFFFH and F000 0000H to FFFF FFFFH	Application dependent[2]

NOTES:

1. When direct addressing mode is enabled (bit 8 of the ATUCR), the region must be programmed to 32-bits wide. When disabled, the peripherals/memory connected to this region define the bus width to be programmed.
2. The user peripheral/memory connected to this region defines the bus width to be programmed.

Table 12-2. Physical Memory Control Registers – PMCON0:15


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: see Table 12-1
PCI: NA

Legend:
NA = Not Accessible   RO = Read Only
RV = Reserved   PR = Preserved   RW = Read/Write
RS = Read/Set   RC = Read Clear
LBA = 80960 Local Bus Address   PCI = PCI Configuration Address Offset

Bit	Default	Description
31:24	00H	Reserved. Initialize to 0.
23:22	00_2	Bus Width Selects the local bus width for a region: (00) = 8-bit (01) = 16-bit (10) = 32-bit bus (11) = reserved (do not use)
21:00	00 0000H	Reserved. Initialize to 0.

12.2.1 Local Bus Width

The local bus width for a region is controlled by the PMCON register. The operation of the i960 Rx I/O processor with different local bus width programming options is described in section 13.3.4, Bus Width (pg. 13-6).

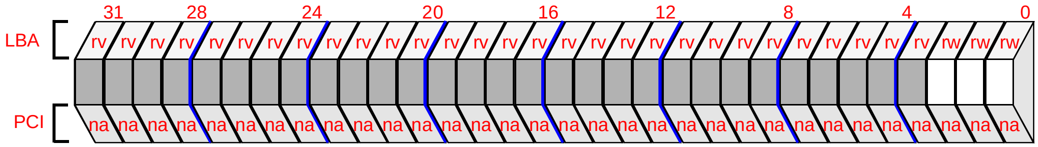
12.3 PHYSICAL MEMORY ATTRIBUTES AT INITIALIZATION

All eight PMCON registers are loaded automatically during system initialization. The initial values are stored in the Control Table in the Initialization Boot Record [see 11.4, INITIAL MEMORY IMAGE (IMI) (pg. 11-11)].

12.3.1 Bus Control Register – BCON

Immediately after a hardware reset, the PMCON register contents are marked invalid in the Bus Control (BCON) register. When the PMCON entries are marked invalid in BCON, the BCU uses the parameters in PMCON14_15 for *all* regions. On a hardware reset, PMCON14_15 is automatically cleared. This operation configures all regions to an 8-bit bus width. Subsequently, the processor loads all PMCON registers from the Control Table. The processor then loads BCON from the Control Table. When bit 2 of BCON is clear, PMCON14_15 remains in use for all local bus accesses. When bit 2 of BCON is set, the region table is valid and the BCU uses the programmed PMCON values for each region.

Table 12-3. Bus Control Register Bit Definitions – BCON


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 86FCH
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:03	0000 0000H	Reserved.
02	0_2	Configuration Entries in Control Table Valid) (0) = PMCON entries not valid, default to PMCON14_15 setting (1) = PMCON entries valid
01	0_2	Internal RAM Protection (0) = Internal data RAM not protected from user mode writes (1) = Internal data RAM protected from user mode write
00	0_2	Supervisor Internal RAM Protection (0) = First 64 bytes not protected from supervisor mode write (1) = First 64 bytes protected from supervisor mode writes

12.4 BOUNDARY CONDITIONS FOR PHYSICAL MEMORY REGIONS

The following sections describe the operation of the PMCON registers during conditions other than “normal” accesses.

12.4.1 Internal Memory Locations

The PMCON registers are ignored during accesses to internal memory or i960 core processor memory-mapped registers. The processor performs those accesses over 32-bit buses, except for local register cache accesses. The register bus is 128 bits wide.

12.4.2 Bus Transactions Across Region Boundaries

An unaligned bus request that spans region boundaries uses the PMCON settings of both regions. Accesses that lie in the first region use that region’s PMCON parameters, and the remaining accesses use the second region’s PMCON parameters.

For example, an unaligned quad word load/store beginning at address 1FFF FFFEH would cross boundaries from region 0_1 to 2_3. The physical parameters for region 0_1 would be used for the first 2-byte access and the physical parameters for region 2_3 would be used for the remaining access.

12.4.3 Modifying the PMCON Registers

An application can modify the value of a PMCON register by using the **st** or **sysctl** instruction. When a **st** or **sysctl** instruction is issued when an access is in progress, the current access is completed before the modification takes effect.

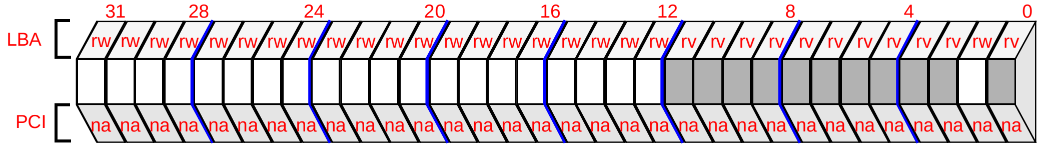
12.5 PROGRAMMING THE LOGICAL MEMORY ATTRIBUTES

Bit field definitions for Logical Memory Address Registers - LMADR1:0 and LMMR1:0 registers are shown in Table 12-4. LMCON registers reside within the i960 core processor memory-mapped control register space. (See APPENDIX C, MEMORY-MAPPED REGISTERS.)

12.5.1 Logical Memory Address Registers - LMADR0:1

The LMADR1:0 registers define the address for the logical data templates and template caching.

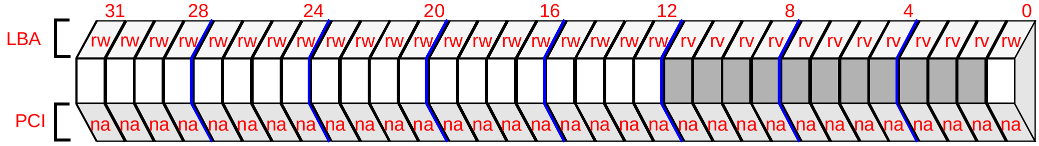
Table 12-4. Logical Memory Address Registers – LMADR0:1


LBA
PCI
31
28
24
20
16
12
8
4
0
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rw rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH0-8108H
	CH1-8110H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

PCI:	NA	
Bit	**Default**	**Description**
31:12	0000 0H	Template Starting Address - Defines upper 20 bits for the address of a logical data template. The lower 12 bits are fixed at zero. The starting address is modulo 4 Kbytes.
11:02	000H	Reserved.
01	0_2	Data Cache Enable - Controls data caching for the template. (0) = Data caching disabled (1) = Data caching enabled Instruction caching is never affected by this bit.
00	0_2	Reserved.

Table 12-5. Logical Memory Mask Registers – LMMR0:1


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

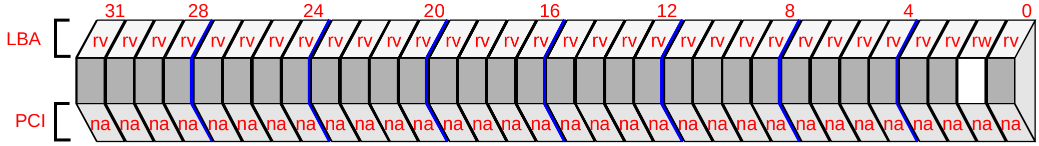
LBA:	CH0-810CH
	CH1-8114H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

PCI:	NA	

Bit	Default	Description
31:12	0000 0H	Template Address Mask - Defines upper 20 bits for the address mask for a logical memory template. The lower 12 bits are fixed at zero (MA). (0) = Mask (1) = Do not mask
11:01	000H	Reserved.
00	0_2	Logical Memory Template Enabled - Enables/disables logical memory template. (0) = LMT disable (1) = LMT enabled

The Default Logical Memory Configuration (DLMCON) register is shown in Table 12-6. The BCU uses the parameters in the DLMCON register when the current access does not fall within one of the two logical memory templates (LMTs).

Table 12-6. Default Logical Memory Configuration Register – DLMCON


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	8100H	**Legend:** NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

PCI:	NA	

Bit	Default	Description
31:02	0000 0000H	Reserved.
01	0_2	Data Cache Enable - Controls data caching for areas not within other logical memory templates. (0) = Data caching disabled (1) = Write-through caching enabled Instruction caching is never affected by this bit.
00	0_2	Reserved.

12.5.2 Defining the Effective Range of a Logical Data Template

For each logical data template, an LMADRx register sets the base address using the bits 31:12. The LMMR register sets the address mask using the bits 31:12. The effective address range for a logical data template is defined by using bits 31:12 in the LMADRx register and bits 31:12 in the LMMRx register.

For each access, only those address bits in the range 31:12 marked as unmasked (defined by bits MA31:12 in the LMMRx register), are compared against bits 31:12 in the LMMRx register. When all of the unmasked bits of the address match bits 31:12 of the LMMRx register, then the address falls within the memory region governed by "x" logical memory template. The lower 12 address bits are not compared and are thus considered masked bits or "don't care" bits. This forces a minimum 4 Kbyte boundary on a memory region governed by a logical memory template. Logically, the operation is as follows:

(EFA31:12 xnor LMADRx31:12) or (not LMMRx31:12)

Where EFA31:12 is the effective address for a bus access. Only when all compared address bits match is the logical data template used for the current access. Two examples help clarify the operation of the address comparators.

- Create a template 64 Kbytes in length beginning at address 0010 0000H and ending at address 0010 FFFFH. Determine the form of the candidate address to match and then program the LMADR and LMMR registers:

Candidate Address is of form:	`0010 XXXX`
LMADR <31:12> should be:	`0010 0...`
LMMR <31:12> should be:	`FFFF 0...`

- Multiple data templates can be created from a single LMADRxLMMRx register pair by aliasing effective addresses. For example, to create sixteen 64 Kbyte templates, each beginning on modulo 1 Mbyte boundaries starting at 0000 0000H and ending with 00F0 0000H, the registers are programmed as follows:

Candidate Address is of form:	`00X0 XXXX`
LMADR <31:12> should be:	`0000 0...`
LMMR <31:12> should be:	`FF0F 0...`

12.5.3 Data Caching Enable

Enabling and disabling data caching for an LMT is controlled via the bit 0 in the LMADR register. Likewise, the bit 1 in the DLMCON enables and disables data-caching for regions of memory that are not covered by the LMCON registers.

Disabling a memory range does not exclude an address range from being cacheable. For cacheable ranges, the BCU promotes all sub-word accesses to word accesses.

12.5.4 Enabling the Logical Memory Template

LMMRx bit 0 activates the logical data template in the LMMR register for the programmed range.

12.5.5 Initialization

Immediately following a hardware reset, all LMTs are disabled. The bit 0 in each of the LMMR registers is cleared (0) and all other bits are undefined. Also the Default Logical Memory Control register Data Caching Enable (LMADRx bit 1) is cleared (Data Caching Disabled). Application software may initialize and enable the logical memory template after hardware reset. The registers are not modified by software initialization.

12.5.6 Boundary Conditions for Logical Memory Templates

The following sections describe the operation of the LMT registers during conditions other than "normal" accesses. See CHAPTER 4, CACHE AND ON-CHIP DATA RAM for a treatment of data cache coherency when modifying an LMT.

12.5.6.1 Internal Memory Locations and Peripheral MMRs

The LMT registers are not used during accesses to i960 core processor memory-mapped registers. Internal data RAM locations are never cached; LMT bits controlling caching are ignored for data RAM accesses. The i960 Rx I/O processor peripheral MMRs, (addresses 0000 1000H through 0000 17FFH) and the ATU windows (8000 0000H through 9001 FFFFH) should be defined as non-cacheable. Further, if direct addressing is enabled (bit 8 of the ATUCR) addresses 0000 0000H through 7FFF FFFFH should be defined as non-cacheable.

12.5.6.2 Overlapping Logical Data Template Ranges

Logical data templates that specify overlapping ranges are not allowed. When an access is attempted that matches more than one enabled LMT range, the operation of the access becomes undefined.

To establish different logical memory attributes for the same address range, program non-overlapping logical ranges, then use partial physical address decoding.

12.5.6.3 Accesses Across LMT Boundaries

Accesses that cross LMT boundaries should be avoided. These accesses are unaligned and broken into a number of smaller aligned accesses, which reside in one or the other LMT, but not both. Each smaller access is completed using the parameters of the LMT in which it resides.

12.5.7 Modifying the LMT Registers

An LMT register can be modified using **st** or **sysctl** instructions. Both instructions ensure data cache coherency and order the modification with previous and subsequent data accesses.

13

LOCAL BUS

CHAPTER 13
LOCAL BUS

This chapter describes the bus interface of the i960® Rx I/O processor. It explains the following:

- Bus states and their relationship to each other
- Bus signals, which consist of address/data, control/status
- Read, write, burst and atomic bus transactions
- Related bus functions such as arbitration

This chapter also serves as a starting point for the hardware designer when interfacing typical peripheral devices to the i960 Rx I/O processor's address/data bus.

For information on programmable bus configuration, refer to CHAPTER 12, CORE PROCESSOR LOCAL BUS CONFIGURATION.

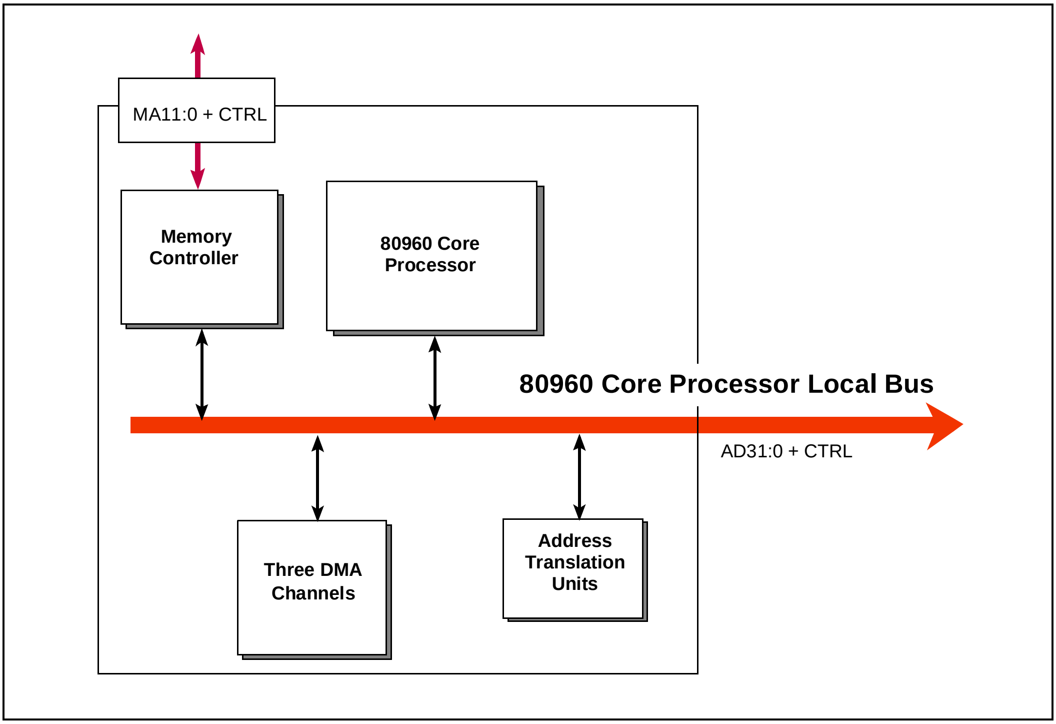


Figure 13-1. The Local Bus

13.1 OVERVIEW

The local bus is the data communication path between the various components of an i960 Rx I/O processor hardware system. It allows the processor to fetch instructions, manipulate data and interact with its I/O environment. To perform these tasks at high bandwidth, the processor features a burst transfer capability which allows successive 32-bit data transfers.

The local bus is controlled by the on-chip bus masters: the i960 core processor, the ATUs and DMA units. While the i960 core processor is limited to a burst length of four transfers, the ATUs and DMA units can burst up to naturally aligned 2 Kbyte boundaries.

The address/data path is multiplexed for economy, and bus width is programmable to 8-, 16- and 32-bit widths for i960 core processor accesses. The ATU and DMA units are limited to 32-bit bus widths. The processor has dedicated control signals for external address latches, buffers and data transceivers. In addition, the processor uses other signals to communicate with alternate bus masters. All bus transactions are synchronized with the processor's clock input (S_CLK); therefore, the memory system control logic can be implemented as state machines.

Users who are familiar with i960 JF processor should note the following differences in functionality between the i960 JF processor and the i960 Rx I/O processor. See Table 13-1.

Table 13-1. Differences Between 80960JF and 80960Rx Local Buses

Topic	80960JF	80960Rx
HOLD function	HOLD recognized during reset.	HOLD not recognized during reset.
Burst access limits	Four-word burst	i960 core processor: Four-word burst DMA units and ATU: 2 Kbyte
Data byte order	Supports big and little endian byte order.	Supports little endian byte order only.
BSTAT signal	Uses BSTAT to provide bus status information.	BSTAT signal not present.
A3:2 signal	A3:2 increments addresses during burst accesses.	A3:2 not present.
Bus width	Supports 8-, 16- or 32-bits bus widths	Peripherals that only interface to the i960 core processor can use 8-, 16 or 32-bit bus widths. Peripherals interfaced to the DMA units and ATUs must use 32-bit bus widths.
Bus alignment	Unaligned accesses broken up by microcode into aligned accesses.	i960 core processor: unaligned accesses broken up by microcode into aligned accesses. DMA units and ATU: No alignment restrictions.

13.1.1 Bus Operation

The terms *request*, *access* and *transfer* are used to describe bus operations. The processor's bus control unit decouples bus activity from instruction execution in the core as much as possible. When a load or store instruction or instruction prefetch is issued, a bus *request* is generated in the bus control unit. The bus control unit independently processes the request and retrieves data from memory for load instructions and instruction prefetches. The bus control unit delivers data to memory for store instructions.

A bus *access* is defined as a bus transaction bounded by the assertion of ADS# (address strobe) and de-assertion of BLAST# (burst last) signals, which are outputs from the processor. During each transfer, the processor either reads data or drives data on the bus. The number of transfers per access and the number of accesses per request is governed by the requested data length, the programmed width of the bus and the alignment of the address.

13.2 BASIC BUS STATES

The bus has five basic bus states: idle (T_I), address (T_A), wait/data (T_W/T_D), recovery (T_R), and hold (T_H). During system operation, the processor continuously enters and exits different bus states.

The bus occupies the idle (T_I) state when no address/data transactions are in progress and when P_RST# is asserted. When the processor needs to initiate a bus access, it enters the T_A state to transmit the address.

Following a T_A state, the bus enters the T_W/T_D state to transmit or receive data on the address/data lines. Assertion of the LRDYRCV# (Local Ready Recover) or RDYRCV# (Ready/Recover) signal indicates completion of each transfer. When data is not ready, the processor can wait as long as necessary for the memory or I/O device to respond.

In the case of a burst transaction, the bus exits the T_D state and re-enters the T_D/T_W state to transfer the next data word. The processor asserts the BLAST# signal during the last T_W/T_D states of an access. Once all data words transfer in a burst access, the bus enters the recovery (T_R) state to allow devices on the bus to recover.

The processor remains in the T_R state until LRDYRCV# or RDYRCV# is deasserted. When the recovery state completes, the bus enters the T_I state when no new accesses are required. When an access is pending, the bus enters the T_A state to transmit the new address.

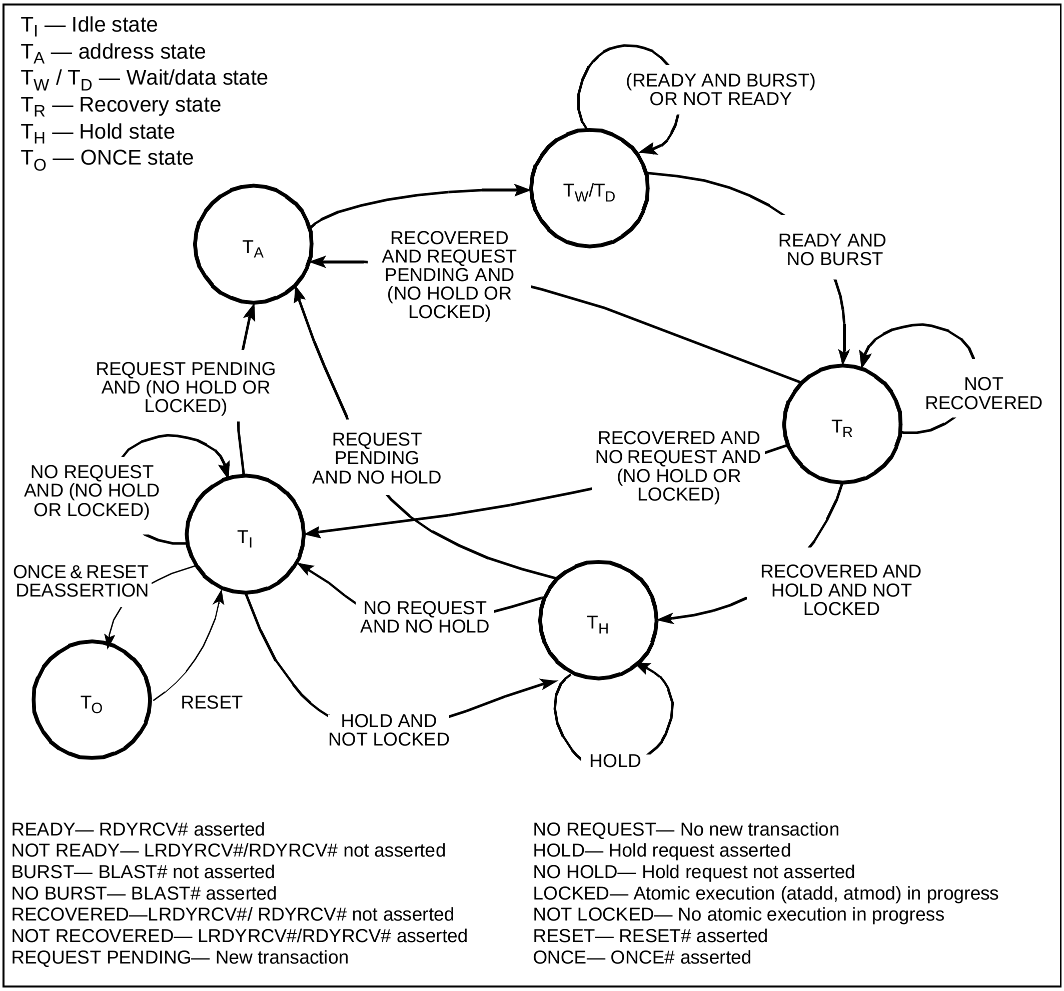

T_I — Idle state
T_A — address state
T_W / T_D — Wait/data state
T_R — Recovery state
T_H — Hold state
T_O — ONCE state
(READY AND BURST) OR NOT READY
T_W/T_D
T_A
RECOVERED AND REQUEST PENDING AND (NO HOLD OR LOCKED)
READY AND NO BURST
NOT RECOVERED
T_R
REQUEST PENDING AND (NO HOLD OR LOCKED)
REQUEST PENDING AND NO HOLD
RECOVERED AND NO REQUEST AND (NO HOLD OR LOCKED)
NO REQUEST AND (NO HOLD OR LOCKED)
T_I
ONCE & RESET DEASSERTION
NO REQUEST AND NO HOLD
T_H
RECOVERED AND HOLD AND NOT LOCKED
T_O
RESET
HOLD AND NOT LOCKED
HOLD
READY— RDYRCV# asserted
NOT READY— LRDYRCV#/RDYRCV# not asserted
BURST— BLAST# not asserted
NO BURST— BLAST# asserted
RECOVERED—LRDYRCV#/ RDYRCV# not asserted
NOT RECOVERED— LRDYRCV#/RDYRCV# asserted
REQUEST PENDING— New transaction
NO REQUEST— No new transaction
HOLD— Hold request asserted
NO HOLD— Hold request not asserted
LOCKED— Atomic execution (atadd, atmod) in progress
NOT LOCKED— No atomic execution in progress
RESET— RESET# asserted
ONCE— ONCE# asserted

Figure 13-2. Bus States with Arbitration

13.3 BUS SIGNAL TYPES

Bus signals consist of three groups: address/data, control/status and bus arbitration. A detailed description of all signals can be found in the *80960RP/RD Intelligent I/O Microprocessor* Data Sheets.

13.3.1 Clock Signal

The S_CLK input signal is the reference for all i960 Rx I/O processor signal timing relationships. Transitions on the AD31:0, ADS#, BE3:0#, WIDTH/HLTD1:0, D/C#, W/R#, DEN#, BLAST#, LRDYRCV# or RDYRCV#, LOCK#/ONCE#, HOLD, and HOLDA signals are always measured directly from the rising edge of S_CLK. The processor asserts ALE directly from the rising S_CLK edge at the beginning of a T_A state but deasserts them approximately half way through the state instead of the next rising S_CLK edge. All transitions on DT/R# are also referenced to a point halfway through the T_A state instead of rising S_CLK edges.

13.3.2 Address/Data Signal Definitions

The address/data signal group consists of 32 lines. These signals multiplex within the processor to serve a dual purpose. During T_A, the processor drives AD31:2 with the address of the bus access. At all other times, these lines are defined to contain data. AD1:0 denote burst size during T_A and data during other states.

The processor routinely performs data transfers less than 32 bits wide for i960 core processor accesses. When the programmed bus width is 32 bits and transfers are 16- or 8-bit, then during write cycles the processor replicates the data being driven on the unused address/data signals. When the programmed bus width is 16 or 8 bits, then during write cycles the processor continues driving address on any unused address/data signals.

Whenever the programmed bus width is less than 32 bits, additional demultiplexed address bits are available on unused byte enable signals. See section 13.3.4, Bus Width (pg. 13-6). These signals increment during burst accesses. The memory controller increments the addresses during bursts. See CHAPTER 14, MEMORY CONTROLLER for more information.

13.3.3 Control/Status Signal Definitions

The control/status signals control data buffers and address latches or furnish information useful to external chip-select generation logic. All output control/status signals are three-state.

Bus accesses begin with the assertion of ADS# (address/data status) during a T_A state. External decoding logic typically uses ADS# to qualify a valid address at the rising clock edge at the end of T_A. The processor pulses ALE (address latch enable) active high for one half clock during T_A to latch the multiplexed address on AD31:2 in external address latches.

The byte enable (BE3:0#) signals denote which bytes on the 32-bit data bus transfers data during an access. The processor asserts byte enables during T_A and deasserts them during T_R. When the data bus is configured for 16 bits, two byte enables become byte high enable and byte low enable and an additional address bit A1 is provided. When the bus is configured for 8 bits, there are no byte enables, but additional address bits A1:0 are provided. Note that the processor always drives byte enable signals to logical 1's during the T_R state, even when they are used as addresses.

The WIDTH1:0, D/C# and W/R# signals yield useful bus access information for external memory and I/O controllers. The WIDTH1:0 signals denote the i960 core processor's programmed physical memory attributes. The data/code signal D/C#, indicates whether an access is a data transaction (1) or an instruction transaction (0). The write/read signal W/R#, indicates the direction of data flow relative to the i960 Rx I/O processor. WIDTH1:0, D/C# and W/R# change state as needed during the T_A state.

DT/R# and DEN# signals control data transceivers. Data transceivers may be used in a system to isolate a memory subsystem or control loading on data lines. DT/R# (data transmit/receive) is used to control transceiver direction. In the second half of the T_A state, it transitions high for write cycles or low for read cycles. DEN# (data enable) is used to enable the transceivers. DEN# is asserted during the first T_W/T_D state of a bus access and deasserted during T_R. DT/R# and DEN# timings ensure that DT/R# does not change state when DEN# is asserted.

A bus access may be either non-burst or burst. A non-burst access ends after one data transfer to a single location. The processor asserts BLAST# (burst last) to indicate the last data cycle of an access in both burst and non-burst situations.

All i960 Rx I/O processor wait states to the local bus are controlled by either LRDYRCV# or RDYRCV#. See section 13.3.7.1, Recovery States (pg. 13-21) for a description of these signals.

13.3.4 Bus Width

Each region's data bus width is programmed in a Physical Memory Region Configuration (PMCON) register (see Chapter 12). The processor allows an 8-, 16- or 32-bit data bus width for each region. The processor places 8- and 16-bit data on low-order data signals, simplifying the interface to narrow bus external devices. As shown in Figure 13-3, 8-bit data is placed on lines AD7:0; 16-bit data is placed on lines AD15:0; 32-bit data is placed on lines AD31:0. The processor encodes bus width on the WIDTH1:0 signals so that external logic may enable the bus correctly. Note that DMA and ATU accesses are limited to 32-bit wide memory regions.

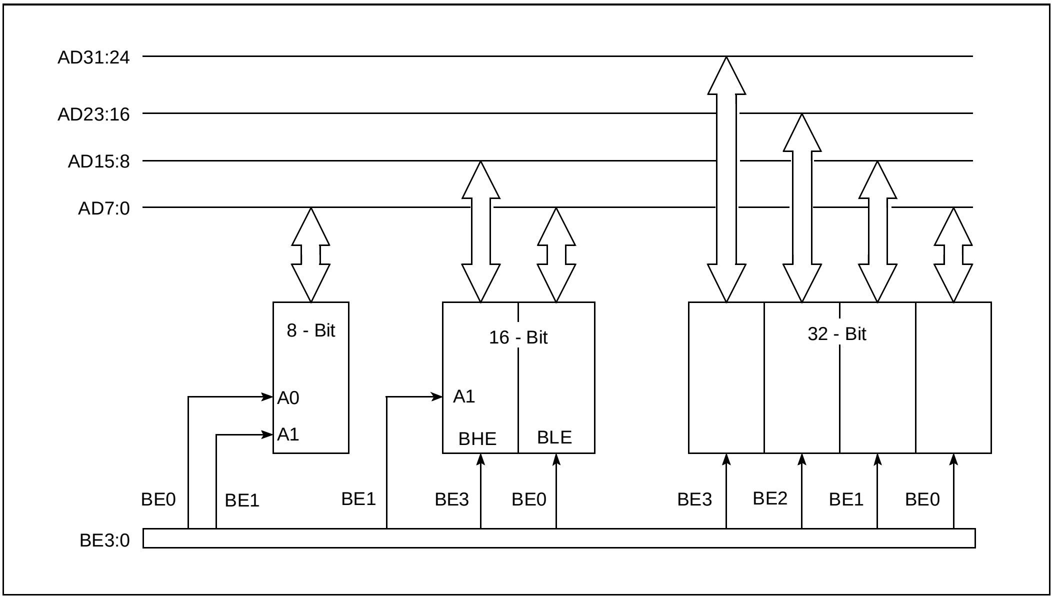


Figure 13-3. Data Width and Byte Encodings

Depending on the programmed bus width, the byte enable signals provide either data enables or low-order address lines:

- 8-bit region: BE0:1# provide the byte address (A0, A1). BE3:2# are not used.
- 16-bit region: BE1# provides the short-word address (A1); BE3# is the byte high enable signal (BHE#); BE0# is the byte low enable signal (BLE#). BE2# is not used.
- 32-bit region: byte enables are not encoded as address signals. Byte enables BE3:0# select bytes 0 through 3 of the 32-bit words addressed by AD31:2.

During initialization, the bus configuration data is read from the Initialization Boot Record (IBR) assuming an 8-bit bus width; however, the IBR can be in 8-bit, 16-bit or 32-bit physical memory. BE3:2# are defined as “1” so that reading the bus configuration data works for all bus widths. Since these byte enables are ignored for actual 8-bit memory, they can be permanently defined this way for ease of implementation.

The i960 Rx I/O processor drives determinate values on all address/data signals during T_W/T_D write operation states. For an 8-bit bus, the processor continues to drive address on unused data signals AD31:8. For a 16-bit bus, the processor continues to drive address on unused data signals AD31:16. However, when the processor does not use the entire bus width because of data width or misalignment (i.e., 8-bit write on a 16- or 32-bit bus or a 16-bit write on a 32-bit bus), data is replicated on those unused portions of the bus.

13.3.5 Basic Bus Accesses

The basic transaction is a read or write of one data word. The first half of Figure 13-4 shows a typical timing diagram for a non-burst, 32-bit read transaction. For simplicity, no wait states are shown.

During the T_A state, the i960 Rx I/O processor transmits the address on the address/data lines. In the figure, the SIZE bits (AD1:0) specify a single word transaction and WIDTH1:0 indicate a 32-bit wide access. For DMA and ATU accesses to the local bus, SIZE is not valid. The processor asserts ALE to latch the address and drives ADS# low to denote the start of the cycle. BE3:0# specify which bytes the processor uses to read the data word. The processor brings W/R# low to denote a read operation and drives D/C# to the proper state. For data transceivers, DT/R# goes low to define the input direction.

During the T_W/T_D state, the processor deasserts ADS# and asserts DEN# to enable any data transceivers. Since this is a non-burst transaction, the processor asserts BLAST# to signify the last transfer of a transaction. Figure 13-4 shows LRDYRCV#/RDYRCV# asserted, so this state is a data state and the processor latches data on a rising S_CLK edge. RDYRCV# is asserted by external logic.

The T_R state follows the T_W/T_D state. This allows the system components adequate time to remove their outputs from the bus before the processor drives the next address on the address/data lines. During the T_R state, BLAST#, BE3:0# and DEN# are inactive. W/R# and DT/R# hold their previous values. The figure indicates a logical high for the LRDYRCV#/RDYRCV# signal, so there is only one recovery state.

After a read, notice that the address/data bus goes to an invalid state during T_I. The processor drives valid logic levels on the address/data bus instead of allowing it to float. See section 13.4, BUS AND CONTROL SIGNALS DURING RECOVERY AND IDLE STATES (pg. 13-23) for the values that are driven during T_I.

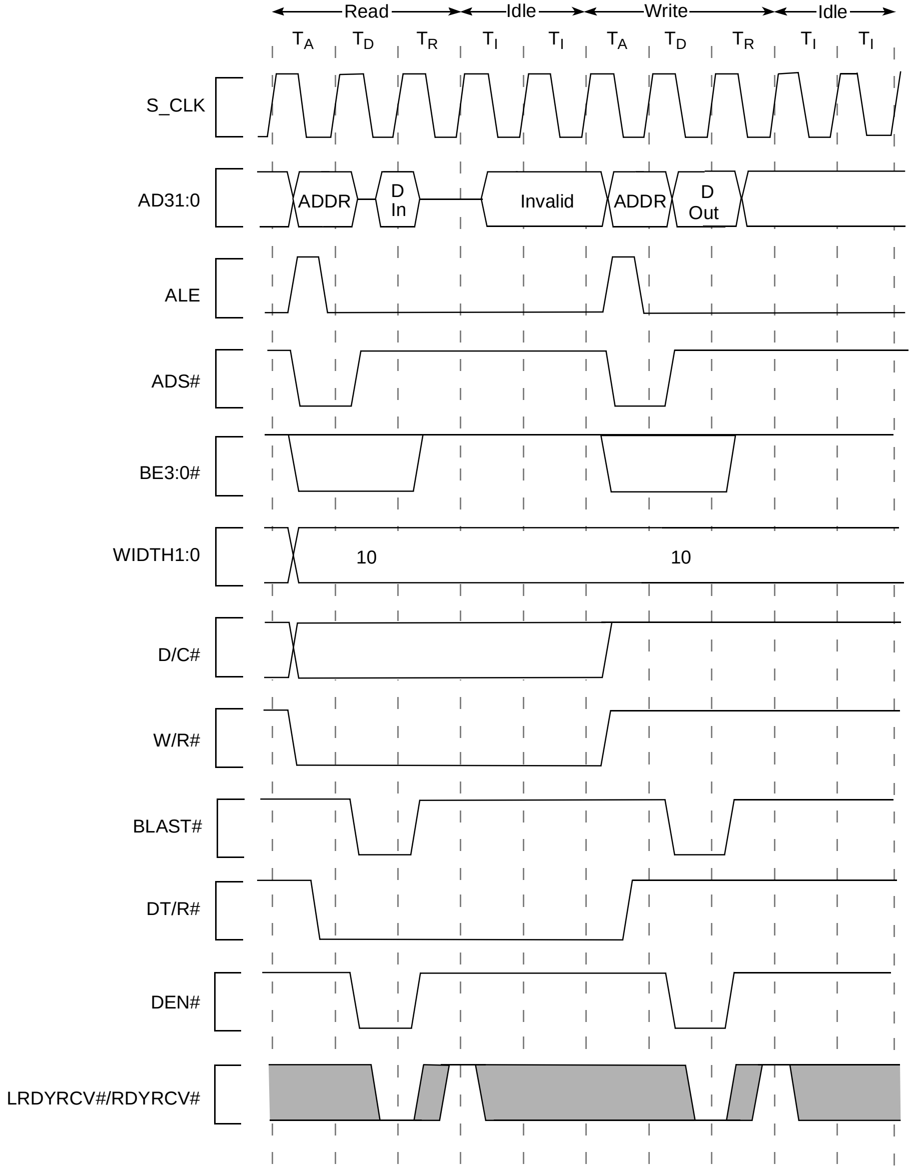


Figure 13-4. Non-Burst Read and Write Transactions Without Wait States, 32-Bit Bus

Figure 13-4 also shows a typical timing diagram for a non-burst, 32-bit write transaction. For the write operation, W/R# and DT/R# are high to denote the direction of the data flow. The D/C# signal is high since instruction code cannot be written. During the T_W/T_D state, the processor drives data on the bus, waiting to sample LRDYRCV#/RDYRCV# low to terminate the transfer. The figure shows LRDYRCV#/RDYRCV# asserted, so this state is a data state and the processor enters the recovery state. RDYRCV# is asserted by external logic.

At the end of a write, notice that the write data is driven during T_R and any subsequent T_I states. After a write, the processor drives write data until the next T_A state. See section 13.4, BUS AND CONTROL SIGNALS DURING RECOVERY AND IDLE STATES (pg. 13-23) for details.

13.3.6 Burst Transactions

A burst access is an address cycle followed by multiple data transfers. The i960 Rx I/O processor uses burst transactions to optimize local bus bandwidth. Burst transactions can be initiated by the i960 core processor, the ATUs and the DMA units. Burst transactions initiated by the i960 core processor have the same burst length and alignment rules as the i960 JF processor. However, burst transactions initiated by the ATUs and DMA units to the local bus have been further optimized to increase bandwidth by supporting much greater burst transfer lengths (up to 2K) and have added hardware support for optimized unaligned transfers.

When interfacing devices to the local bus that are accessed by on-chip i960 core processor only, the same burst length and alignment rules from the i960 JF processor apply. If devices connected to the local bus are targeted by either the ATUs or the DMA units, those device must support the additional local bus optimizations added by those units.

13.3.6.1 i960® Core Processor Burst Transactions

The maximum i960 core processor burst size is four data transfers, independent of bus width. These transfers are used by the i960 core processor for instruction fetching and accessing system data structures (i.e., load and store instructions). For an 8- and 16-bit bus widths, this means that some bus requests may result in multiple burst accesses. For example, a quad word load request (**ldq** instructions) to an 8-bit data region results in four 4-byte burst accesses.

For the i960 core processor, the burst accesses on the local bus are always aligned, meaning that byte lanes always carry valid data for each burst transfer (BE3:0# asserted). Table 13-2 summarizes the natural boundaries for load and store accesses from the i960 core processor.

When processing unaligned data requests from the i960 core processor, the Bus Control Unit breaks these accesses into a series of aligned burst accesses. The alignment rules for load and store requests are based on address offsets from natural data boundaries. Table 13-3 through Table 13-5 list all possible combinations of bus accesses resulting from aligned and unaligned requests. Figure 13-5 and Figure 13-6 depict the combinations for 32-bit buses.

The Process Control Block (PRCB) fault configuration word can configure the i960 core processor to handle unaligned accesses non-transparently by generating an OPERATION.UNALIGNED fault after executing any unaligned accesses. See section 11.4.2, Process Control Block – PRCB (pg. 11-17).

Table 13-2. i960® Core Processor Natural Boundaries for Load and Store Accesses

Data Width	Natural Boundary (Bytes)
Byte	1
Short Word	2
Word	4
Double Word	8
Triple Word	16
Quad Word	16

Table 13-3. i960® Core Processor Summary of Byte Load and Store Accesses

Address Offset from Natural Boundary (in Bytes)	Accesses on 8-Bit Bus (WIDTH1:0=00)	Accesses on 16 Bit Bus (WIDTH1:0=01)	Accesses on 32 Bit Bus (WIDTH1:0=10)
+0 (aligned)	byte access	byte access	byte access

Table 13-4. i960® Core Processor Summary of Short Word Load and Store Accesses

Address Offset from Natural Boundary (in Bytes)	Accesses on 8-Bit Bus (WIDTH1:0=00)	Accesses on 16 Bit Bus (WIDTH1:0=01)	Accesses on 32 Bit Bus (WIDTH1:0=10)
+0 (aligned)	burst of 2 bytes	short-word access	short-word access
+1	2 byte accesses	2 byte accesses	2 byte accesses

Table 13-5. i960® Core Processor Summary of *n*-Word Load and Store Accesses (*n* = 1, 2, 3, 4)

Address Offset from Natural Boundary in Bytes	Accesses on 8-Bit Bus (WIDTH1:0=00)	Accesses on 16 Bit Bus (WIDTH1:0=01)	Accesses on 32 Bit Bus (WIDTH1:0=10)
+0 (aligned) (*n* =1, 2, 3, 4)	• *n* burst(s) of 4 bytes	• case *n*=1: burst of 2 short words • case *n*=2: burst of 4 short words • case *n*=3: burst of 4 short words burst of 2 short words • case *n*=4: 2 bursts of 4 short words	• burst of *n* word(s)
+1 (*n* =1, 2, 3, 4) +5 (*n* = 2, 3, 4) +9 (*n* = 3, 4) +13 (*n* = 3, 4)	• byte access • burst of 2 bytes • *n*-1 burst(s) of 4 bytes • byte access	• byte access • short-word access • *n*-1 burst(s) of 2 short words • byte access	• byte access • short-word access • *n*-1 word access(es) • byte access
+2 (*n* =1, 2, 3, 4) +6 (*n* = 2, 3, 4) +10 (*n* = 3, 4) +14 (*n* = 3, 4)	• burst of 2 bytes • *n*-1 burst(s) of 4 bytes • burst of 2 bytes	• short-word access • *n*-1 burst(s) of 2 short words • short-word access	• short-word access • *n*-1 word access(es) • short-word access
+3 (*n* =1, 2, 3, 4) +7 (*n* = 2, 3, 4) +11 (*n* = 3, 4) +15 (*n* = 3, 4)	• byte access • *n*-1 burst(s) of 4 bytes • burst of 2 bytes • byte access	• byte access • *n*-1 burst(s) of 2 short words • short-word access • byte access	• byte access • *n*-1 word access(es) • short-word access • byte access
+4 (*n* = 2, 3, 4) +8 (*n* = 3, 4) +12 (*n* = 3, 4)	• *n* burst(s) of 4 bytes	• *n* burst(s) of 2 short words	• *n* word access(es)

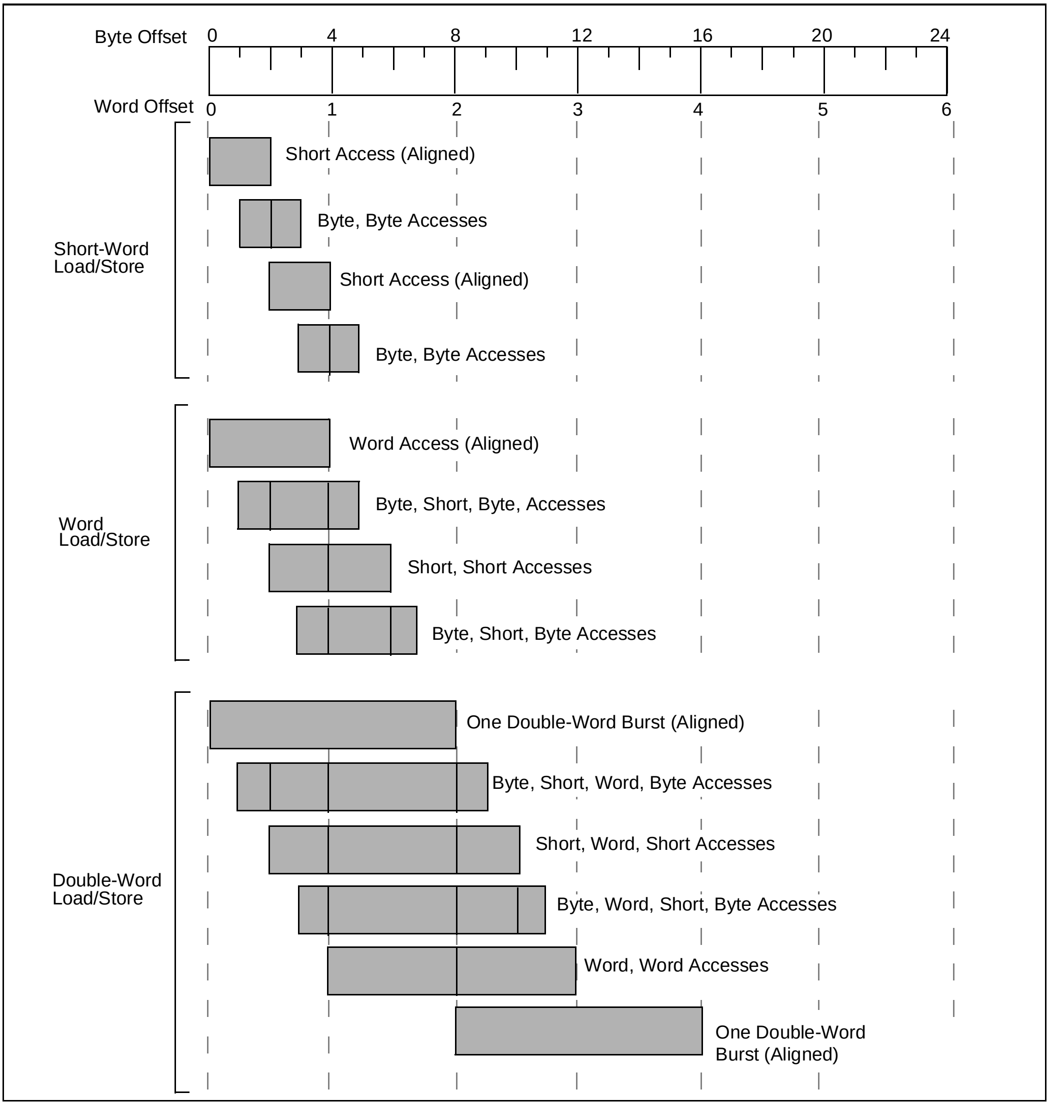


Figure 13-5. i960® Core Processor Summary of Aligned and Unaligned Accesses (32-Bit Bus)

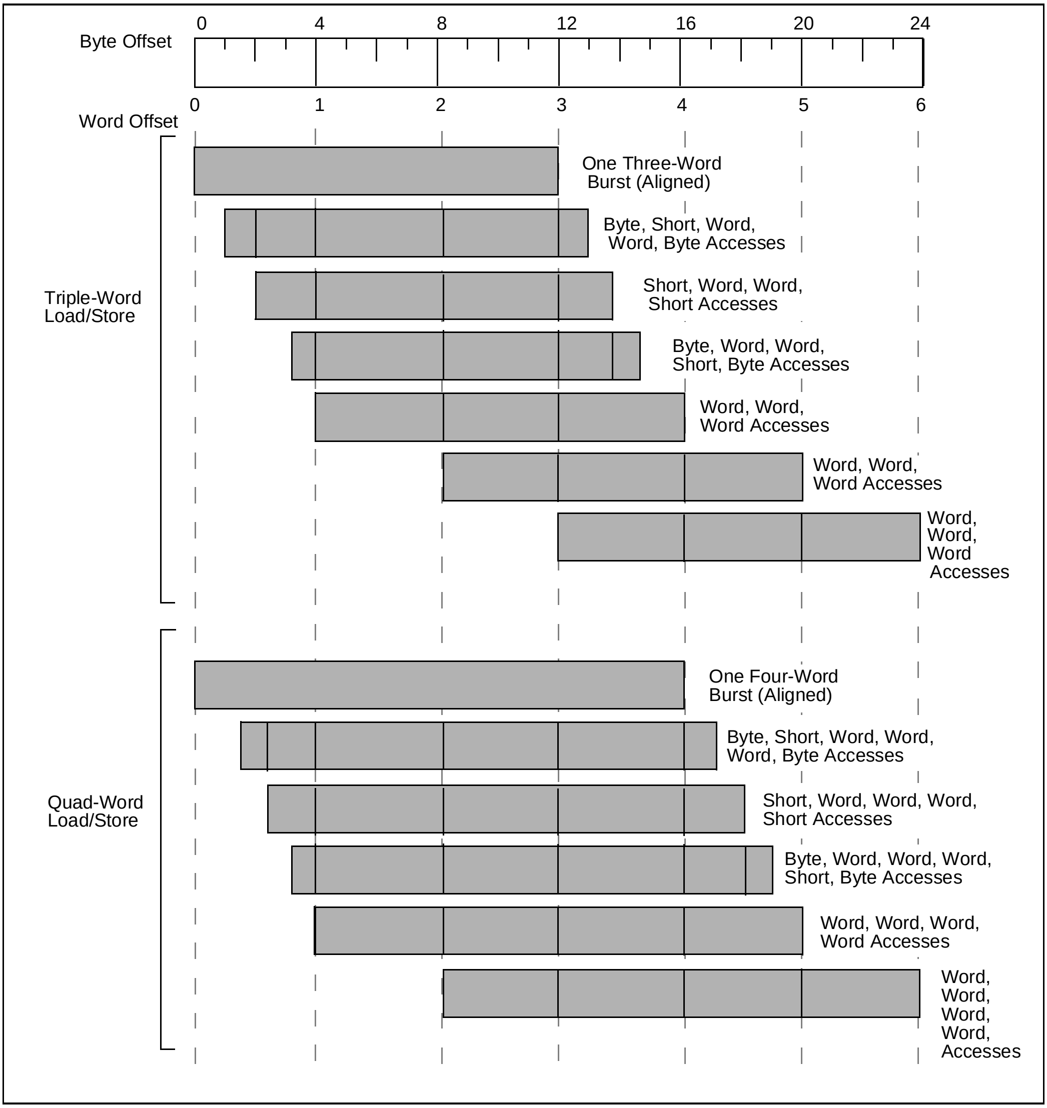

Byte Offset
0
4
8
12
16
20
24
Word Offset
0
1
2
3
4
5
6
Triple-Word Load/Store
One Three-Word Burst (Aligned)
Byte, Short, Word, Word, Byte Accesses
Short, Word, Word, Short Accesses
Byte, Word, Word, Short, Byte Accesses
Word, Word, Word Accesses
Word, Word, Word Accesses
Word, Word, Word Accesses
Quad-Word Load/Store
One Four-Word Burst (Aligned)
Byte, Short, Word, Word, Word, Byte Accesses
Short, Word, Word, Word, Short Accesses
Byte, Word, Word, Word, Short, Byte Accesses
Word, Word, Word, Word Accesses
Word, Word, Word, Word, Accesses

Figure 13-6. i960® Core Processor Summary of Aligned and Unaligned Accesses (32-Bit Bus) (Continued)

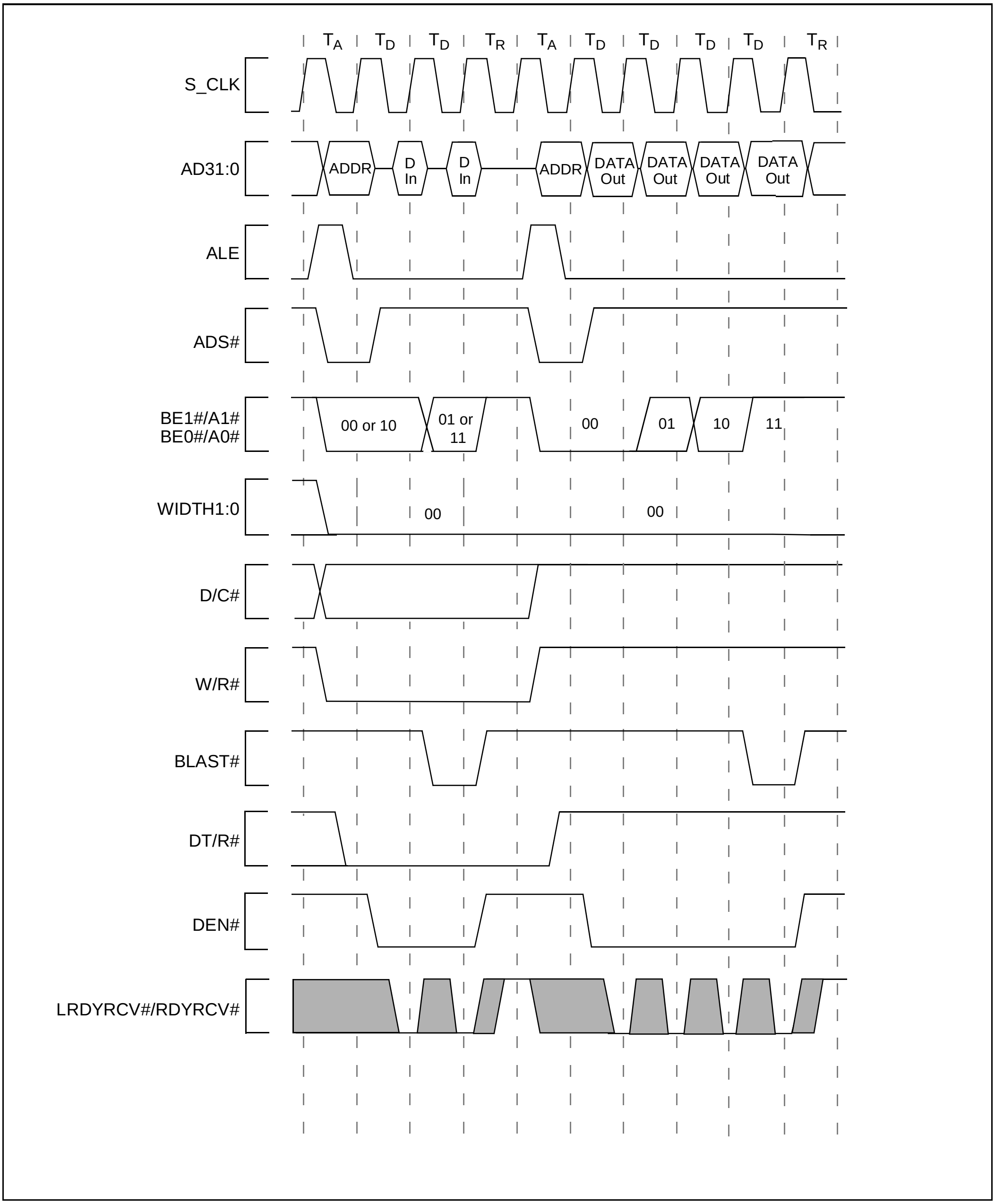


Figure 13-7. Burst Read and Write Transactions w/o Wait States, 8-bit Bus

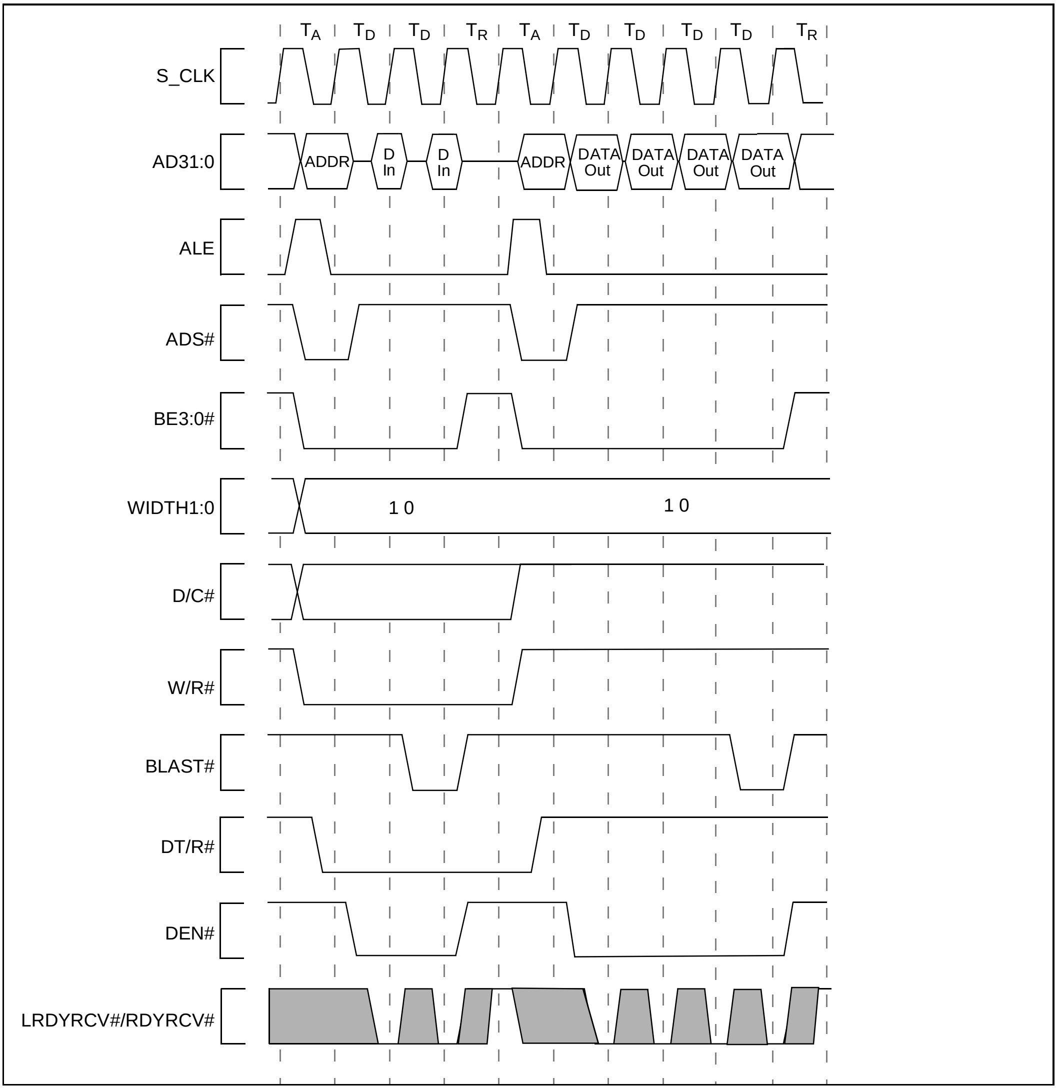


Figure 13-8. Burst Read and Write Transactions w/o Wait States, 32-bit Bus

13.3.6.2 ATU and DMA Burst Transactions

While the i960 core processor generates local bus accesses in response to data requests (**LD** and **ST** instructions) or instruction prefetching, the ATU and DMA units generate local bus accesses to move large blocks of data to and from the PCI buses. For most i960 Rx I/O processor applications, these burst accesses are translated by the on-chip memory controller directly to either DRAM or SRAM. However, it is possible for the DMA or ATU units to access external peripherals connected to the local bus.

To facilitate these large transfers, these units burst transfers up to naturally aligned 2K boundaries to the local bus. Because of this, the SIZE value driven on the AD1:0 signals during the T_A state is invalid. The cycle still begins with ADS# and ends with BLAST#.

The ATU and DMA units also do not break unaligned burst accesses into aligned accesses. For i960 core burst accesses, BE3:0# are unconditionally asserted for both reads and writes because the transfers are aligned. For the ATU and DMA unit write cycles, BE3:0# can change for each data transfer during a burst access to optimize the alignment. Figure 13-9 shows a seven-word burst write from either the DMA or ATU units that is offset from the word boundary by one byte. The transfer requires 8 burst data transfers, with 3 bytes valid for the first burst transfer, and one byte valid for the last transfer.

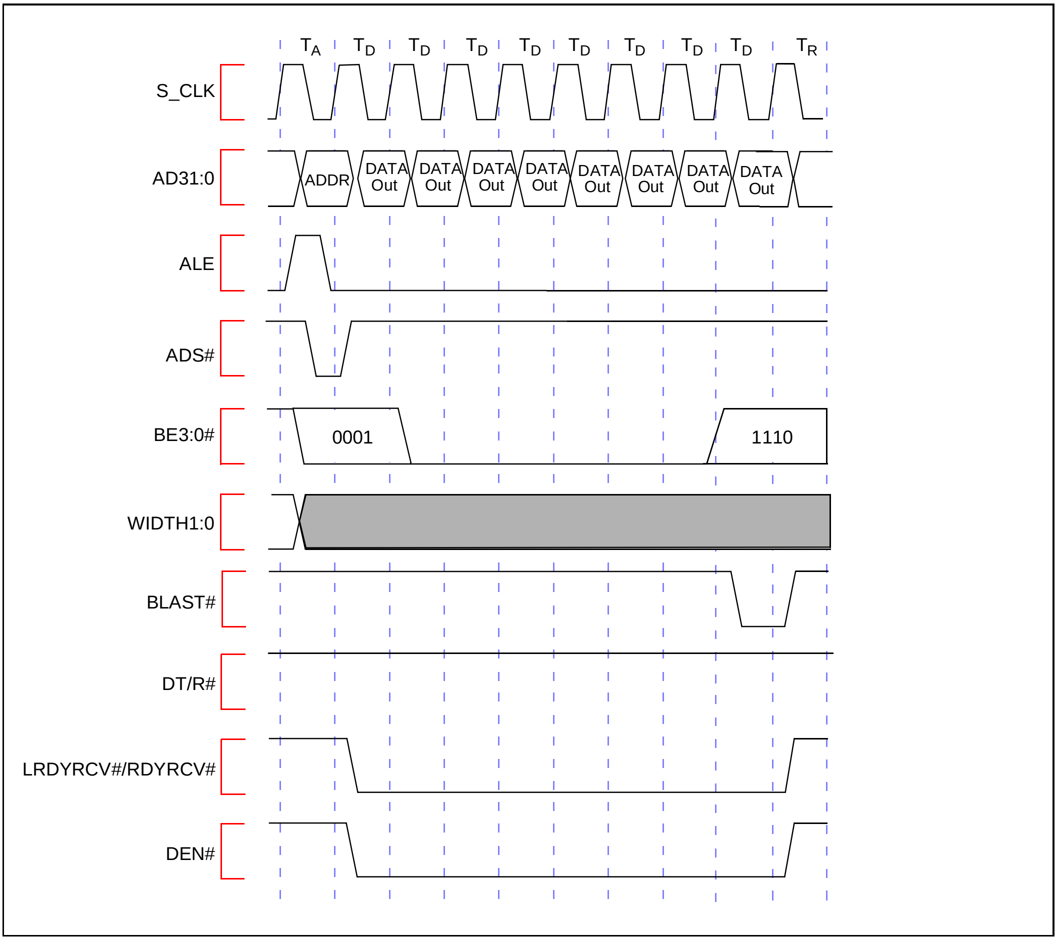


Figure 13-9. ATU or DMA 7-Word Unaligned Burst Transfer

13.3.7 Wait States

Wait states lengthen the processor’s bus cycles, allowing data transfers with slow memory and I/O devices. The i960 Rx I/O processor supports three types of wait states: *address-to-data*, *data-to-data* and *turnaround* or *recovery*. All three types are controlled through the processor’s LRDYRCV#/RDYRCV# signal. RDYRCV# is a synchronous input.

The processor's bus states follow the state diagram in Figure 13-2. After the T_A state, the processor enters the T_W/T_D state to perform a data transfer. When the memory (or I/O) system is fast enough to allow the transfer to complete during this clock (i.e., "ready"), LRDYRCV#/ is asserted. The processor samples LRDYRCV#/RDYRCV# low on the next rising clock edge, completing the transfer; the state is a data state. When the memory system is too slow to complete the transfer during this clock, LRDYRCV#/RDYRCV# is driven high and the state is an address-to-data wait state. Additional wait states may be inserted in similar fashion.

When the bus transaction is a burst, the processor re-enters the T_W/T_D state after the first data transfer. The processor continues to sample LRDYRCV#/RDYRCV# on each rising clock edge, adding a data-to-data wait state when LRDYRCV#/RDYRCV# is high and completing a transfer when LRDYRCV#/RDYRCV# is low. The process continues until all transfers are finished, with LRDYRCV#/RDYRCV# assertion denoting every data acquisition. The LRDYRCV# signal is generated internally by the 80960Rx for accesses by the memory controller and does not have to be generated externally.

Figure 13-10 illustrates a quad word burst write transaction with wait states. There are two address-to-data wait states single data-to-data wait states between transfers.

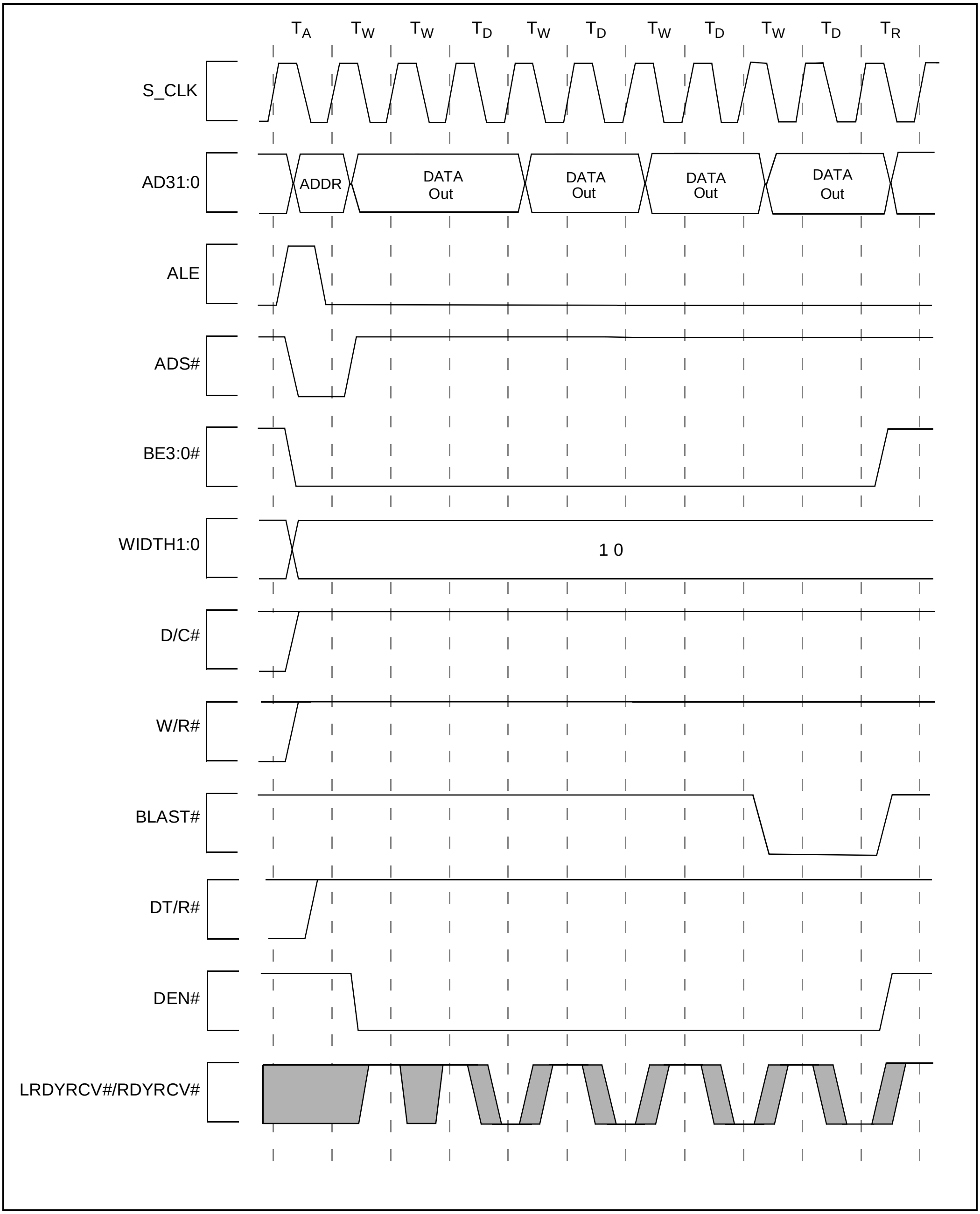


Figure 13-10. Burst Write Transactions With 2,1,1,1 Wait States, 32-bit Bus

13.3.7.1 Recovery States

The state following the last data transfer of an access is a recovery (T_R) state. By default, i960 Rx I/O processor bus transactions have one recovery state. External logic can cause additional recovery states to be inserted by driving the LRDYRCV#/RDYRCV# signal low at the end of T_R.

Recovery wait states are an important feature of the i960 Rx I/O processor because it employs a multiplexed bus. Slow memory and I/O devices often need a long time to turn off their output drivers on read accesses before the microprocessor drives the address for the next bus access. Recovery wait states are also useful to force a delay between back-to-back accesses to I/O devices with their own specific access recovery requirements.

System ready logic is often described as normally-ready or normally-not-ready. Normally-ready logic asserts a microprocessor's input signal during all bus states, except when wait states are desired. Normally-not-ready logic deasserts a processor's input signal during all bus states, except when the processor is ready. The subtle nomenclature distinction is important for i960 Rx I/O processor systems because the active sense of the LRDYRCV#/RDYRCV# signal reverses for recovery states.

- During the T_R state, logic 0 means "continue to recover" or "not ready"
- for T_W/T_D states, logic 0 means "ready"

Logic must assure "ready" and "not recover" are generated to terminate an access properly. Be certain to not hang the processor with endless recovery states. Conventional ready logic implemented as normally-not-ready operates correctly (but without adding turnaround wait states).

Figure 13-11 is a timing waveform of a read cycle followed by a write cycle, with an extra recovery state inserted into the read cycle.

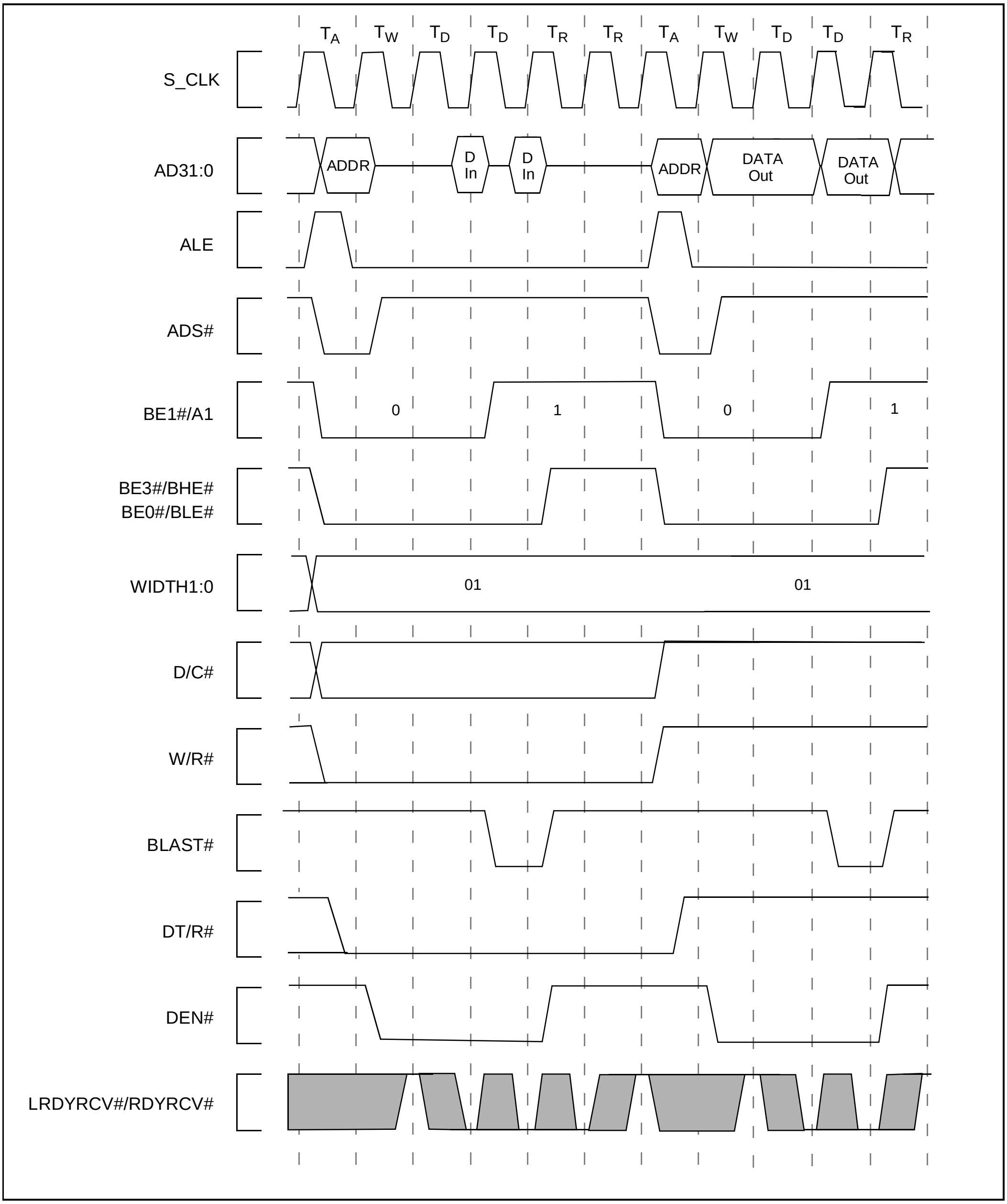


Figure 13-11. Burst Read/Write Transactions with 1,0 Wait States - Extra T_R State on Read, 16-Bit Bus

13.4 BUS AND CONTROL SIGNALS DURING RECOVERY AND IDLE STATES

Valid bus transactions are bounded by ADS# going active at the beginning of T_A states and BLAST# going inactive at the beginning of T_R states. During T_R and T_I states, bus and control signal logic levels are defined in such a way as to avoid unnecessary signal transitions that waste power. In all cases, the bus and control signals are completely quiet for instruction fetches and data loads that are cache hits.

When the last bus cycle is a read, the address/data bus floats during all T_R states. When the last bus cycle is a write, the address/data bus freezes during T_R states. The processor drives control signals such as ALE, ADS#, BLAST# and DEN# to their inactive states during T_R. Byte enables BE3:0# are always driven to logic high during T_R, even when the processor uses them under alternate definitions. Outputs without clearly defined active/inactive states such as WIDTH/HLTD1:0, D/C#, W/R# and DT/R# freeze during T_R.

When the bus enters the T_I state, the bus and control signals also freeze to inactive states. The exact states of the address/data signals depend on how the processor enters the T_I state. When the processor enters T_I from a T_R ending a write cycle, the processor continues driving data on AD31:0. When the processor enters T_I from a read cycle or from a T_H state, AD31:4 are driven with the upper 28 bits of the read address. The processor usually drives AD1:0 with the last SIZE information. In cases where the core cancels a previously issued bus request, AD1:0 are indeterminate.

13.5 ATOMIC BUS TRANSACTIONS

The atomic instructions, **atadd** and **atmod**, consist of a load and store request to the same memory location. Atomic instructions require indivisible, read-modify-write access to memory. That is, another bus agent must not access the target of the atomic instruction between read and write cycles. Atomic instructions are necessary to implement software semaphores.

13

For atomic bus accesses, the i960 Rx I/O processor asserts the LOCK# signal during the first T_A of the read operation and deasserts LOCK# in the last data transfer of the write operation. LOCK# is deasserted at the same clock edge that BLAST# is asserted. The i960 Rx I/O processor does not assert LOCK# except while a read-modify-write operation is in progress. While LOCK# is asserted, the processor can perform other, non-atomic, accesses such as fetches. However, the i960 Rx I/O processor does not acknowledge HOLD requests. This behavior is an enhancement over earlier i960 microprocessors. Figure 13-12 illustrates locked read/write accesses associated with an atomic instruction.

Note that LOCK# is only valid during i960 core processor accesses to external memory. Atomic accesses to the outbound ATU windows or ATU address space while direct addressing is enabled are not supported.

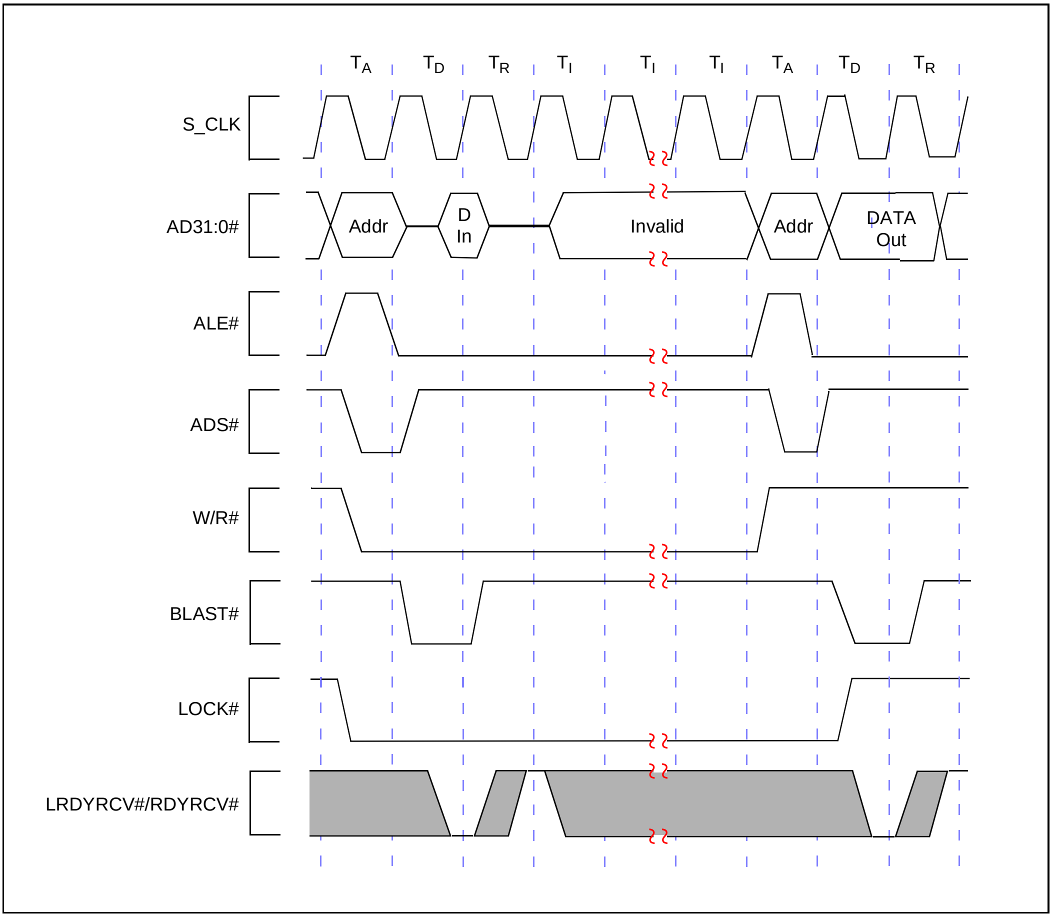


Figure 13-12. The LOCK# Signal

13.6 BUS ARBITRATION

The i960 Rx I/O processor can share the bus with other bus masters, using its built-in arbitration protocol. The protocol assumes two bus masters: a default bus master (typically the i960 Rx I/O processor) that controls the bus and another that requests bus control when it performs an operation. More than two bus masters may exist on the bus, but this configuration requires external arbitration logic. External bus masters do not have access to the 80960Rx's internal local bus. Therefore, an external bus master cannot access any of the 80960Rx's internal peripherals (e.g., the Memory Controller, the i960 core, etc.).

Two processor signal signals comprise the bus arbitration signal group.

13.6.1 HOLD/HOLDA Protocol

In most cases, the i960 Rx I/O processor controls the bus; an I/O peripheral (e.g., a communications controller) requests bus control. The processor and I/O peripheral device exchange bus control with two signals, HOLD and HOLDA.

HOLD is an i960 Rx I/O processor synchronous input signal which indicates that the alternate master needs the bus. HOLD may be asserted at any time so long as the transition meets the processor setup and hold requirements. HOLDA (hold acknowledge) is the processor output which indicates surrender of the bus. When the i960 Rx I/O processor asserts HOLDA, it enters the Hold state (see Figure 13-2). When the last bus state was T_I or the last T_R of a bus transaction, the processor is guaranteed to assert HOLDA and float the bus on the same clock edge in which it recognizes HOLD. Similarly, the processor deasserts HOLDA on the same edge in which it recognizes the deassertion of HOLD. Thus, bus latency is no longer than it takes the processor to finish any bus access in progress.

When the bus is in hold and the i960 Rx I/O processor needs to regain the bus to perform a transaction, the processor does not deassert HOLDA.

Unaligned load and store bus requests are broken into multiple accesses and the processor can relinquish the bus between those transactions. When the alternate bus master gives control of the bus back to the i960 Rx I/O processor, the processor immediately enters a T_A state to continue those accesses and respond to any other bus requests. When no requests are pending, the processor enters the idle state.

Figure 13-13 illustrates a HOLD/HOLDA arbitration sequence.

NOTE: External bus masters do not have access to the i960 Rx I/O processor internal, local bus. Therefore, an external bus master can not access any of the i960 Rx I/O processor internal peripherals (e.g., memory controller, i960 core processor, and memory-mapped registers).

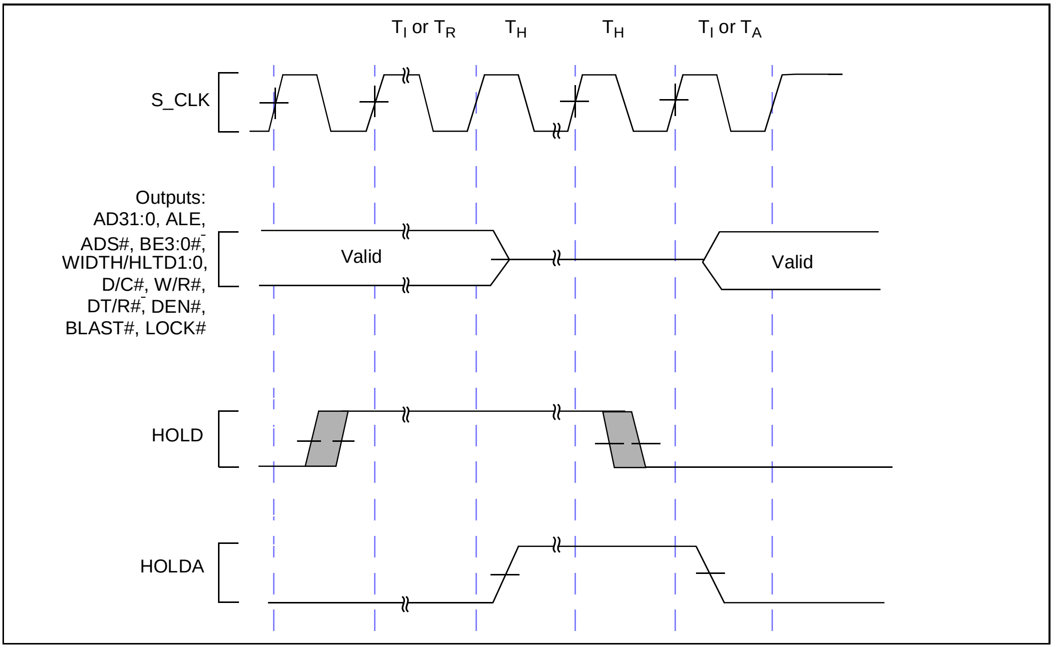


Figure 13-13. Arbitration Timing Diagram for a Bus Master

The 80960Rx arbitration logic enables external bus masters to control 80960Rx local bus. The Local Bus Arbitration Unit maintains the basic 80960Rx protocol for the HOLD/HOLDA except that the 80960Rx processor will not respond to the assertion of the HOLD signal (i.e., assert the HOLDA signal) during reset. This includes Processor Reset and Local Bus Reset.

14

MEMORY CONTROLLER

CHAPTER 14
MEMORY CONTROLLER

This chapter describes the i960® Rx I/O processor's integrated memory controller, including the supported memory types and theory of operation. This chapter also provides guidelines for connecting the memory controller to SRAM/ROM and DRAM systems. Figure 14-1 provides an overview of the i960 Rx I/O processor's integrated memory controller.

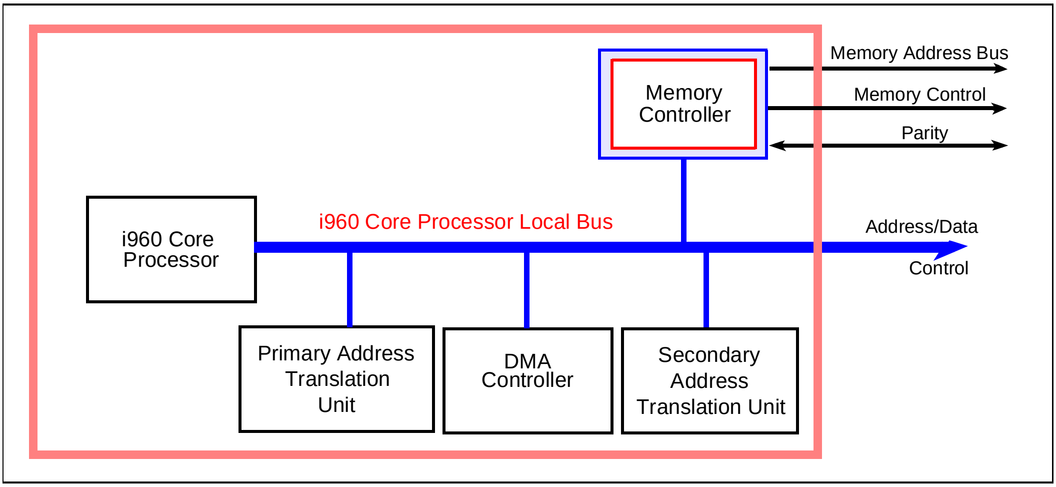


Figure 14-1. 80960Rx Processor Integrated Memory Controller

14.1 SUPPORTED MEMORY TYPES

The i960 Rx I/O processor integrates a memory controller to provide a direct interface with a memory system. The memory controller supports:

- Two independent memory banks of SRAM/ROM. Each bank can contain up to 16 Mbytes of 8- or 32-bit SRAM/ROM.
- Up to 256 Mbytes of 32-bit or 36-bit (32-bit memory data plus 4 parity bits) of:
  - Fast Page-Mode (FPM) Interleaved DRAM
  - Non-Interleaved DRAM
  - Extended Data Out (EDO) DRAM
  - Burst Extended Data Out (BEDO) DRAM

For a DRAM array, the memory controller generates row-address strobes (RAS3:0#), column-address strobes (CAS7:0#), write enables (DWE1:0#) and 12-bit multiplexed addresses (MA11:0). For interleaved DRAM, the DRAM address-latch enables (DALE1:0) and LEAF1:0# signals provide address and data latching.

Byte-wide data parity is supported for DRAM systems. Once enabled, the memory controller provides parity checking for all reads from memory. A parity error generates an error signal, which may be used for fault isolation.

The memory controller supports two banks of SRAM, ROM or Flash memory. Each bank supports from 64 Kbytes to 16 Mbytes of memory and can be configured independently for 8-bit or 32-bit wide memory. The memory controller also provides chip enables (CE1:0#), memory write enables (MWE3:0#) and an incrementing burst address for SRAM/ROM. The memory controller supports 0 wait-state performance for both read and write transactions.

14.2 THEORY OF OPERATION

The memory controller translates the i960 core processor’s burst access protocol to that of the memory being addressed. The memory controller decodes local bus addresses presented on the internal address/data bus, and generates the proper address and control signals to the memory array. Burst accesses generated by local-bus masters provide the first address. The memory controller provides incremental addresses that are presented to the memory array on the MA11:0 pins. The address increments until either the cycle has completed by the local-bus master, signified by asserting the BLAST# signal, or a local bus parity error for a DRAM read cycle occurs.

The address presented on the MA11:0 bus depends on the type of memory bank addressed. For DRAM, the MA11:0 pins provide the multiplexed row and column address. The column address increments to the nearest 2 Kbyte address boundary. on-chip bus master must implement a 2 Kbyte address boundary to prevent bursts from crossing a DRAM page. For both SRAM and Flash/ROM banks, the MA11:0 bus is based on the address presented on the AD13:2 signals during the address phase. For burst data, the memory controller increments the address to the nearest 2 Kbyte boundary.

Configuration registers select characteristics associated with each type of memory used in a system. The memory controller configuration registers are located in the address range 0000 1500H to 0000 15FFH. The memory-mapped registers are summarized in APPENDIX C, MEMORY-MAPPED REGISTERS. Once configured, the memory controller responds to addresses within an address range by issuing the appropriate memory-interface and bus-control signals.

Byte wide data parity generation and checking can be enabled for DRAM arrays. Parity checking provides a memory fault error upon detection of a parity error. The faulting word address is captured in a register.

The memory controller provides hardware DRAM refresh for CAS#-before-RAS# refresh cycles. It also provides hardware support for detecting address ranges that do not return an external RDYRCV# signal. This mechanism detects accesses to undefined address ranges. Upon detection of an error, the memory controller generates an internal LRDYRCV# signal to complete the bus accesses and optionally generates a bus fault signal.

Figure 14-2 shows the interface signals. Refer to the *80960RP/RD Intelligent I/O Microprocessor* Data Sheets for a complete description.

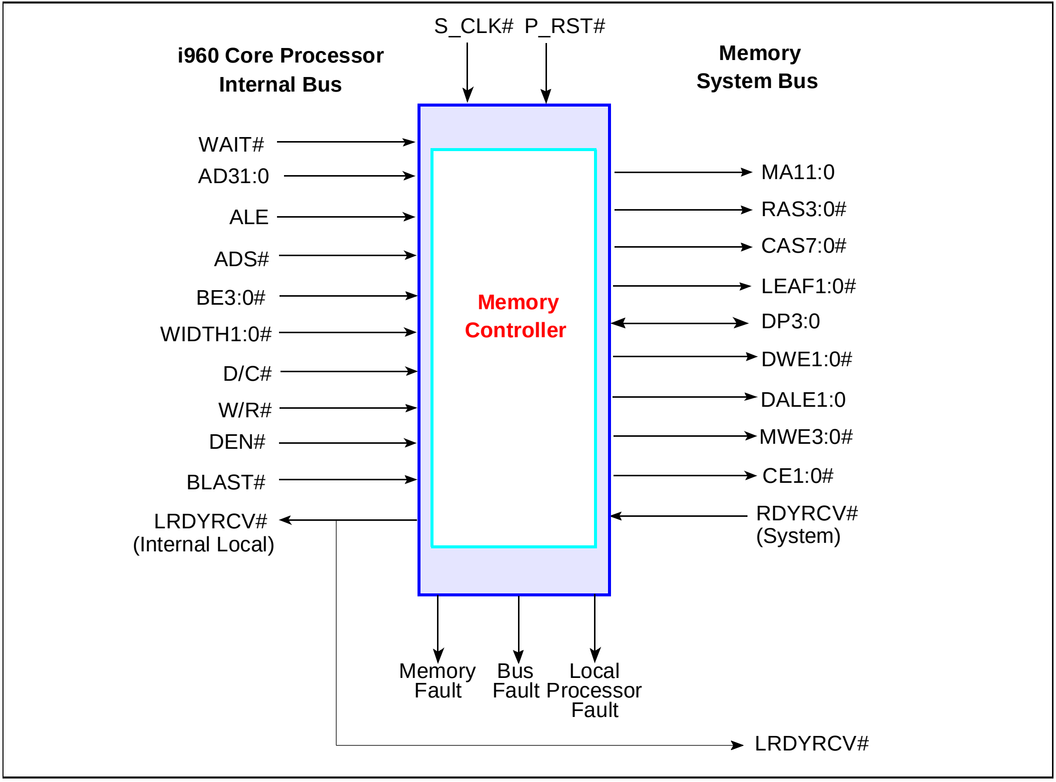


Figure 14-2. Memory Controller Signal Overview

14.3 Memory Controller Wait States

The memory controller generates the number of wait states programmed into the memory controller registers for controlling the signals connected to the memory arrays, see Section 14.5.3 on page 14-11. In addition, the WAIT# signal generated by the DMA unit (except the i960 core processor) indicates when additional wait states are required during a memory access. See Chapter 20, DMA CONTROLLER for more information on WAIT#.

14.4 ROM, SRAM and FLASH CONTROL

The memory controller supports two independent banks of ROM, SRAM, or Flash devices. Devices that use these memory banks may be organized as 8-bit or 32-bit wide memory. Each SRAM/ROM bank has a window of addresses that can be programmed to respond to any 80960 local bus address. Memory banks must not overlap with reserved addresses. See section 14.10, OVERLAPPING MEMORY REGIONS (pg. 14-48). The memory controller asserts the chip enable signals (CE1:0#) when the address on the 80960Rx local bus falls within the programmed window for the SRAM/ROM bank. The SRAM/ROM banks have independent control to support different memory types in each bank. The memory write enable signals, MWE3:0#, provide the write strobes for the selected memory bank. Connecting SRAM/ROM to the memory controller requires a combination of memory controller signals and local bus signals. Table 14-1 summarizes the memory controller signals and the local bus signals used when connecting SRAM/ROM to the memory banks.

Table 14-1. ROM, SRAM and Flash Control Signals

Source	Signal Name	Description
Memory Controller	MA11:0	Demultiplexed A13:2
	MWE3:0#	Memory write enable signifying valid data • MWE3# - Data valid on D31:24 • MWE2# - Data valid on D23:16 • MWE1# - Data valid on D15:08 • MWE0# - Data valid on D07:00
	CE1:0#	Chip Enable: • CE1# - Memory Bank 1 Chip Enable • CE0# - Memory Bank 0 Chip Enable
80960Rx Local Bus	AD31:0	Multiplexed Address/Data Bus
	W/R#	Specifies the access is a Read or Write transaction
	BE1:0#	Byte Enables - used for 8-bit memory only • BE1# - Becomes A1 • BE0# - Becomes A0
	ALE	Indicates Address Valid during an address cycle

For memory accesses that fall within the address windows for memory banks 0 and 1, the MA11:0 pins are translated to address bits during the address cycle. For 32-bit wide memory, the MA11:0 pins latch the address and provide a incrementing address during burst data accesses. The MA11:0 increments for burst data transfers up to a 2 Kbyte Page size boundary.

Eight-bit wide memory has a maximum burst count of four accesses. The incrementing burst address is presented on the BE1:0# pins, which translate to A1:0.

Figure 14-3 shows an example of a 2 Mbyte, 32-bit ROM or SRAM system connected to memory bank 0.

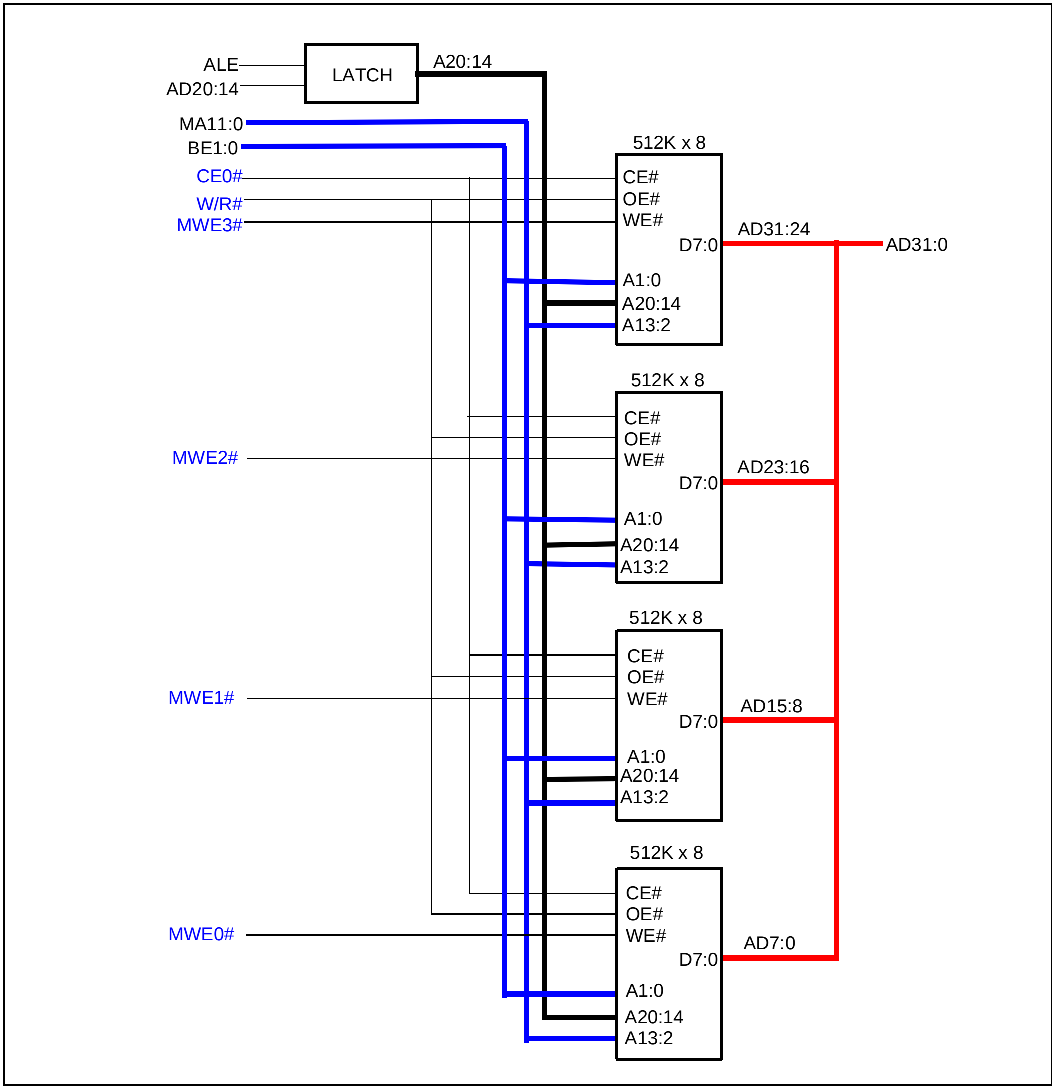


Figure 14-3. Bank0 32-Bit ROM or SRAM System

Figure 14-4 shows an example of an 1 Mbyte, 8-bit ROM or SRAM system connected to memory bank 0.

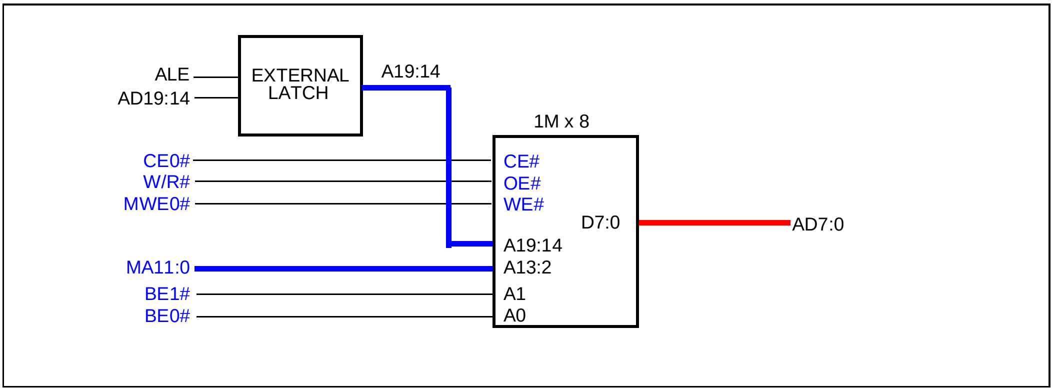


Figure 14-4. Bank0 8-Bit ROM or SRAM System

During ROM, SRAM and Flash memory accesses, the memory controller generates the incrementing address bits in conjunction with the control signals. The lower twelve bits of the address are generated on the MA11:0 memory address bus, and the upper address bits are generated on the AD31:14 multiplexed address/data bus. When addressing 8-bit memory, BE1# becomes A1 and BE0# becomes A0 as shown in Figure 14-4. Since the memory controller only latches A13:2, external logic must use ALE to latch the upper address bit during an address cycle. The CE1:0# signals provide unique chip enables that are used to select the device and activate its control logic during a memory access.

The write enable signals, MWE3:0#, select the byte lanes used during memory write accesses. During a memory write access, the appropriate combination of MWE3:0# and CE1:0# are asserted for the data cycle. The W/R# signal from the processor is driven high preventing the memory output from being enabled onto the address/data bus. During a memory read access, the MWE3:0# signals remain high while the appropriate CE1:0# is driven low by the memory controller. The W/R# signal from the processor is also driven low enabling the device's output onto the address/data bus.

The MWE3:0# signals may be used to select individual byte-wide Flash memory devices during programming without the use of external logic. The memory write enable bit allows the memory controller to assert MWE3:0# during write cycles. This bit is controlled in the Memory Bank Control Register (MBCR) shown in Figure 14-3. If either memory bank 0 or 1 is used for SRAM, the memory write enable bit must be set to enable the assertion of the MWE3:0# signals for memory write transactions.

14.5 MEMORY BANK PROGRAMMING REGISTERS

Seven memory-mapped registers provide independent control of memory banks 0 and 1:

Table 14-2. Memory Bank Register Summary

Section	Register Name, Acronym	Page	Size (Bits)	Channel	80960 Local Bus Address	PCI Config Addr Offset
14.5.1	Memory Bank Control Register - MBCR	14-7	32		0000 1500H	NA
14.5.2	Memory Bank Base Address Registers - MBBAR0:1	14-10	32	0 1	0000 1504H 0000 1510H	NA
14.5.3.1	Memory Bank Read Wait State Registers - MBRWS0:1	14-11	32	0 1	0000 1508H 0000 1514H	NA
14.5.3.2	Memory Bank Write Wait State Registers - MBWWS0:1	14-13	32	0 1	0000 150CH 0000 1518H	NA

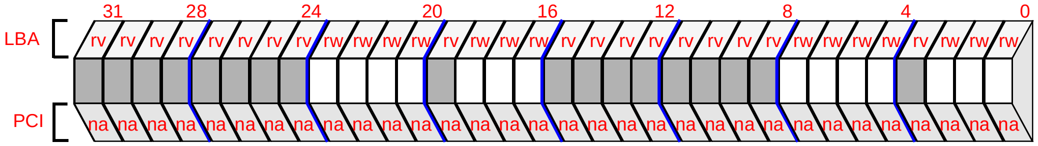
Refer to APPENDIX C, MEMORY-MAPPED REGISTERS for the memory-mapped registers address mappings.

14.5.1 Memory Bank Control Register - MBCR

The Memory Bank Control Register (MBCR) specifies parameters that dictate the memory controller operating environment for the two memory banks. The MBCR should be programmed after initializing the other memory bank registers. Table 14-3 shows the register format for the MBCR. The memory bank enable bits should be disabled prior to modifying the memory bank base address and wait state registers.

Memory Bank 0 initializes to an enabled state on the rising edge of P_RST# to support a Boot ROM for the i960 core processor. Bank size, wait state profiles and memory enables initialize to the maximum programmable values. Once the i960 core processor begins code execution, software should re-program the memory controller for the actual bank size and wait state profiles for the physical memory connected. Refer to section 14.5.2, Memory Bank Base Address Registers - MBBAR0:1 (pg. 14-10) for additional information.

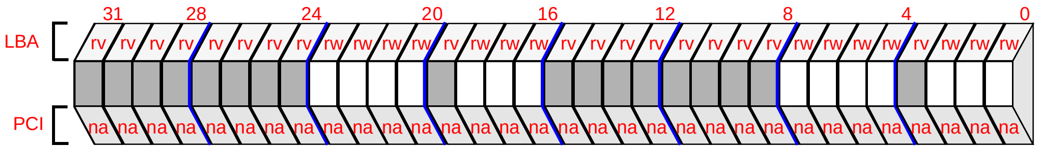
Table 14-3. Memory Bank Control Register – MBCR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rw rw rw rw rv rw rw rw rv rv rv rv rv rv rv rv rw rw rw rw rv rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1500H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:24	00H	Reserved
23:20	0H	**Memory Bank 1 Size Field** - This bit field contains the total block size of memory connected to memory bank 1. Memory may be ROM, SRAM or Flash with the size ranging from 64 Kbytes to16 Mbytes. Each bank may be organized as 8- or 32-bit wide memory, but must consist of a uniform memory type. 0000 64 Kbytes 0001 128 Kbytes 0010 256 Kbytes 0011 512 Kbytes 0100 1 Mbyte 0101 2 Mbytes 0110 4 Mbyte 0111 8 Mbytes 1xxx 16 Mbytes
19	0_2	Reserved
18	0_2	**Memory Bank 1 Extended** MWE3:0# **Bit** - This bit field enables or disables extending the deassertion period for the MWE3:0# signal during burst write cycles. The bit also enables one clock of MA11:0 and BE1:0 hold time relative to the rising edge of MWE# during writes to this region. When cleared (0), deassertion period is one-half of a CLKIN period. When set (1), the deassertion period is extended by the wait state profile defined in the MBWWS1 registers in addition to the one-half clock in period. Also when set, the MA11:0 and BE1:0 keep their current state for one clock after MWE3:0# are deasserted. This also adds an extra wait state. MWE wait states can be calculated by the following: Address or Data Wait States = (t_{WWX} * 2) + 1 where t_{WWX} = t_{WWA} or t_{WWD}
17	0_2	**Memory Bank 1 Write Enable Bit** - This bit enables or disables the MWE3:0# signals during write cycles to memory bank 1. When cleared (0), the MWE3:0# is not asserted during write cycles to memory bank 1. When set (1), the MWE3:0# signals is asserted during write cycles to memory bank 1.

Table 14-3. Memory Bank Control Register – MBCR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rw rw rw rw rv rw rw rw rv rv rv rv rv rv rv rv rw rw rw rw rv rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1500H

PCI: NA

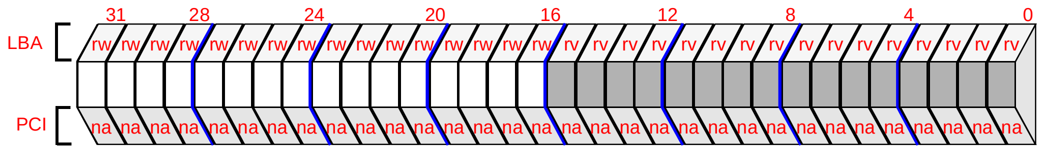
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
16	0_2	**Memory Bank 1 Enable Bit** - enables or disables CE1# for memory bank 1. When cleared (0), the memory controller does not assert CE1#. When set (1), memory controller decodes local bus addresses and asserts CE1# when local bus address falls within the window of address programmed into MBBAR1 in conjunction with memory bank 1 size control bits.
15:08	00H	Reserved
07:04	0H	**Memory Bank 0 Size Field** - contains the total block size of memory connected to memory bank 0. Memory connected may be ROM, SRAM or Flash memory; size may range from 64 Kbytes to 16 Mbytes. Each bank may be organized as 8 or 32 bit wide memory, and must consist of a uniform memory type. See Memory Bank 1 Size Field for block size settings.
03	0_2	Reserved
02	0_2	**Memory Bank 0 Extended** MWE3:0# **Bit** - This bit field enables or disables extending the deassertion period for the MWE3:0# signal during burst write cycles. The bit also enables one clock of MA11:0 and BE1:0 hold time relative to the rising edge of MWE# during writes to this region. When cleared (0), deassertion period is one-half of a CLKIN period. When set (1), the deassertion period is extended by the wait state profile defined in the MBWWS0 registers in addition to the one-half clock in period. Also when set, the MA11:0 and BE1:0 keep their current state for one clock after MWE3:0# are deasserted. This also adds an extra wait state. MWE wait states can be calculated by the following: Address or Data Wait States = $(t_{WWX} * 2) + 1$ where $t_{WWX} = t_{WWA}$ or t_{WWD}
01	0_2	**Memory Bank 0 Write Enable Bit** - This bit enables or disables the MWE3:0# signals during write cycles to memory bank 0. When cleared (0), the MWE3:0# is not asserted during write cycles to memory bank 0. When set (1), the MWE3:0# signals is asserted during write cycles to memory bank 0.
00	1_2	**Memory Bank 0 Enable Bit** - enables or disables CE0# for memory bank 0. When cleared (0), the memory controller does not assert the CE0#. When set (1), the memory controller decodes the local bus addresses and asserts CE0# when the local bus address falls within the window of addresses programmed into the MBBAR0 in conjunction with the memory bank 0 size control bits. Memory Bank 0 defaults as enabled. This memory bank should be used for connecting boot ROM for booting the i960 core processor.

14.5.2 Memory Bank Base Address Registers - MBBAR0:1

The memory bank base addresses are programmed through the Memory Bank Base Address Registers (MBBAR0:1). The base address for each memory bank must be on an address boundary equal to its size. For example, a memory bank size of 1 Mbyte must have a starting address located on a 1 Mbyte address boundary. The MBBARx register definitions are shown in Table 14-4.

Table 14-4. Memory Bank Base Address Registers – MBBAR0:1


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: CH0-1504H
CH1-1510H
PCI: 04H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:16	FE00H (Bank 0) 0000H (Bank 1)	**Memory Bank 0/1 Base Address** - These bits define the base address to which the memory bank responds when addressed from the local bus. The default base address for memory bank 0 is FE00 0000H with a bank size of 16 Mbytes used to address the Initialization Boot Record table for booting the i960 core processor.
15:0	0000H	Reserved

The Initialization Boot Record (IBR) is the primary data structure required to initialize the i960 core processor and must be located at address FEFF FF30H. Since the processor must access the IBR before the memory controller has been configured, a bank base address of FE00 0000H and a bank size of 16 Mbytes are used by default for Memory Bank 0. These values result in an address decode range of FE00 0000H to FEFF FFFFH for memory bank 0 when the memory controller is reset. For the i960 core processor to boot from ROM or Flash memory, the memory devices must use Memory Bank 0 and its associated chip enable signal, CE0#. The default address is used by the memory controller for address decoding until it is configured by programming the Memory Bank 0 Base Address Register and Memory Bank 0 Size with the ROM bank base address and size information, respectively.

NOTE: The i960 core processor does not generate external bus cycles for transactions within the address range of 0 to 0000 03FFH or FF00 0000H to FFFF FFFFH. These address ranges are reserved by the processor for internal data RAM and memory-mapped registers, respectively. The memory bank base address registers should not be programmed with a value within these reserved address ranges.

14.5.3 Memory Bank Wait State Registers - MBRWS0:1, MBWWS0:1

Bus cycle timing for ROM, SRAM and Flash memory accesses are programmed through the internal wait-state registers (see Table 14-2 for register summaries):

- Memory Bank 0 Read Wait States Register (MBRWS0)
- Memory Bank 1 Read Wait States Register (MBRWS1)
- Memory Bank 0 Write Wait States Register (MBWWS0)
- Memory Bank 1 Write Wait States Register (MBWWS1)

The number of wait states for each access in a bus cycle is programmed in 1x increments of S_CLK. The i960 core processor requires one recovery cycle, but it may need to be extended to accommodate slower memory devices. Each memory bank contains registers to independently program the read and write wait states. The programmable values support:

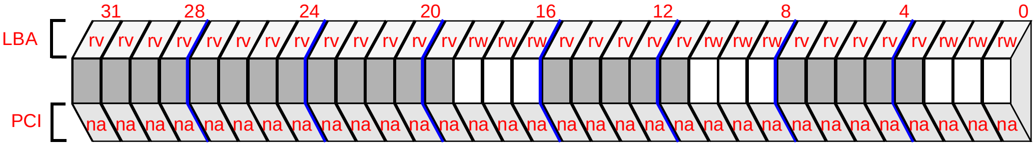
- Address-to-Data wait states
- Data-to-Data wait states
- Data-to-Address wait states (i.e., turnaround cycles)

The programmable range of values is sufficient to support memory access cycle times from 60 to 200 ns while operating the processor at 25 or 33 MHz. The register definitions for the memory bank read wait states registers are shown in Figure 14-3.

14.5.3.1 Memory Bank Read Wait State Registers - MBRWS0:1

The Memory Bank Read Wait State Register (MBRWS) describes the wait states during Read cycles.

Table 14-5. Memory Bank Read Wait States Register – MBRWS0:1


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rv rv rv rv rv rw rw rw rv rv rv rv rv rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: Bank 0 = 1508H
Bank 1 = 1514H
PCI: N/A

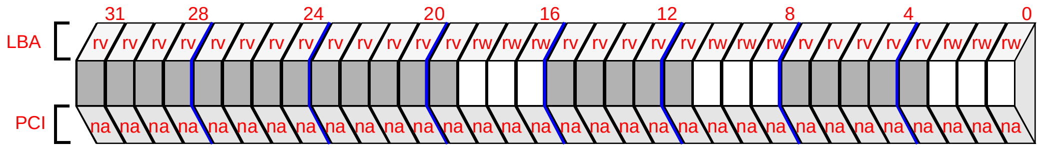
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:19	0000H	Reserved
18:16	111_2	**Read Cycle Address-to-First Data Wait States (t_{WRA})** - This bit field represents the number of wait states between address and the first data for read transactions. The bit field is encoded as: 000 0 Address-to-Data wait states 001 1 Address-to-Data wait state 010 2 Address-to-Data wait states 011 3 Address-to-Data wait states 100 4 Address-to-Data wait states 101 5 Address-to-Data wait states 110 6 Address-to-Data wait states 111 7 Address-to-Data wait states
15:11	00H	Reserved
10:8	111_2	**Read Cycle Data-to-Data Wait states (t_{WRD})** - This bit field represents the number of wait states between burst Data to Data for read transactions. The bit field encodings are the same as those shown for Read Cycle Address-to-First Data Wait States (t_{WRA}).
7:3	00H	Reserved
2:0	111_2	**Read Cycle Additional Recovery Cycles (t_{WRR})** - The local bus defines one recovery cycle between the last data and the next address. This bit field represents the number of additional recovery cycles between the last data and the next address after completing a for read transactions. The bit field is encoded as: 000 0 additional recovery cycles 001 1 additional recovery cycle 010 2 additional recovery cycles 011 3 additional recovery cycles 100 4 additional recovery cycles 101 5 additional recovery cycles 110 6 additional recovery cycles 111 7 additional recovery cycles

14.5.3.2 Memory Bank Write Wait State Registers - MBWWS0:1

The Memory Bank Write Wait State Register (MBWWS) describes wait states during write cycles.

Table 14-6. Memory Bank Write Wait States Register – MBWWS0:1


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rv rv rv rv rv rw rw rw rv rv rv rv rv rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: Bank 0 = 150CH
Bank 1 = 1518H
PCI: N/A

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:19	0000H	Reserved
18:16	111_2	**Write Cycle Address-to-First Data Wait States (t_{WWA})** - This bit field represents the number of wait states between address and the first data for write transactions. Encoded as follows: 000 0 Address-to-Data wait states 001 1 Address-to-Data wait state 010 2 Address-to-Data wait states 011 3 Address-to-Data wait states 100 4 Address-to-Data wait states 101 5 Address-to-Data wait states 110 6 Address-to-Data wait states 111 7 Address-to-Data wait states
15:11	00H	Reserved
10:8	111_2	**Write Cycle Data-to-Data Wait States (t_{WWD})** - This bit field represents the number of wait states between burst Data to Data for write transactions. Bit field encodings are the same as those shown for Write Cycle Address-to-First Data Wait States (t_{WWA})
7:3	00H	Reserved
2:0	111_2	**Write Cycle Additional Recovery Cycles (t_{WWR})** - The local bus defines one recovery cycle between the last data and the next address. This bit field represents the number of additional recovery cycles between the last data and the next address after completing a for write transactions. The bit field is encoded as follows: 000 0 additional recovery cycles 001 1 additional recovery cycle 010 2 additional recovery cycles 011 3 additional recovery cycles 100 4 additional recovery cycles 101 5 additional recovery cycles 110 6 additional recovery cycles 111 7 additional recovery cycles

14.5.4 Memory Bank Waveforms

Programming the wait states for each of the bus cycles allows the memory controller to support SRAM, ROM and Flash memory. Figure 14-5 shows a burst read transaction with a wait state profile of 2,1,1,1.

Table 14-7. Burst Flash Memory, Read Access Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{WRA}	02_2	2
t_{WRD}	01_2	1
t_{WRR}	00_2	0

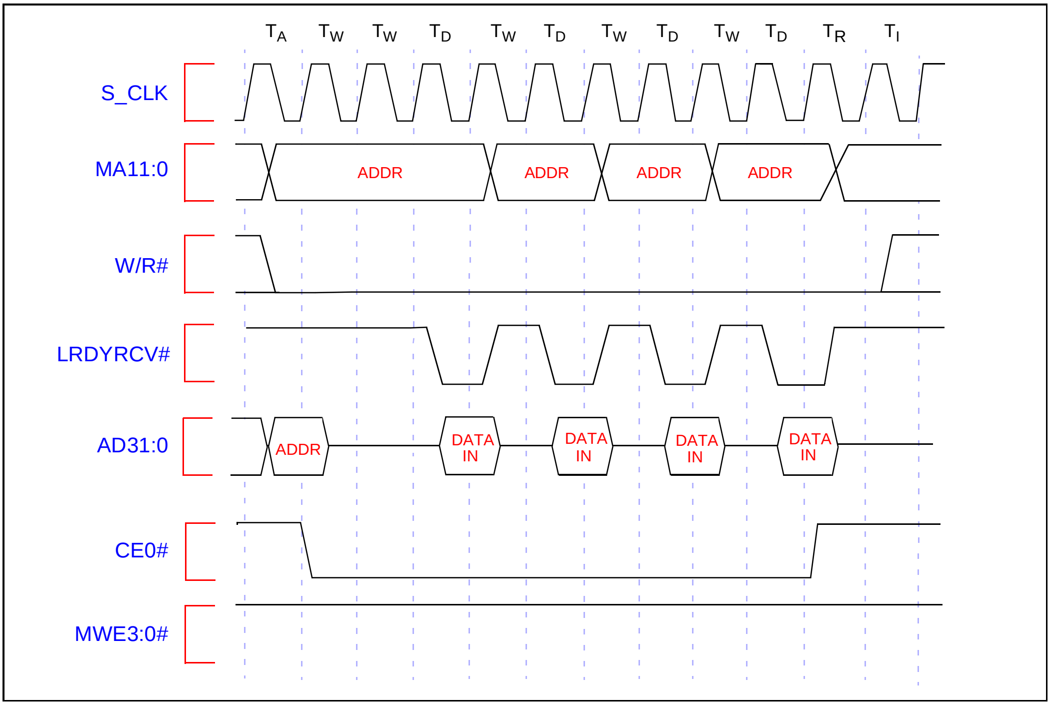


Figure 14-5. 32-Bit Bus, Burst Flash Memory, Read Access with 2,1,1,1 Wait States

Figure 14-6 represents a burst write transaction to Flash memory with a wait state profile of 2,1,1,1. The Extended MWE3:0# control bit in the MBCR is cleared in this example.

Table 14-8. SRAM Write Access Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{WWA}	02_2	2
t_{WWD}	01_2	1
t_{WWR}	00_2	0

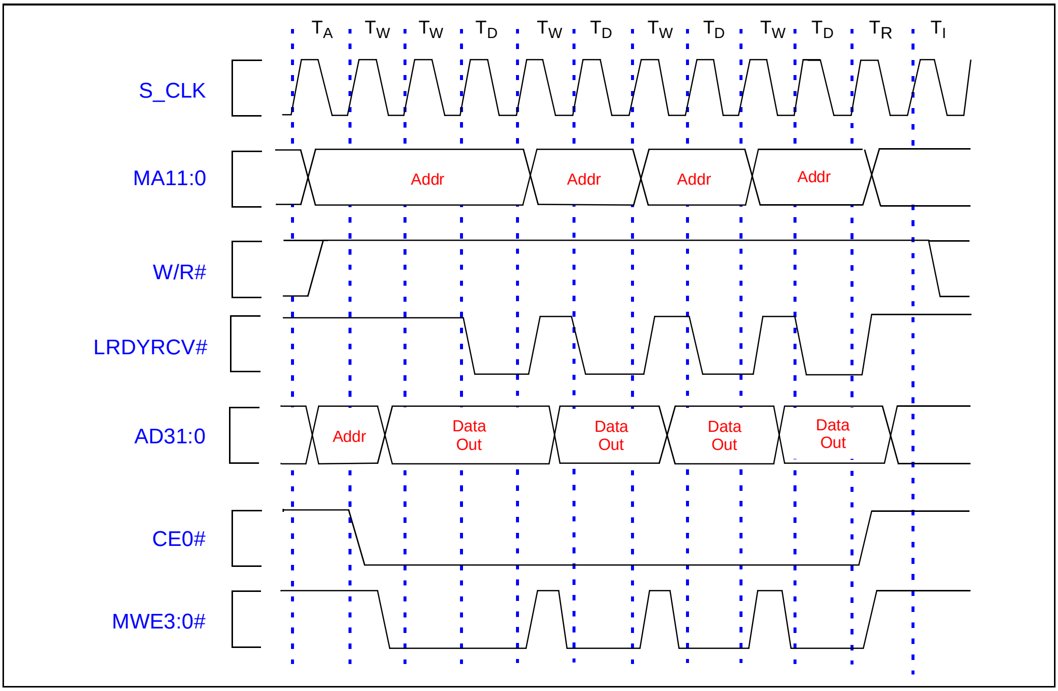


Figure 14-6. 32-Bit Bus, SRAM Write Access with 2,1,1,1, Wait States

Programming the wait states for each of the bus cycles allows the memory controller to support burst transactions with SRAMs. Figure 14-7 shows a read transaction with 0 wait state SRAM.

Table 14-9. SRAM Read Access Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{WRA}	00_2	0
t_{WRD}	00_2	0
t_{WRR}	00_2	0

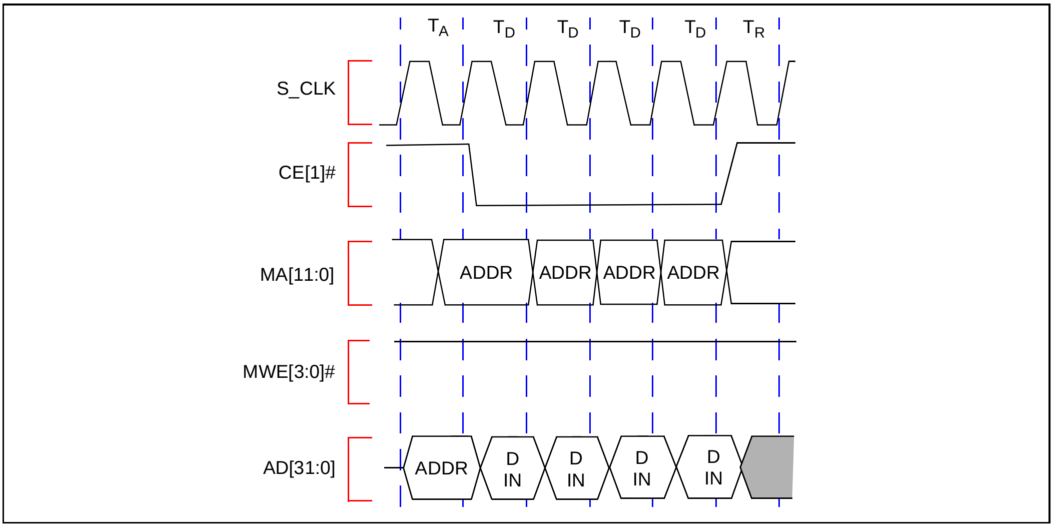


Figure 14-7. 32-Bit Bus, SRAM Read Accesses with 0 Wait States

Figure 14-8 represents a 0 wait state write transaction from SRAM.

Table 14-10. SRAM Write Access Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{WWA}	00_2	0
t_{WWD}	00_2	0
t_{WWR}	00_2	0

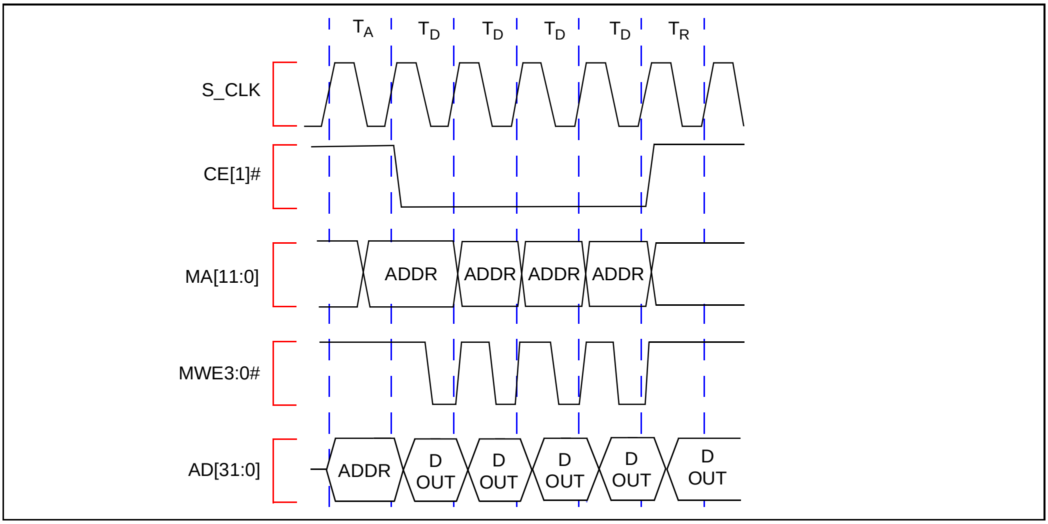


Figure 14-8. 32-Bit Bus, SRAM Write Access With 0 Wait States

14.5.5 Extending Memory Write Enable Signals

The extended MWE3:0# write enable control bit in the MBCR allows the MWE3:0# to be extended during the deassertion period between burst data accesses. In addition, the LRDYRCV# signal assertion is delayed. The characteristics of the other memory controller signals remain the same. Figure 14-9 shows a 2-word burst of an extended MWE3:0# write cycle.

Table 14-11. Write Access with Extended MWE3:0# Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{WWA}	02_2	5
t_{WWD}	01_2	3
t_{WWR}	00_2	1

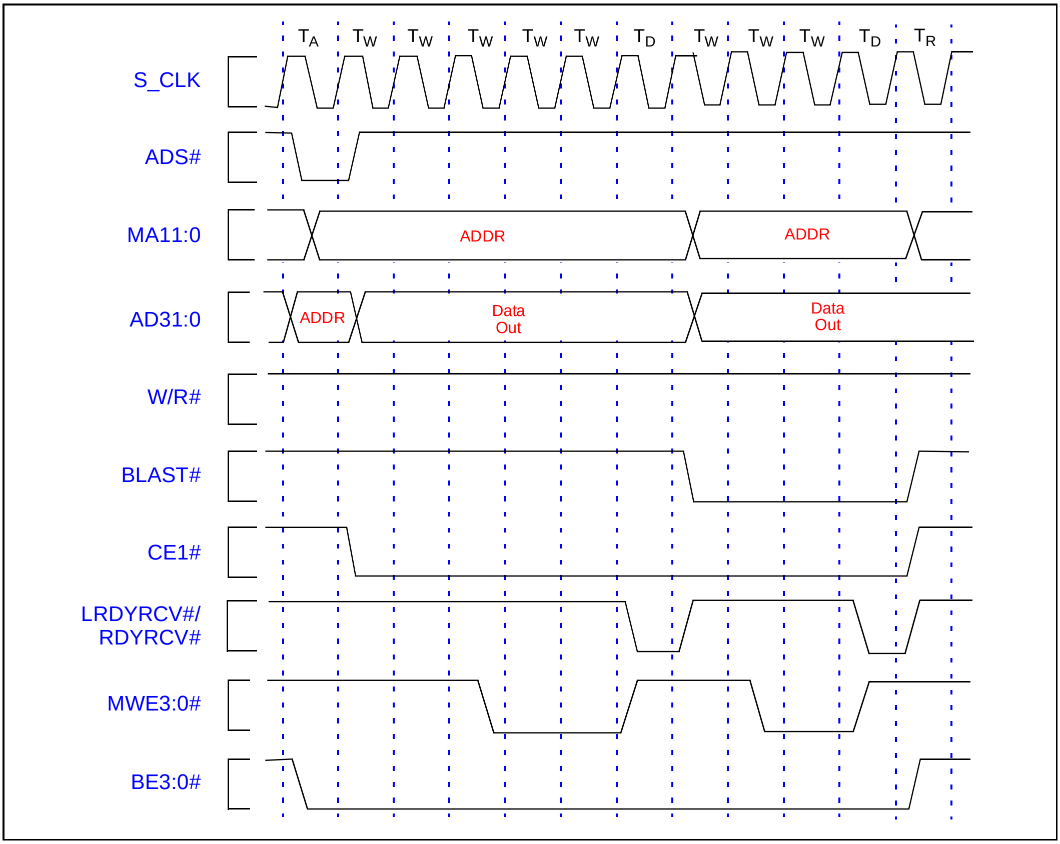


Figure 14-9. 32-Bit Bus, Write Access with Extended MWE3:0#

14.6 DRAM CONTROL

The DRAM bank may be organized as 32-bit without parity or 36-bit with parity. The memory controller provides a direct interface for a minimum of 1 Mbyte and a maximum of 256 Mbytes of DRAM by generating the signals shown in Table 14-12.

Table 14-12. DRAM Control Signals

Signal Name	Source	Description
MA11:0	Memory Controller	Memory Address Bus - Specifies address path to the DRAM.
DP3:0	Memory Controller	DRAM Data Parity, DP3:0 - Specifies the byte wide parity bit for data transfers: • DP3 - Parity value for data on AD31:24 • DP2 - Parity value for data on AD23:16 • DP1 - Parity value for data on AD15:8 • DP0 - Parity value for data on AD7:0
DALE1:0	Memory Controller	DRAM Address Latch Enable: DALE1:0 - Specifies address valid during an address cycle.
DWE1:0#	Memory Controller	DRAM Write Enable: DWE1:0# - Write Cycle. Individual byte enables during write cycles are controlled with the individual CAS7:0# signals. For non-interleaved operation, these signals are identical and can be used interchangeably.
CAS7:0#	Memory Controller	Column Address Strobe. Indicates the presence of a valid column address on the memory address bus MA11:0.
RAS3:0#	Memory Controller	Row Address Strobe. Indicates the presence of a valid row address on the memory address bus MA11:0.
LEAF1:0#	Memory Controller	LEAF OE# control. For non-interleaved DRAM, LEAF1:0# controls the OE#. For interleaved DRAM, the LEAF1:0# signals control the OE# data latches. For non-interleaved operation, these signals are identical and can be used interchangeably.
AD31:0	Local Bus	Multiplexed Address/Data Bus. Data path to and from the DRAM.

The memory controller supports from one to four banks of DRAM organized as 32 or 36 bits wide. The memory banks may be configured as non-interleaved or two-way interleaved. The memory controller supports three different types of DRAM: Fast Page-Mode (FPM), Extended Data Out (EDO) and Burst Extended Data Out (BEDO). Interleaved Fast Page-Mode DRAM is also supported. DRAM refresh is supported through the programmable DRAM refresh counter.

14.6.1 DRAM Organization and Configuration

The memory controller provides a programmable address window for DRAM that decodes local bus addresses and drives the corresponding DRAM control signals. The address window is programmed through the memory controller memory-mapped registers. Additional memory-mapped registers control timings for different speed ratings of DRAM, DRAM bank sizes, DRAM types, DRAM initialization, and DRAM organization.

To prevent bursts from crossing a DRAM page, the maximum burst size for a single data transfer cycle to the memory controller is 2 Kbytes. On-chip bus masters accessing the memory controller are required to adhere to the 2 Kbyte address boundary. The 80960Rx closes the DRAM Page. RAS# deasserts during the first recovery cycle and stays deasserted through ADS#.

DRAM organization is programmable through control bits in the DRAM Bank Control Register (DBCR). The memory controller provides support for up to four banks of non-interleaved DRAM. Up to two banks of non-interleaved DRAM can be connected with each bank containing two leaves. Table 14-13 summarizes the supported DRAM organization and type.

Table 14-13. Supported DRAM Configurations

Interleaved DRAM (Fast Page-mode DRAM Only)	Non-Interleaved DRAM (FPM, EDO or BEDO DRAM)
1 Bank (2 leaves)	1 Bank
2 Banks (4 leaves)	2 Banks
	4 Banks

An example of a single 16 Mbyte bank of DRAM, organized as 32-bit non-interleaved, is shown in Figure 14-10. As shown, the 80960Rx is a direct connect to the non-interleaved memory subsystem (no additional logic is required).

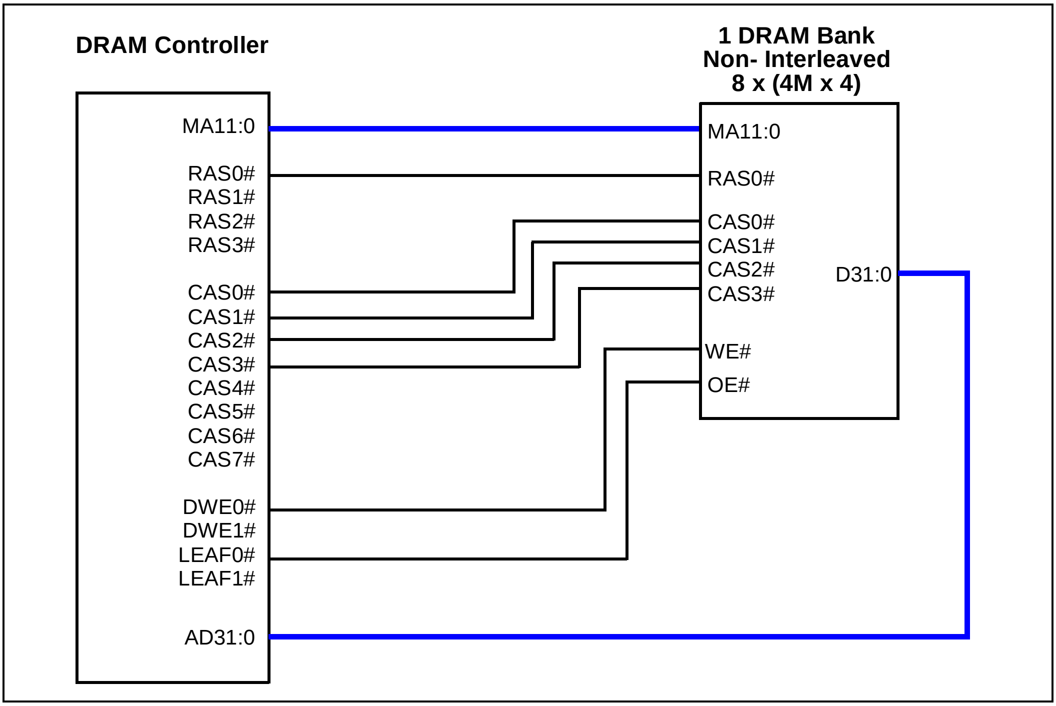


Figure 14-10. Non-Interleaved, 32-Bit, Single Bank, DRAM System

Figure 14-11 shows a sample memory system using 32-bit interleaved DRAM. The memory controller provides eight CAS# signals for the support of interleaved memory. The CAS3:0# signals provide the byte selection for one leaf, while CAS7:4# provide for the second leaf. It is necessary to control external buffer output enables during read transactions in an interleaved memory system. Two signals, LEAF1:0#, are provided to control the multiplexing of data from each memory leaf onto the processor address/data bus. These signals are tied to the OE# pins of the data transceivers in an interleaved memory array. In a non-interleaved memory array, the OE# pins are typically tied to signal ground. Standard DRAM device sizes from 1 Mbit to 64 Mbit are supported without the use of external logic to generate control signals. Two identical write enable signals, DWE1:0#, are provided to control the WE# input of DRAM devices during read and write transactions.

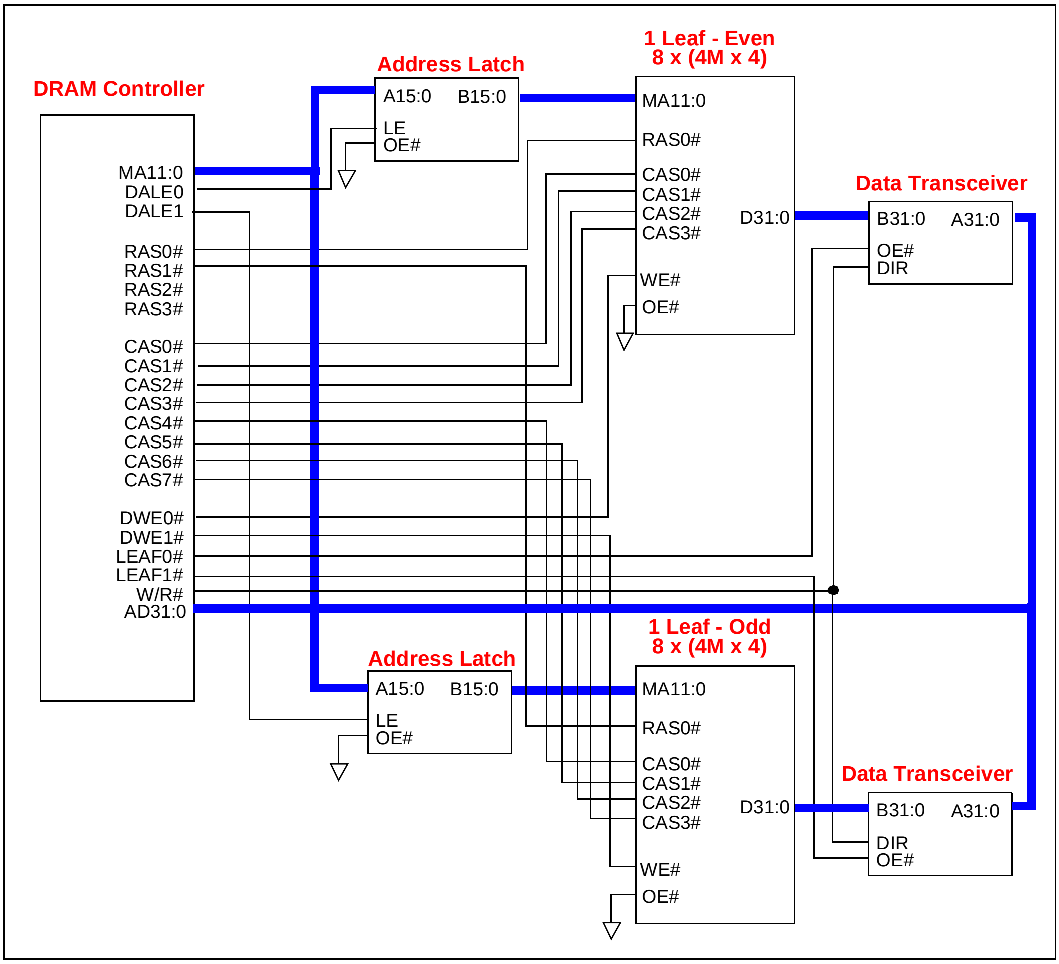


Figure 14-11. Interleaved 32-Bit DRAM System, 1 Bank, 2 Leaves

The DRAM types supported include 1-, 4-, 16- and 64-Mbit devices. These memory types are supported without the use of external logic to generate control signals. The arrangement for each technology is summarized in Table 14-14.

Table 14-14. Supported DRAM Configurations (Symmetric Addressing Only)

DRAM Technology	DRAM Arrangement	Address Size (in Bits)		Bank / Leaf Size[1]	Non-Interleaved DRAM (in Mbytes)		Interleaved DRAM (in Mbytes)	
		Row	Col.		Min.	Max.	Min.	Max.
1 Mbit	1M x 1	10	10	4	4	16	8	16
	256K x 4	9	9	1	1	4	2	4
4 Mbit	4M x 1	11	11	16	16	64	32	64
	1M x 4	10	10	4	4	16	8	16
	256K x 16	9	9	1	1	4	2	4
16 Mbit	16M x 1	12	12	64	64	256	128	256
	4M x 4	11	11	16	16	64	32	64
	1M x 16	10	10	4	4	16	8	16
64 Mbit	16M x 4	12	12	64	64	256	128	256
	4M x 16	11	11	16	16	64	32	64

1. Every bank (or leaf) must use the same memory type. Mixed combinations of FPM, EDO, or BEDO are not permitted. The DRAM bank size must also remain the same among banks (or leaves).

14.6.2 DRAM Addressing

The memory controller drives the DRAM address on the MA11:0 pins. This multiplexed address is ordered to support 1 through 64 Mbyte DRAM arrays. Table 14-15 shows the address bits that are presented on the MA11:0 pins during the row and column address cycle. The ordering depends on the arrangement of the DRAM arrays, either non-interleaved or interleaved.

Table 14-15. MA11:0 Address Bits for Non-Interleaved/Interleaved

MA Bit	Non-Interleaved		Interleaved	
	Row	Column	Row	Column
0	11	2	11	10
1	12	3	12	3
2	13	4	13	4
3	14	5	14	5
4	15	6	15	6
5	16	7	16	7
6	17	8	17	8
7	18	9	18	9
8	19	10	19	20
9	21	20	21	22
10	23	22	23	24
11	25	24	25	26

14.6.3 DRAM Registers

The DRAM controller provides registers for configuring and controlling DRAM. Six memory-mapped registers control the memory controller for independent operation:

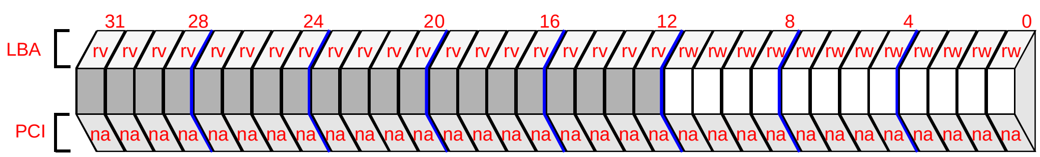
Table 14-16. DRAM Register Summary

Section	Section, Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
14.6.4	DRAM Bank Control Register — DBCR	14-25	32	0000 151CH	N/A
14.6.5	DRAM Base Address Register — DBAR	14-27	32	0000 1520H	N/A
14.6.6	DRAM Read Wait State Register — DRWS	14-28	32	0000 1524H	N/A
14.6.7	DRAM Write Wait State Register — DWWS	14-30	32	0000 1528H	N/A
14.6.8	DRAM Refresh Interval Register — DRIR	14-32	32	0000 152CH	N/A
14.7.1	DRAM Parity Enable Register — DPER	14-35	32	0000 1530H	N/A

14.6.4 DRAM Bank Control Register — DBCR

The DRAM Bank Control Register (DBCR) specifies the parameters used to control the DRAM banks. The DBCR should be programmed after initializing the other DRAM registers. Figure 14-17 shows the register format for the DBCR. This register can be read or written at any time. The DRAM bank enable bits should be disabled prior to modifying the DRAM bank base address and wait-state registers.

Table 14-17. DRAM Bank Control Register — DBCR (Sheet 1 of 2)

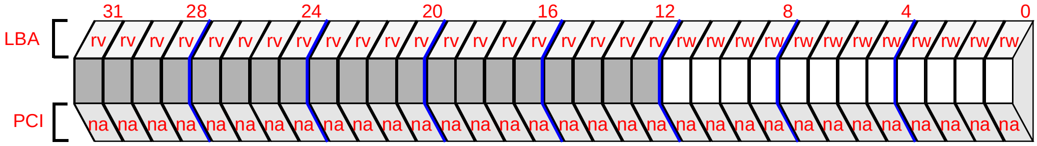

31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 151CH
PCI: N/A

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 Local Bus Address
PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Reserved
11	0_2	**MA11:0 High-Drive Enable Bit** - This bit controls the MA11:0 output. When clear (0) MA11:0 has normal output buffer drive strength. When set (1) MA11:0 has higher output buffer drive strength.
10	0_2	**CAS7:0# High-Drive Enable Bit** - This bit controls the CAS7:0# output. When clear (0) the CAS7:0# has normal output buffer drive strength. When set (1) the CAS7:0# has higher output buffer drive strength.
9	0_2	**RAS3:0# High-Drive Enable Bit** - This bit controls the RAS3:0# output. When clear (0) the RAS3:0# has normal output buffer drive strength. When set (1) the RAS3:0# has higher output buffer drive strength.
8	0_2	**DWE1:0# High-Drive Enable Bit** - This bit controls the DWE1:0# output. When clear (0) the DWE1:0# has normal output buffer drive strength. When set (1) the DWE1:0# has higher output buffer drive strength.

Table 14-17. DRAM Bank Control Register — DBCR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 151CH
PCI: N/A

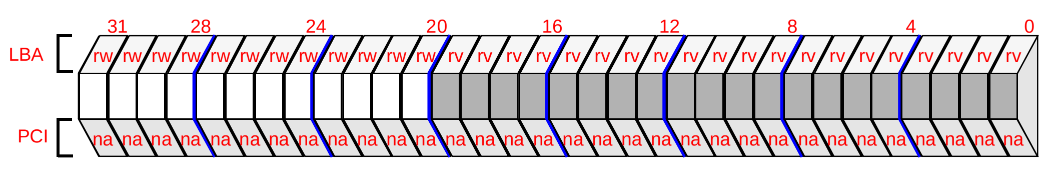
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
7:3	0H	**DRAM Bank Type/Arrangement Field** - This bit field contains the DRAM type and block size of memory connected. The memory connect may be FPM, EDO, or BEDO DRAM. Each bank must be organized as 32-bit wide memory and must consist of a uniform memory type. 000 00 Fast Page-Mode DRAM, 1 Bank 000 01 Fast Page-Mode DRAM, 2 Banks 000 1x Fast Page-Mode DRAM, 4 Banks 001 x0 Fast Page-Mode DRAM, Interleaved, 1 Bank 001 x1 Fast Page-Mode DRAM, Interleaved, 2 Banks 010 00 Extended Data Out (EDO) DRAM, 1 Bank 010 01 Extended Data Out (EDO) DRAM, 2 Banks 010 1x Extended Data Out (EDO) DRAM, 4 Banks 1xx 00 Burst Extended Data Out (BEDO) DRAM, 1 Bank 1xx 01 Burst Extended Data Out (BEDO) DRAM, 2 Banks 1xx 1x Burst Extended Data Out (BEDO) DRAM, 4 Banks
2:1	00_2	**DRAM Bank/Leaf Size** - This bit field defines the bank size of DRAM connected for non-interleaved mode. For Interleaved DRAM, this bit field defines the leaf size. 00 1 Mbyte DRAM per bank/leaf 01 4 Mbytes DRAM per bank/leaf 10 16 Mbytes DRAM per bank/leaf 11 64 Mbytes DRAM per bank/leaf
0	0_2	**DRAM Bank Enable Bit** - This bit enables or disables the DRAM bank. When cleared (0), the memory controller does not assert the DRAM control signals. When set (1), the memory controller decodes the local bus addresses and assert the DRAM control signals when the local bus address falls within the window of address programmed into the DBAR0.

14.6.5 DRAM Base Address Register — DBAR

The DRAM Base Address Register (DBAR) stores the base address for the DRAM. This address must be on an address boundary equal to the total size of the DRAM. For example, a 4 Mbyte DRAM bank must have a starting address located on a 4 Mbyte address boundary. The register definition is shown in Figure 14-18.

Table 14-18. DRAM Base Address Register — DBAR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1520H
PCI: N/A
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

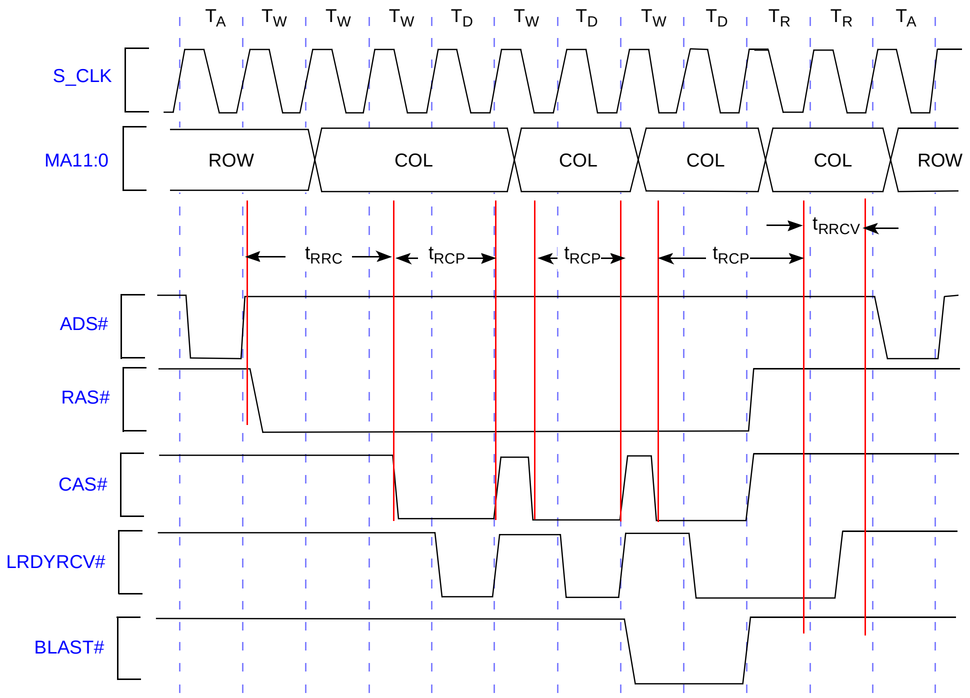
Bit	**Default**	**Description**
31:20	000H	**DRAM Bank Base Address** - These bits define the upper 12 bits of the base address the DRAM bank responds to when addressed from the local bus.
19:00	0 0000H	Reserved

On memory controller reset, the default DRAM base address is indeterminate until it is overwritten by programming DBAR. Since the DRAM bank is disabled at reset, this causes no addressing conflict with the internal data RAM.

NOTE: The i960 core processor does not generate external bus cycles for transactions within the address range of 0 to 0000 03FFH or FF00 0000H to FFFF FFFFH. The processor reserves these address ranges for internal data RAM and memory-mapped registers, respectively. Do not program the DRAM base address register with a value within these reserved address ranges.

14.6.6 DRAM Read Wait State Register — DRWS

The bus cycle timing for DRAM read accesses is programmed through the DRAM Read Wait States Register (DRWS). The software programs the number of wait states for each access in a bus cycle in 1x increments of S_CLK. The symbols t_{RRC}, t_{RCP} and t_{RRCV}, which represent the number of wait states programmed for the address, data and recovery cycles for read transfers, are shown in Figure 14-12. The register definitions for the DRAM Bank Read Wait States Register are shown in Table 14-19. The number of t_{RRC}, t_{RCP} and t_{RRCV} wait states is encoded in two-bit fields, which are also shown in Table 14-19.



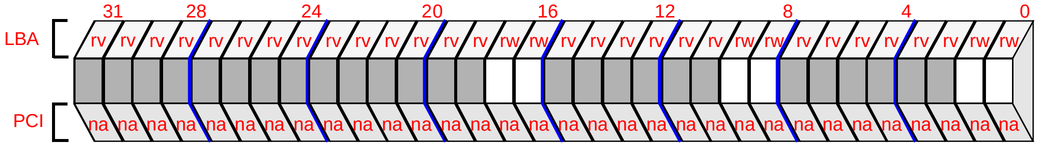
Programmed t_{RRC} = 01

Programmed t_{RCP} = 00

Programmed t_{RRCV} = 01; Total Recovery Cycles = t_{RRCV} + 1

Figure 14-12. DRAM Read Cycle Programmable Parameter Example

Table 14-19. DRAM Bank Read Wait State Register — DRWS

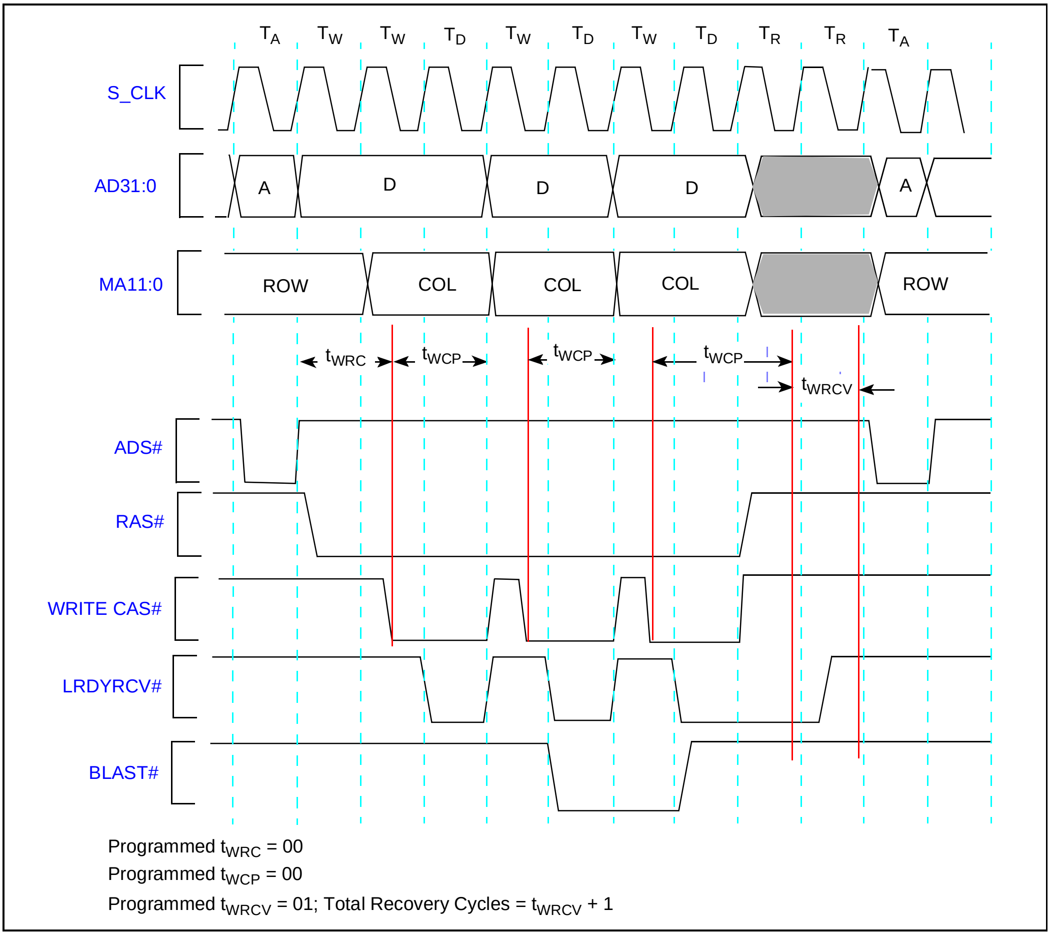

31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rv rv rv rv rv rv rw rw rv rv rv rv rv rv rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1524H
PCI: N/A
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:18	0000H	Reserved
17:16	00_2	**DRAM Read cycle RAS-to-CAS delay (t_{RRC})** - This field affects the number of cycles between the assertion of RAS3:0# and the assertion of CAS7:0#. FPM EDO BEDO 00 1.5 cycles 1 cycles 1 cycles 01 2.5 cycles 2 cycles 2 cycles 10 3.5 cycles 3 cycles 3 cycles 11 4.5 cycles 4 cycles 4 cycles
15:10	00H	Reserved
9:8	00_2	**DRAM Read cycle CAS pulse width (t_{RCP})** - This field affects the number of cycles that CAS7:0# is asserted. Fast Page-Mode DRAM: 0x 1.5 cycles (defaults to 1.5 for FPM DRAM) 10 2.5 cycles 11 3.5 cycles EDO DRAM and BEDO DRAM (this parameter is fixed for EDO or BEDO DRAM types): xx 0.5 cycles
7:2	00H	Reserved
1:0	00_2	**DRAM Read cycle additional recovery wait states (t_{RRCv})** - These are the number of extra wait states that are inserted at the end of a DRAM transaction. The purpose is to increase the RAS precharge time for the DRAM (t_{RP}). 00 0 additional recovery cycles 01 1 additional recovery cycle 10 2 additional recovery cycles 11 3 additional recovery cycles

14.6.7 DRAM Write Wait State Register — DWWS

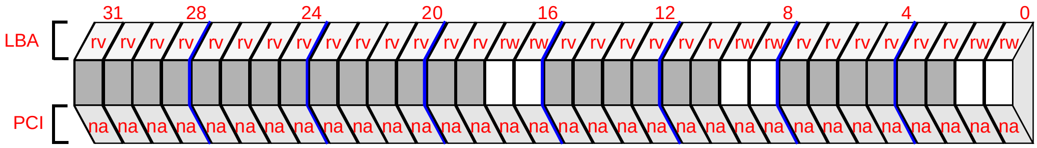
The bus cycle timing for DRAM write accesses is programmed through the DRAM Write Wait States register (DWWS). The software programs the number of wait states for each access in a bus cycle in 1x increments of S_CLK. The symbols t_{WRC}, t_{WCP} and t_{WRCV}, which represent the number of wait states programmed for the address, data and recovery cycles for write transactions, are shown in Figure 14-13. The number of t_{WRC}, t_{WCP} and t_{WRCV} wait states is encoded in two-bit fields as shown in Table 14-20.



Programmed t_{WRC} = 00
Programmed t_{WCP} = 00
Programmed t_{WRCV} = 01; Total Recovery Cycles = t_{WRCV} + 1

Figure 14-13. DRAM Write Cycle Programmable Parameter Example

Table 14-20. DRAM Bank Write Wait State Register — DWWS


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rv rv rv rv rv rv rw rw rv rv rv rv rv rv rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1528H
PCI: N/A
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:18	0000H	Reserved
17:16	00_2	**DRAM Write cycle RAS-to-CAS delay (t_{WRC})** - This field affects the number of cycles between the assertion of RAS3:0# and the assertion of CAS7:0#. FPM EDO BEDO 00 1.5 cycles 1 cycles 1 cycles 01 2.5 cycles 2 cycles 2 cycles 10 3.5 cycles 3 cycles 3 cycles 11 4.5 cycles 4 cycles 4 cycles
15:10	00H	Reserved
9:8	00_2	**DRAM Write cycle CAS pulse width (t_{WCP})** - This field affects the number of cycles that CAS7:0# is asserted. Fast Page-Mode DRAM: **0x 1.5 cycles (defaults to 1.5 for FPM DRAM)** 10 2.5 cycles 11 3.5 cycles EDO DRAM and BEDO DRAM: xx 0.5 cycles
7:2	00H	Reserved
1:0	00_2	**DRAM Write cycle additional recovery wait states (t_{WRCV})** - The number of extra wait states inserted at the end of a DRAM transaction. The purpose is to increase RAS precharge time for DRAM (t_{RP}). 00 0 additional recovery cycles 01 1 additional recovery cycle 10 2 additional recovery cycles 11 3 additional recovery cycles

14.6.8 DRAM Refresh Interval Register — DRIR

The memory controller supports CAS# Before RAS# (CBR) refresh cycles for DRAM devices. Figure 14-14 shows an example of a typical CBR refresh cycle.

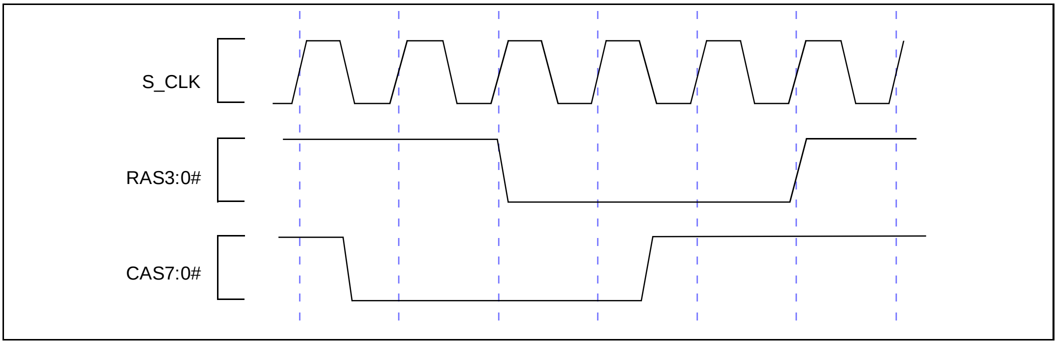


Figure 14-14. CAS#-Before-RAS# DRAM Refresh

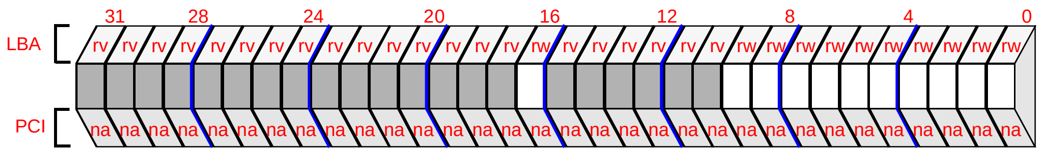
The internal DRAM Refresh Interval Register (DRIR) (Table 14-21) provides the time delay between DRAM refresh cycles and is programmed in increments of S_CLK. The value programmed is determined as follows:

Programmed Value = (DRAM Refresh Cycle Rate x Input Clock Frequency)

The register provides ten bits for the programmed value that corresponds to a time delay range of 0 to 34.1 µs at 33 MHz.

The DRAM controller performs hidden refreshes which can occur in the middle of burst transfers on the local bus.

Table 14-21. DRAM Refresh Interval Register — DRIR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 152CH
PCI: N/A
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:17	0000H	Reserved
16	0	**DRAM Refresh Disable Bit** - This bit disables the DRAM refresh cycles from occurring. When cleared (0) the DRAM refresh counter decrements the value found in the DRAM refresh interval value field until a zero value is reached. At that time, the DRAM refresh initiates a CBR cycle. When set (1) the DRAM refresh counter is disabled and does not generate any CBR cycles.
15:10	00H	Reserved
9:00	78H	**DRAM Refresh Interval Value** - This bit field defines the number of 1x S_CLK cycles between generating refresh cycles. The DRAM refresh interval defaults to a value that meets the minimum interval typically used with the DRAM types supported on the 80960Rx.

Using a standard DRAM refresh cycle rate of 15.625 µs, the programmed value for a 33 MHz clock is calculated as follows:

DRAM Refresh Interval = (15.625 µs x 33 MHz) = 516 = 0x0000 0204

An initial pause of 100 to 200 µs after power-up followed by eight RAS3:0# cycles is typically required before proper DRAM device operation is assured. This requirement is satisfied by using a 200 µs delay between memory system power-up and memory controller reset, and a default refresh interval of approximately 3.6 µs. The default value in the DRAM Refresh Interval Register is 120 or 0000 0078H, which is 4.8 µs with a 25 MHz clock or 3.6 µs with a 33 MHz clock.

14.7 ERROR CHECKING AND REPORTING

The memory controller provides two mechanisms for reporting error conditions. The first is DRAM parity and the second is a bus monitor used to detect invalid local bus addresses and when no RDYRCV# signal is returned to signify valid data.

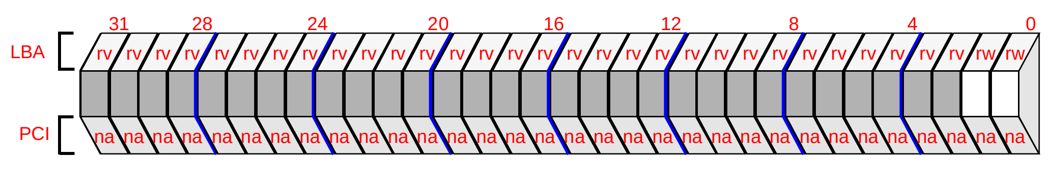
Table 14-22. Error Checking and Reporting Register Summary

Section	Section, Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
14.7.1	DRAM Parity Enable Register — DPER	14-35	32	0000 1530H	N/A
14.7.2	Bus Monitor Enable Register — BMER	14-36	32	0000 1534CH	N/A
14.7.3	Memory Error Address Register — MEAR	14-37	32	0000 1538H	N/A
14.7.4	Local Processor Interrupt Status Register — LPISR	14-38	32	0000 153CH	N/A

14.7.1 DRAM Parity Enable Register — DPER

The use of parity is programmable through the DRAM Parity Enable Register (DPER), shown in Figure 14-23. When data parity is enabled, the memory controller generates a parity bit for each byte written to DRAM, and presents it to the parity bus DP3:0. Parity is checked on all DRAM read accesses when enabled.

Table 14-23. DRAM Parity Enable Register — DPER


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1530H
PCI: N/A
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
			RO = Read Only
			RW = Read/Write

Bit	Default	Description
31:2	0000 0000H	Reserved
1	0_2	**DRAM Parity Polarity Bit** - This bit defines the parity polarity. When clear (0) Even Parity Checking and Generation are performed When set (1) Odd Parity Checking and Generation are performed.
0	0_2	**DRAM Parity Enable Bit** - This bit enables parity checking and generation. When clear (0) Parity Checking and Generation is disabled. When set (1) Parity Checking and Generation is enabled and an interrupt is generated upon detecting a parity error.

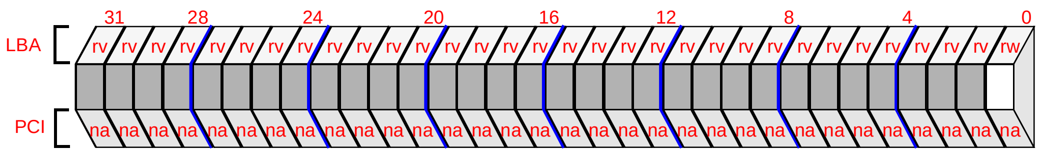
Upon detection of a parity error, the 30-bit address of the faulty memory location is latched and stored in the Memory Error Address Register (MEAR), Table 14-25. The memory controller detects parity errors for any on-chip bus master. These include the primary ATU, secondary ATU, DMA channel 0, 1 or 2 and the i960 core processor. Upon detecting a parity error, the faulty address is latched and an interrupt is generated. The memory controller detects when the i960 core processor is the bus master and sets the parity error status bit in the local processor status register and generates an NMI#. When the i960 core processor is not the bus master, the memory controller notifies the other bus masters of the error condition. The bus masters then latch the error and generate an NMI# to the i960 core processor.

14.7.2 Bus Monitor Enable Register — BMER

The memory controller bus monitor examines all bus accesses to any memory region configured for an external ready. When RDYRCV# is not returned to terminate an access, the processor stalls. Under normal conditions, however, the application can enable or disable the interrupt generated to the i960 core processor from the memory controller. When the valid data is not returned within 127 S_CLK periods, the memory controller asserts the ready signal, LRDYRCV#, which terminates the current data cycle. When the bus monitor interrupt enable bit in the bus monitor enable register is set, the Memory Controller also asserts a bus fault signal to the on-chip bus masters when the timer expires. The on-chip bus master generates an interrupt to the i960 core processor when it receives the bus fault signal. The memory controller is responsible for generating the interrupt when the i960 core processor is the bus master.

The external bus monitor is enabled by programming the Bus Monitor Enable Register (BMER) as shown in Table 14-24. On memory controller reset, the bus monitor interrupt is disabled.

Table 14-24. Bus Monitor Enable Register — BMER


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	1534H	**Legend:** NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

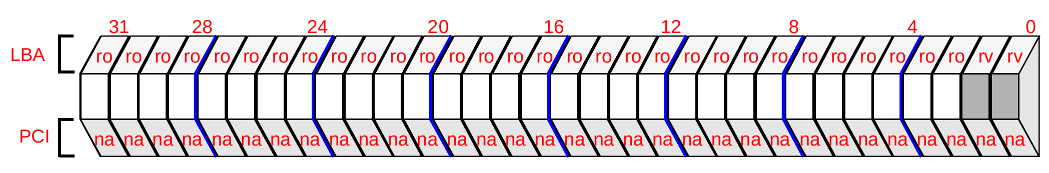
PCI:	N/A	

Bit	Default	Description
31:1	0000 0000H	Reserved
0	0_2	**Bus Monitor Interrupt Enable Bit** - This bit enables the assertion of the bus fault to bus masters when the bus monitor timer expires. It also enables the generation of an interrupt to the i960 core processor when the bus monitor timer expires and the bus master is the i960 core processor. When clear (0), the memory controller does not signal a bus fault to any bus master and does not generate an NMI interrupt to the i960 core processor. When set (1) the Memory Controller signals a bus fault to all bus masters when the bus monitor timer expires and generates an NMI interrupt to the i960 core processor when the core processor is the bus master.

14.7.3 Memory Error Address Register — MEAR

Upon detecting a parity error or bus fault condition, the 30-bit address that generates the fault is latched in the Memory Error Address Register (MEAR). Interrupt service routines can generate individual bus cycles to determine the exact byte address that generated the error condition. The MEAR retains the address until the i960 core processor clears the respective status bit in the local processor status register, primary or secondary ATU status register or in the DMA channel status register(s). When multiple errors occur, the MEAR register preserves the first address that generated the error, however, multiple error status bits may be set.

Table 14-25. Memory Error Address Register — MEAR


31
28
24
20
16
12
8
4
0
LBA
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

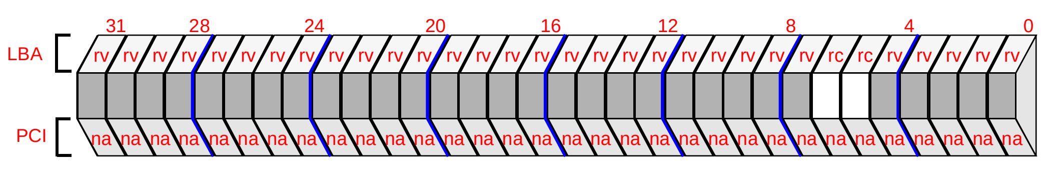
LBA: 1538H
PCI: N/A
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000H	**Memory Error Address Field** - These bits define the upper 30 bits of local bus address that generated either a parity error or a bus fault condition. Clearing the error status bits in the local processor status register for i960 core processor errors allows the MEAR to latch new error addresses. When the DMA units or the ATUs generate the error, status bits in their respective status registers must be cleared to allow the MEAR to latch new error addresses.
01:00	00_2	Reserved

14.7.4 Local Processor Interrupt Status Register — LPISR

Upon detecting a parity error or bus fault condition, when the core was local bus master, the memory controller sets the corresponding bit within the Local Processor Interrupt Status Register (LPISR). This register is used as a status for the i960 core processor to differentiate between the two error conditions. Clearing the status bit within the LPISR register clears the memory controller interrupt and allows additional memory controller interrupts to be generated. The interrupt is cleared by writing a 1 to the respective interrupt status bit.

Table 14-26. Local Processor Interrupt Status Register — LPISR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rv rv rv rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 153CH
PCI: N/A
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
			RO = Read Only
			RW = Read/Write

Bit	Default	Description
31:7	0000 000H	Reserved
6	0_2	**Memory Fault Interrupt Status Bit** - This bit signifies a memory fault error condition occurred, when the core was local bus master. When cleared (0) no memory fault (parity error) interrupt generated. When set (1) a memory fault (parity error) interrupt is pending.
5	0_2	**Local Bus Fault Interrupt Status Bit** - This bit signifies a local bus fault error condition occurred, when the core was local bus master. When cleared (0) no local bus fault interrupt generated. When set (1) a local bus fault interrupt is pending.
4:0	00H	Reserved

14.8 DRAM WAVEFORMS

Waveforms showing FPM, EDO and BEDO DRAM read and write cycles are contained in the following sections. Also included are interleaved and non-interleaved FPM examples.

14.8.1 Non-Interleaved Fast Page-Mode DRAM Waveform

Figure 14-15 and Figure 14-16 represent non-interleaved FPM DRAM system read and write cycle waveforms. The programmed timings used in these two examples are shown in Table 14-27.

Table 14-27. FPM (Non-Interleaved) DRAM Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{RRC}	01_2	2.5
t_{RCP}	00_2	1.5
t_{RRCV}	00_2	0
t_{WRC}	00_2	1.5
t_{WCP}	00_2	1.5
t_{WRCV}	01_2	1

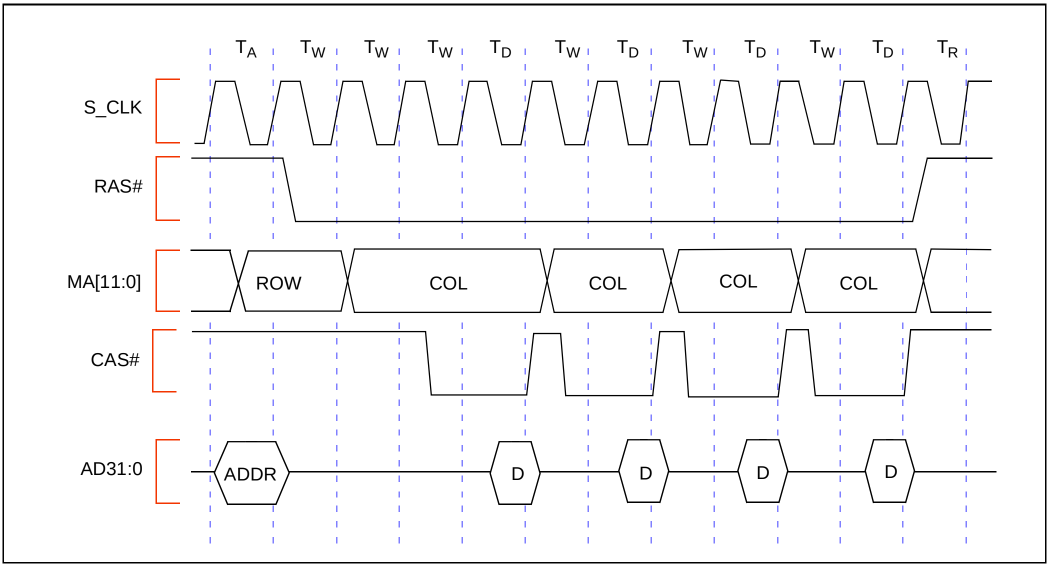


Figure 14-15. FPM DRAM System Read Access, Non-Interleaved, 3,1,1,1, Wait States

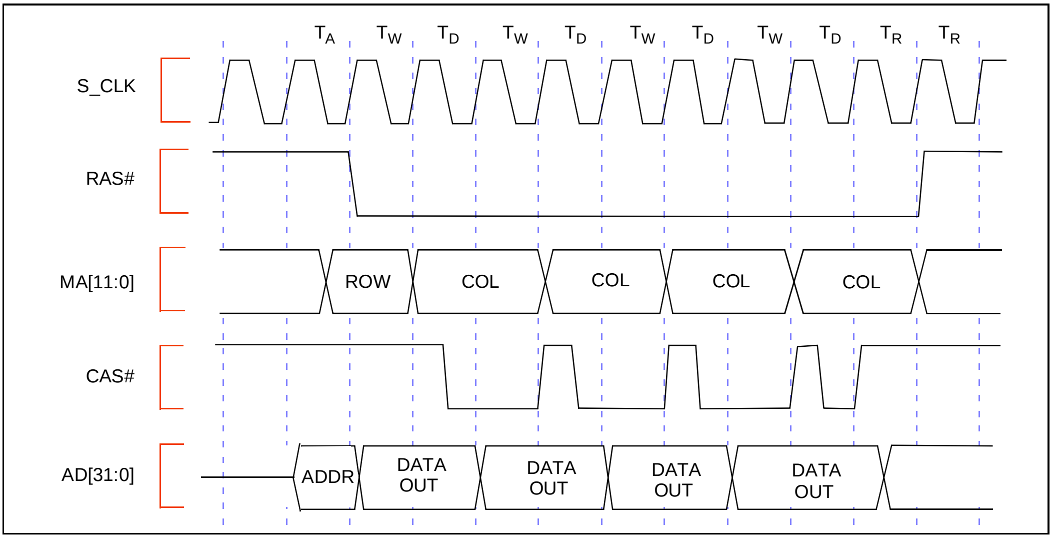


Figure 14-16. FPM DRAM System Write Cycle

14.8.2 Interleaved FPM DRAM Waveform

The memory banks may be configured as an interleaved memory region consisting of up to two banks, where each bank contains two leaves of DRAM. The maximum interleaved configuration is 256 Mbytes organized as two leaves with each leaf containing two banks of DRAM. The memory controller provides eight CAS7:0# signals for the support of interleaved memory:

- CAS3:0# signals provide the byte selection for leaf 0
- CAS7:4# signals provide byte selection for leaf 1

It is necessary to control output enables during read transactions in an interleaved memory system. Two signals, LEAF1:0#, control the multiplexing of data from each memory leaf onto the processor address/data bus. These signals may be tied to the OE# pins of the DRAM devices in an interleaved memory array. The LEAF1:0# signals are generated when the DRAM type selected is FPM, interleaved in the DBCR. Refer to section 14.6.4, DRAM Bank Control Register — DBCR (pg. 14-25).

The QA31:0 and QB31:0 signals refer to the even and odd leaf (respectively) data transceiver outputs between the DRAM and the 80960Rx. For an interleaved DRAM system, Figure 14-17 and Figure 14-18 represent typical read and write transactions. The programmed timings used in the examples are shown in Table 14-28.

Table 14-28. FPM (Interleaved) DRAM Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{RRC}	00_2	1.5
t_{RCP}	00_2	1.5
t_{RRCV}	00_2	0
t_{WRC}	00_2	1.5
t_{WCP}	00_2	1.5
t_{WRCV}	01_2	1

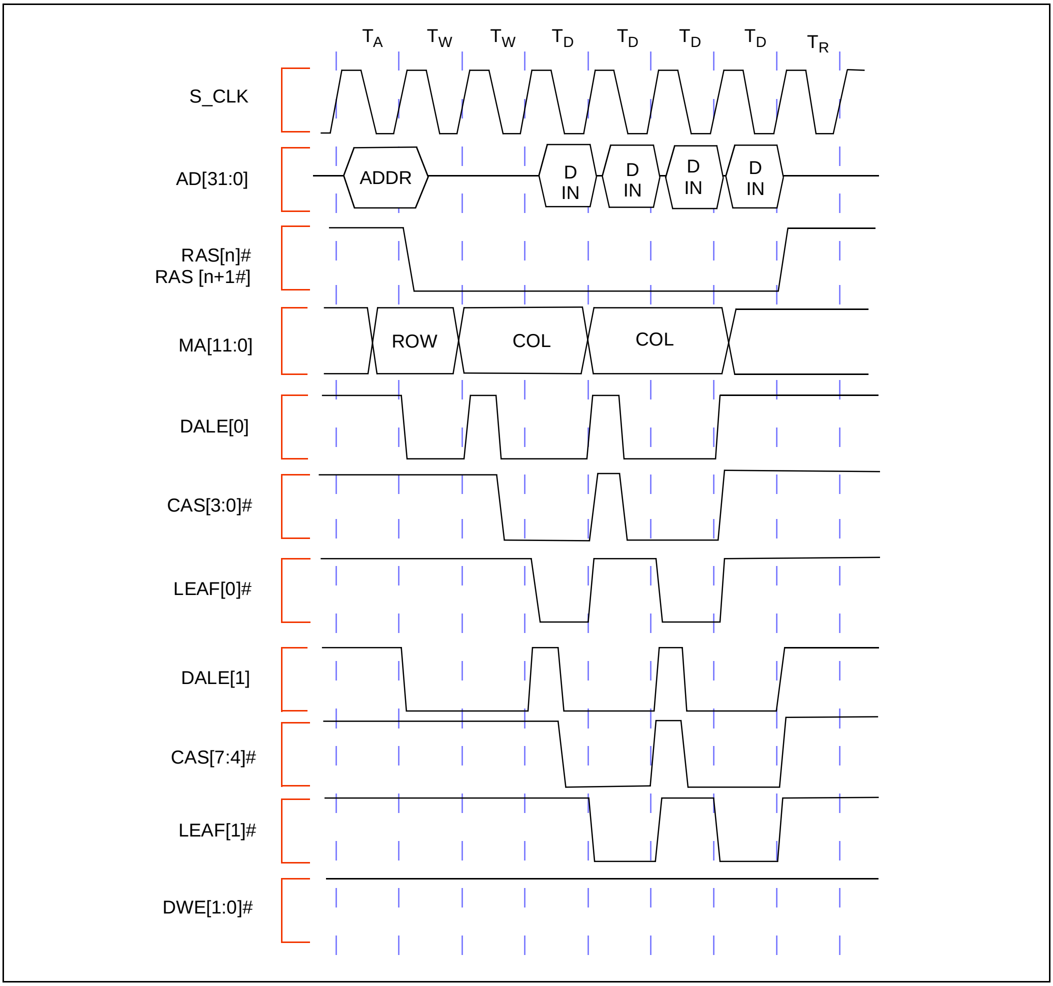


Figure 14-17. FPM DRAM System Read Access, Interleaved, 2,0,0,0 Wait States

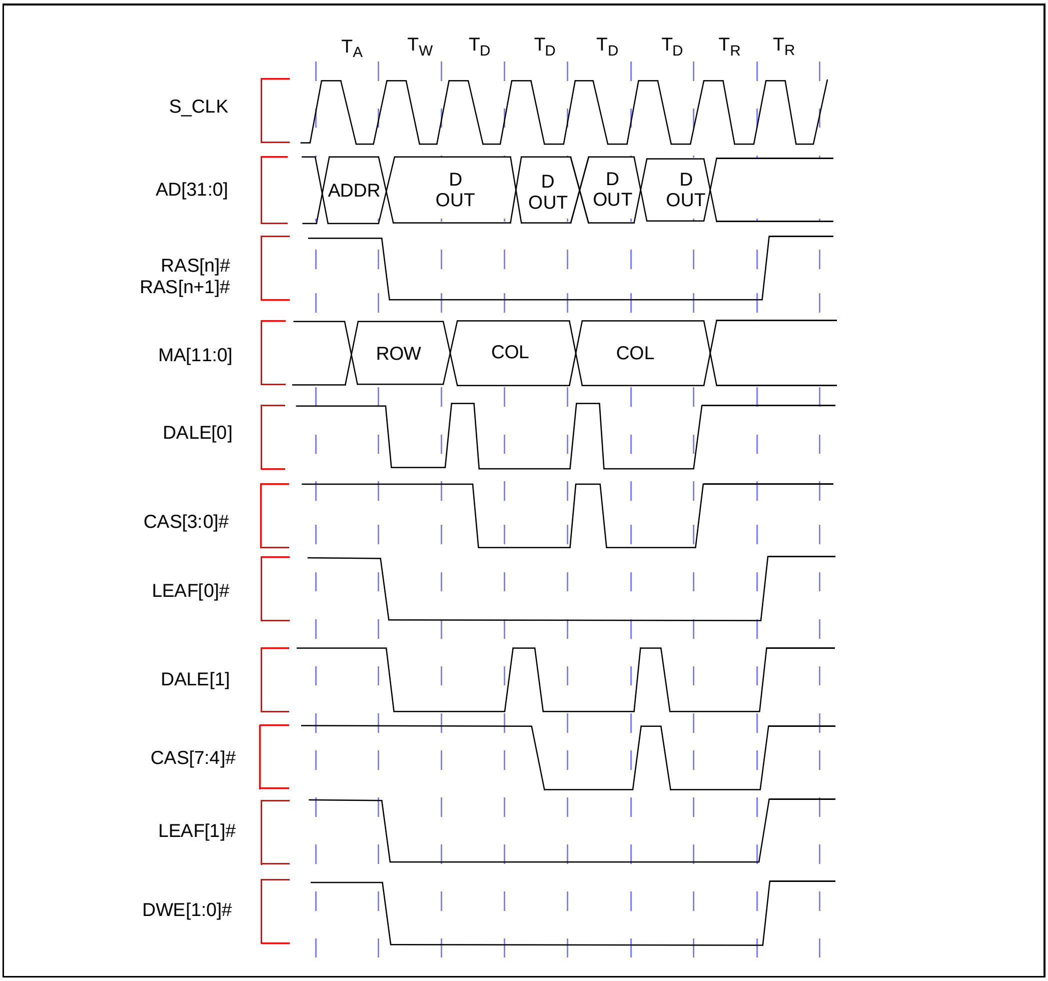


Figure 14-18. FPM DRAM System Write Access, Interleaved, 1,0,0,0 Wait States

14.8.3 EDO DRAM Waveform

Figure 14-19 and Figure 14-20 represent EDO DRAM system read and write cycle waveforms. The programmed timings are shown in Table 14-29.

Table 14-29. EDO DRAM Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{RRC}	01_2	2
t_{RCP}	Fixed at 00_2 for EDO DRAM	0.5
t_{RRCV}	00_2	0
t_{WRC}	00_2	1.5
t_{WCP}	Fixed at 00_2 for EDO DRAM	0.5
t_{WRCV}	00_2	0

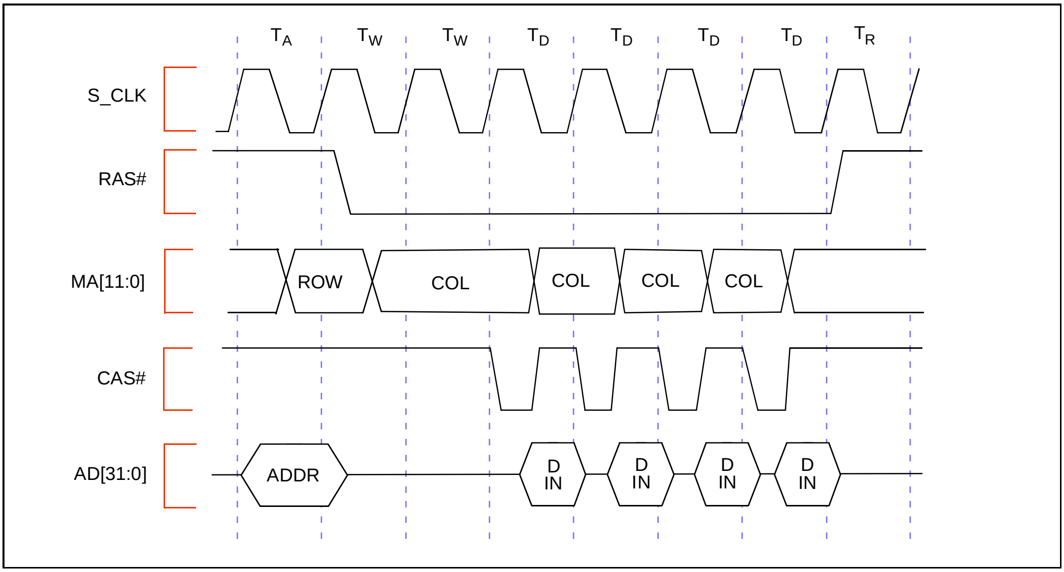


Figure 14-19. EDO DRAM System Read Access, 2,0,0,0, Wait States

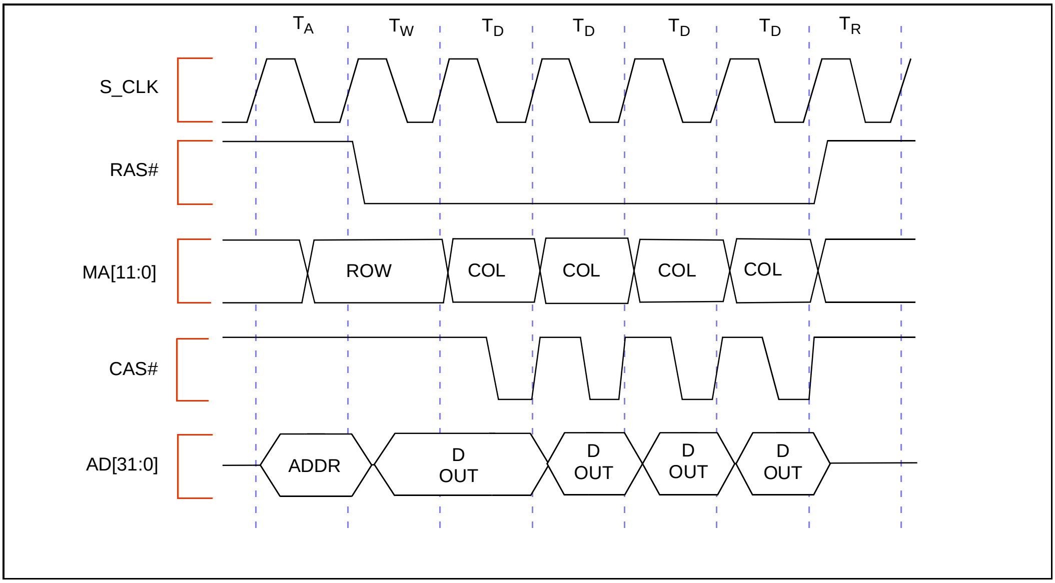


Figure 14-20. EDO DRAM System Write Access, 1,0,0,0 Wait States

14.8.4 BEDO DRAM Waveform

Figure 14-21 and Figure 14-22 represent BEDO DRAM system read and write cycle waveforms. The programmed timings for these examples are shown in Table 14-30.

Table 14-30. BEDO DRAM Example Programming Summary

Timing Symbol	Programmed Value	Cycles
t_{RRC}	01_2	2
t_{RCP}	Fixed at 00_2 for BEDO DRAM	0.5
t_{RRCV}	00_2	0
t_{WRC}	00_2	1.5
t_{WCP}	Fixed at 00_2 for BEDO DRAM	0.5
t_{WRCV}	00_2	0

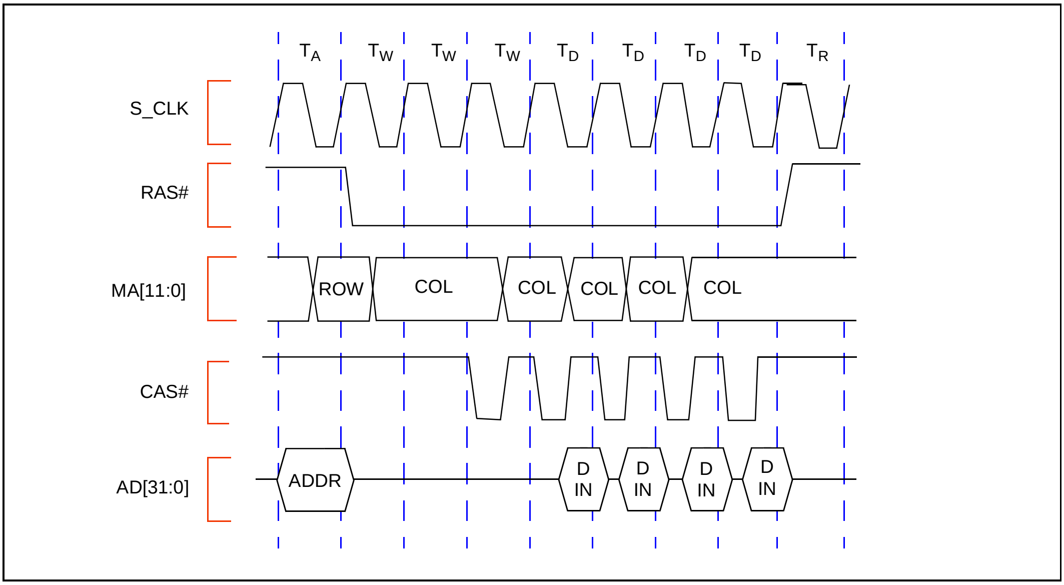


Figure 14-21. BEDO DRAM System Read Access, 3,0,0,0, Wait States

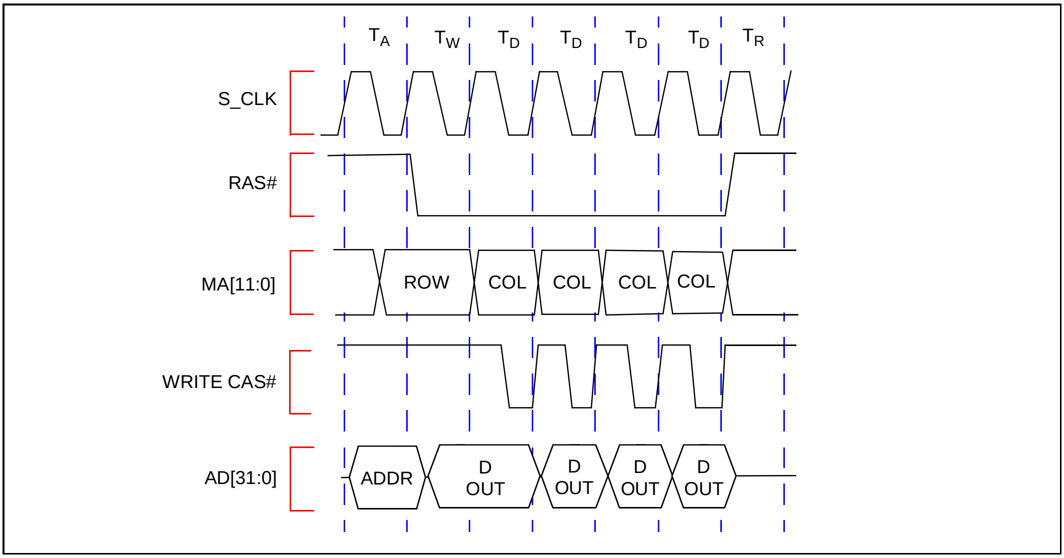


Figure 14-22. BEDO DRAM System Write Access, 1,0,0,0 Wait States

14.9 INITIALIZING DRAM DEVICES

All three DRAM types, FPM, EDO and BEDO require a minimum of eight CAS# Before RAS# cycles prior to the first memory access. In addition, BEDO devices require programming the device to place the DRAM array in linear or interleave burst mode.

To satisfy the initialization cycles required by all three DRAM types, the memory controller uses the refresh counter to generate the CBR (CAS# before RAS#) cycles. The application must wait until at least eight CBR cycles have been performed prior to the first access.

BEDO DRAM requires an additional programming cycle to specify the burst mode. This programming cycle is initiated when the DBCR register is programmed with a value selecting the BEDO DRAM type. Once the DBCR register has been written, the WCBR is initiated. One CBR cycle must be performed after the WCBR (CAS# before RAS# with WE# held low) cycle for the BEDO DRAMs to exit the programming mode. Figure 14-23 shows the WBCR cycle generated by the memory controller.

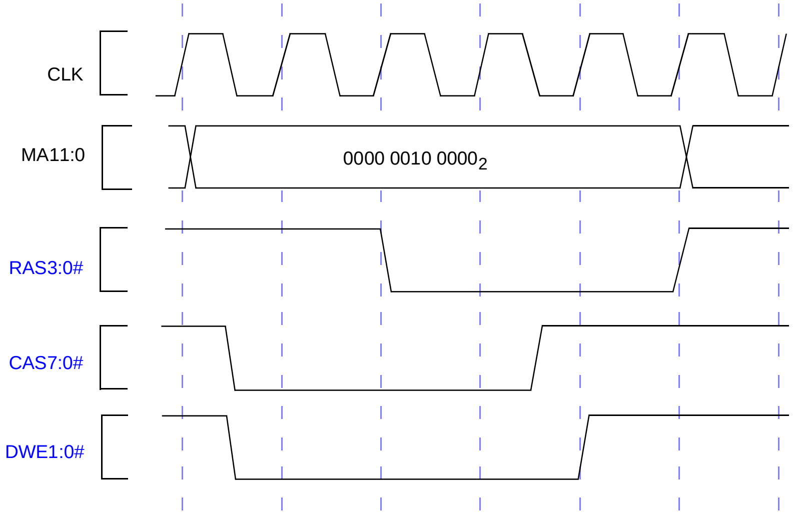


Figure 14-23. BEDO WBCR Program Cycle

The value placed on the MA11:0 bus programs the BEDO DRAM. MA0 determines the burst mode. The memory controller programs the BEDO DRAM for linear burst mode. The delay period before the DRAM access is a combination of eight CBR cycles, programming the DRAM controller, and one additional CBR cycle.

14.10 OVERLAPPING MEMORY REGIONS

Applications can program the address windows for which the memory controller decodes and generates memory cycles. However, certain address within the local bus address space are reserved for memory-mapped registers and ATU-outbound translation. Memory windows can be inadvertently programmed such that they overlap the reserved address space and other memory controller windows. Table 14-31 summarizes memory precedence used when this overlapping occurs.

Table 14-31. Memory Precedence

Priority	Address Region
Highest	Memory-Mapped Register Address Space
	Primary Outbound Address Translation Unit Address Space
	Secondary Outbound Address Translation Unit Address Space
	DRAM Address Space
	Memory Bank 0 Address Space
Lowest	Memory Bank 1 Address Space

15

PCI-TO-PCI BRIDGE UNIT

CHAPTER 15
PCI-TO-PCI BRIDGE UNIT

15.1 OVERVIEW

The PCI-to-PCI bridge unit “extends” the PCI bus beyond its limited physical constraint of ten electrical PCI loads. The bridge unit uses *hierarchical buses* — each bus in the hierarchy is electrically a separate entity yet all buses within the hierarchy are logically one bus. The PCI-to-PCI bridge unit does not increase the bandwidth of a PCI bus, it only extends that bus for applications requiring more I/O components than PCI electrical specifications allow.

The bridge unit provides a connection path between two independent PCI buses. Its primary function is to allow transactions between a master on one PCI bus and a target on the other PCI bus. The PCI-to-PCI bridge unit’s features include:

- Full compliance to the *PCI Local Bus Specification*, revision 2.1
- Full compliance to the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0
- Independent 32-bit primary and secondary PCI buses with support for concurrent operations in either direction
- Separate memory and I/O address spaces on the bridge’s secondary side
- Two 64-byte posting buffers for both upstream and downstream transactions
- VGA palette snooping and VGA compatible addressing on the secondary bus

In addition, the PCI-to-PCI bridge unit supports:

- 64-bit addressing mode from the secondary PCI interface
- Private device configuration and address space for private PCI devices on the secondary PCI bus
- A “special mode” for positive decoding on the primary and secondary interfaces, described in Section 15.5, ”ADDRESS DECODING” (pg. 15-9)

15.2 THEORY OF OPERATION

The bridge unit operates as an address filter between the primary and the secondary PCI buses. PCI supports three separate address spaces:

- 4 Gbyte memory address space
- 64 Kbyte I/O address space (with 16-bit addressing)
- Separate configuration space

A PCI-to-PCI bridge is programmed with a contiguous range of addresses in the memory and I/O address spaces, which then become the secondary PCI address space. Any address on the primary side which falls within the programmed secondary space is forwarded from the primary to the secondary side. The bridge ignores addresses outside the secondary space.

The secondary side works in reverse of the primary side: it ignores addresses within the programmed secondary address space and forwards addresses outside the secondary space to the primary side as illustrated in Figure 15-1.

The PCI-to-PCI bridge's primary and secondary interfaces each implement *PCI Local Bus Specification*, revision 2.1-compliant master and target devices. A PCI transaction initiated on one side of the bridge addresses the initiating bus bridge interface as a target; the target bus interface operating as a master device completes the transaction. The bridge is logically transparent to PCI devices on either side.

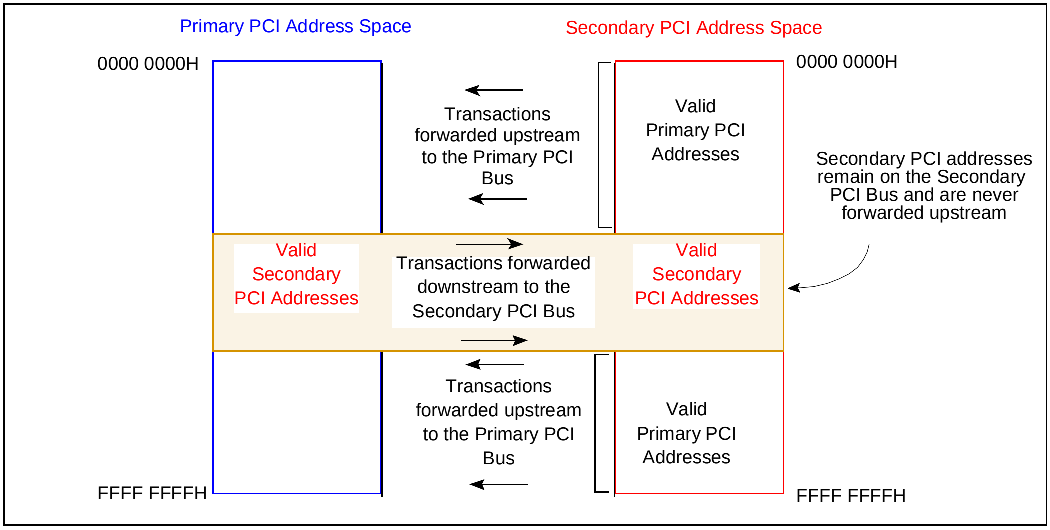

Primary PCI Address Space
Secondary PCI Address Space
0000 0000H
0000 0000H
Transactions forwarded upstream to the Primary PCI Bus
Valid Primary PCI Addresses
Secondary PCI addresses remain on the Secondary PCI Bus and are never forwarded upstream
Valid Secondary PCI Addresses
Transactions forwarded downstream to the Secondary PCI Bus
Valid Secondary PCI Addresses
Transactions forwarded upstream to the Primary PCI Bus
Valid Primary PCI Addresses
FFFF FFFFH
FFFF FFFFH

Figure 15-1. Bridge Operation

15.3 ARCHITECTURAL DESCRIPTION

Figure 15-2 identifies the PCI-to-PCI bridge unit's four major functional blocks. Refer to the following sections for a functional overview:

- 15.3.1, Primary PCI Interface (pg. 15-3)
- 15.3.2, Secondary PCI Interface (pg. 15-4)
- 15.3.3, Buffers (pg. 15-5)
- 15.3.4, Configuration Registers (pg. 15-5)

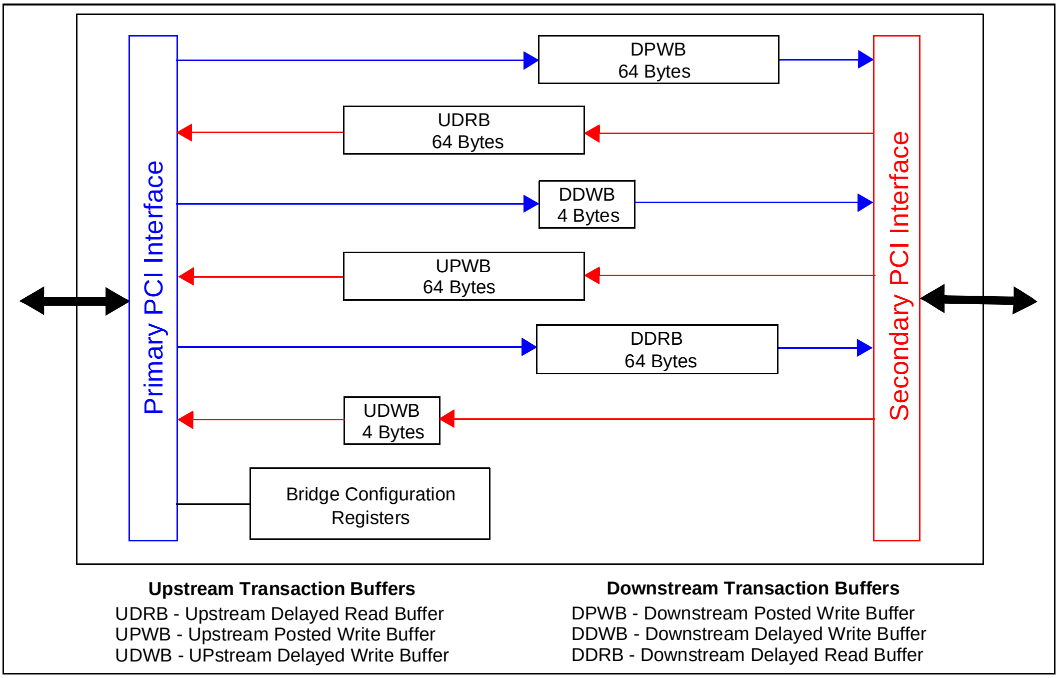

DPWB
64 Bytes
UDRB
64 Bytes
DDWB
4 Bytes
UPWB
64 Bytes
DDRB
64 Bytes
UDWB
4 Bytes
Bridge Configuration Registers
Primary PCI Interface
Secondary PCI Interface
Upstream Transaction Buffers
UDRB - Upstream Delayed Read Buffer
UPWB - Upstream Posted Write Buffer
UDWB - UPstream Delayed Write Buffer
Downstream Transaction Buffers
DPWB - Downstream Posted Write Buffer
DDWB - Downstream Delayed Write Buffer
DDRB - Downstream Delayed Read Buffer

Figure 15-2. PCI-to-PCI Bridge Unit Block Diagram

15.3.1 Primary PCI Interface

The primary PCI interface functions as either a target or initiator of a PCI bus transaction. Typically, the primary PCI interface connects to the PCI side of a Host/PCI bridge, which is usually the lowest numbered PCI bus in a system hierarchy. The primary interface consists of:

- 50 signal pins (mandatory; as defined in the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0)
- Four interrupt pins (optional)

Refer to the *PCI Local Bus Specification*, revision 2.1 for a complete description of individual pin functionality.

The primary PCI interface implements both an initiator (master) and a target (slave) PCI device. When a transaction is initiated on the secondary bus, that needs to be completed on the primary bus, the primary master state machine completes the transaction (write or read) as if it was the initiating device. The primary PCI interface — as a PCI target for transactions that need to complete on the secondary bus — accepts the transaction and forwards the request to the secondary side. As a target, the primary PCI interface uses positive decoding to claim the PCI transaction addressed within the bridge address space and then forwards the transaction onto the secondary master interface.

The primary PCI interface is responsible for all PCI command interpretation, address decoding and error handling. It also performs PCI configuration for:

- primary and secondary interfaces
- interrupt routing logic - described in section 8.3.3, Internal Peripheral Interrupt Routing (pg. 8-26)
- secondary PCI bus arbitration - described in CHAPTER 18, BUS ARBITRATION

Configuration space registers support these functions.

15.3.2 Secondary PCI Interface

The secondary PCI interface functions in almost the same manner as the primary interface. It consists of both a PCI master and a PCI slave device and implements the “second” PCI bus with a new set of PCI electrical loads for use by the system. The secondary PCI interface consists of the mandatory 50 pins and four optional interrupt pins. Note that S_RST# is an output instead of an input on the secondary side.

- As a slave (target), the secondary PCI interface claims PCI transactions that do not fit within the bridge’s secondary memory or I/O address space and forwards them up the bridge to the master on the primary side.
- As a master (initiator), the secondary PCI interface completes transactions initiated on the primary side.

The secondary PCI interface uses inverse decoding of the bridge address registers and only forwards addresses within the primary address space across the bridge.

The secondary PCI interface implements a separate address space for private PCI devices on the secondary bus, where it ignores and does not forward a range of primary addresses that the i960 core processor defines at configuration time. Support for private PCI devices is discussed in section 15.4, CONFIGURATION ACCESSES (pg. 15-6).

The secondary PCI interface optionally claims Dual Address Caches (DACs) on the secondary PCI interface.

As a special mode of operation, the secondary PCI interface performs positive address decoding based upon its own set of memory and I/O address registers. This mode of operation is enabled through the Secondary Decode Enable Register (SDER). Once this mode is enabled, the standard inverse decoding mechanism of the bridge address registers is disabled.

15.3.3 Buffers

The PCI-to-PCI bridge unit implements six buffers which hide the latency incurred in the arbitration and acquisition of a PCI target during read and write transactions. Three downstream buffers are for data flow from the primary interface to the secondary interface. Three upstream buffers are for data flow from the secondary interface to the primary interface. The upstream and downstream buffers are:

- Posted Write Buffer (64 bytes)
- Delayed Write Buffer (4 bytes)
- Delayed Read Buffer (64 bytes)

The bridge supports both Delayed and Posted transactions:

- In a Delayed transaction, the information required to complete the transaction is latched and the transaction is terminated with a Retry. The bridge then performs the transaction on behalf of the initiator. The initiator must repeat the original transaction that was terminated with a Retry to complete the transaction.
- In a Posted transaction, the transaction completes on the initiating bus before completing on the target bus.

Refer to section 15.6, BRIDGE OPERATION (pg. 15-22) for information about Delayed and Posted transactions; refer to section 15.7, BUFFERS (pg. 15-29) for more information about posting buffers.

15.3.4 Configuration Registers

The configuration registers hold all the necessary address decode, error condition and status information for both sides of the bridge. All PCI devices implement a separate configuration address space and configuration registers. The *PCI Local Bus Specification*, revision 2.1 requires a 256-byte configuration space, and the first 64 bytes must adhere to a predefined header format. The PCI-to-PCI bridge contains additional configuration registers. Refer to section 15.13, REGISTER DEFINITIONS (pg. 15-38) for full description.

The bridge configuration header formats' first 16 bytes, implement the common configuration registers which all PCI devices require. The read-only Header Type Register value defines the format for the remaining 48 bytes in the header and returns an 81H which defines a PCI-to-PCI bridge and multifunction PCI device.

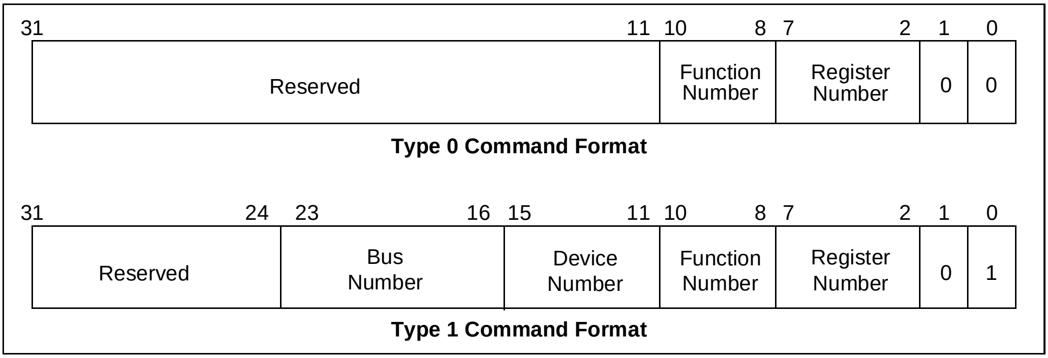
Devices on the primary bus can only access the PCI-to-PCI bridge configuration space with Type 0 configuration commands. Devices on the secondary PCI bus *cannot* access bridge configuration space with PCI configuration cycles.

15.4 CONFIGURATION ACCESSES

The i960 Rx I/O processor supports configuration access commands and types as defined in the *PCI Local Bus Specification*, revision 2.1. The various command types supported are:

- Type 0 configuration commands
- Type 1 configuration commands and Type 1 to Type 0 Conversions
- Type 1 to Type 1 Forwarding
- Type 1 to Special Cycle Conversion

Address encoding during a configuration command distinguishes the target of the command. Figure 15-3 shows the various address encodings associated with each PCI configuration command type. Type 0 and Type 1 configuration commands are distinguished by address bits AD1:0.



31 11	10 8	7 2	1	0
Reserved	Function Number	Register Number	0	0

Type 0 Command Format

31 24	23 16	15 11	10 8	7 2	1	0
Reserved	Bus Number	Device Number	Function Number	Register Number	0	1

Type 1 Command Format

Figure 15-3. PCI Configuration Access Formats

On the primary interface, the bridge ignores or accepts Type 0 configuration commands, depending on the value of the P_IDSEL input. A Type 1 configuration command on the primary interface may be ignored, forwarded downstream unaltered, converted to a Type 0 command on the secondary interface, or converted to a special cycle on the secondary interface.

On the secondary interface, Type 1 configuration write command may be ignored, forwarded upstream under certain conditions, or converted to a special cycle on the primary interface. The bridge cannot convert a Type 1 configuration command on the secondary side to a Type 0 on the primary side. The bridge ignores configuration reads and Type 0 configuration writes on the secondary interface.

The primary interface accepts configuration commands only when the Extended Bridge Command Register (EBCR) Configuration Cycle Disable bit is cleared. When the Configuration Cycle Disable bit is set, the primary PCI interface signals a Retry on all Type 1 and Type 0 configuration commands.

15.4.1 Private Configuration Commands (Type 0) on the Secondary Interface

The i960 Rx I/O processor's Address Translation Unit can generate Type 0 read and write configuration commands on the secondary interface which are not originally initiated as Type 1 configuration commands on the primary bus. Type 0 configuration commands must configure PCI devices on the secondary bus which reside in private PCI address space. All devices mapped into the private address space are not part of the standard secondary PCI address space and, therefore, the system host processor cannot configure these devices.

For Type 0 configuration commands on the secondary interface, S_AD31:11 select the target device's IDSEL input. In Type 1 to Type 0 conversions, P_AD15:11 are decoded to assert a unique address line from S_AD31:16 on the secondary interface as described in Section 15.4.2, Special Cycles. This leaves S_AD15:11 on the secondary interface open for a possibility of up to five address lines for IDSEL assertion of private PCI devices. These five address lines are reserved for private PCI devices on the secondary PCI bus.

When more than five unique address lines are required, the Secondary IDSEL Select Register (SISR) can be programmed to block secondary addresses (S_AD20:16 for 80960RP 33/5.0 and S_AD25:16 for 80960RP 33/3.3) from being used during Type 1 to Type 0 conversions from the primary interface. By setting the appropriate SISR register bits (bits 4:0 for 80960RP 33/5.0 and bits 9:0 for 80960RP 33/3.3), the associated address line can be forced to remain deasserted for the P_AD15:11 encodings of (00000_2 - 01000_2 for 80960RP 33/5.0 and 00000_2 - 010010_2 for 80960RP 33/3.3) and therefore is free to be used as an IDSEL select line for private secondary PCI devices.

Table 15-1 shows the possible configurations of S_AD31:11 for private Type 0 configuration commands on the secondary interface. For example, when SISR Bit 0 is set, S_AD16 is never asserted during a Type 1 to Type 0 conversion. Only the Address Translation Unit can assert this signal.

When the primary interface receives a Type 1 configuration command which specifies one of the S_AD address lines reserved for private PCI devices, the bridge performs the Type 1 to Type 0 conversion, but does not assert the reserved S_AD address line. The Type 0 configuration command is then ignored on the secondary PCI bus.

By using the SISR register and the five reserved address lines, a total of (10 for 80960RP 33/5.0 and 15 for 80960RP 33/3.3) IDSEL signals are available for private PCI devices.

Figure 15-4 shows an example of connecting S_AD lines to IDSEL inputs of PCI devices and private PCI devices.

15.4.2 Special Cycles

Except for conversion cycles, the bridge unit neither initiates nor accepts PCI special cycle commands on either interface.

Table 15-1. Private PCI Memory IDSEL Select Configurations

Secondary Addresses AD31:11	Secondary IDSEL Select Register Bits 9 - 0 for 80960RP 33/3.3	Use		
0000 0000 0000 0000 0000 1_2	XXXXX $XXXXX_2$	Reserved for private PCI devices		
0000 0000 0000 0000 0001 0_2	XXXXX $XXXXX_2$			
0000 0000 0000 0000 0010 0_2	XXXXX $XXXXX_2$			
0000 0000 0000 0000 0100 0_2	XXXXX $XXXXX_2$			
0000 0000 0000 0000 1000 0_2	XXXXX $XXXXX_2$			
0000 0000 0000 0001 0000 0_2	XXXXX $XXXX1_2$	Can be used for private PCI devices only when the associated bit in SISR is set	5.0 V	3.3 V
0000 0000 0000 0010 0000 0_2	XXXXX $XXX1X_2$			
0000 0000 0000 0100 0000 0_2	XXXXX $XX1XX_2$			
0000 0000 0000 1000 0000 0_2	XXXXX $X1XXX_2$			
0000 0000 0001 0000 0000 0_2	XXXXX $1XXXX_2$			
0000 0000 0010 0000 0000 0_2	XXXX1 $XXXXX_2$		3.3 V ONLY	
0000 0000 0100 0000 0000 0_2	XXX1X $XXXXX_2$			
0000 0000 1000 0000 0000 0_2	XX1XX $XXXXX_2$			
0000 0001 0000 0000 0000 0_2	X1XXX $XXXXX_2$			
0000 0010 0000 0000 0000 0_2	1XXXX $XXXXX_2$			

NOTES:

1. X = Don’t Care

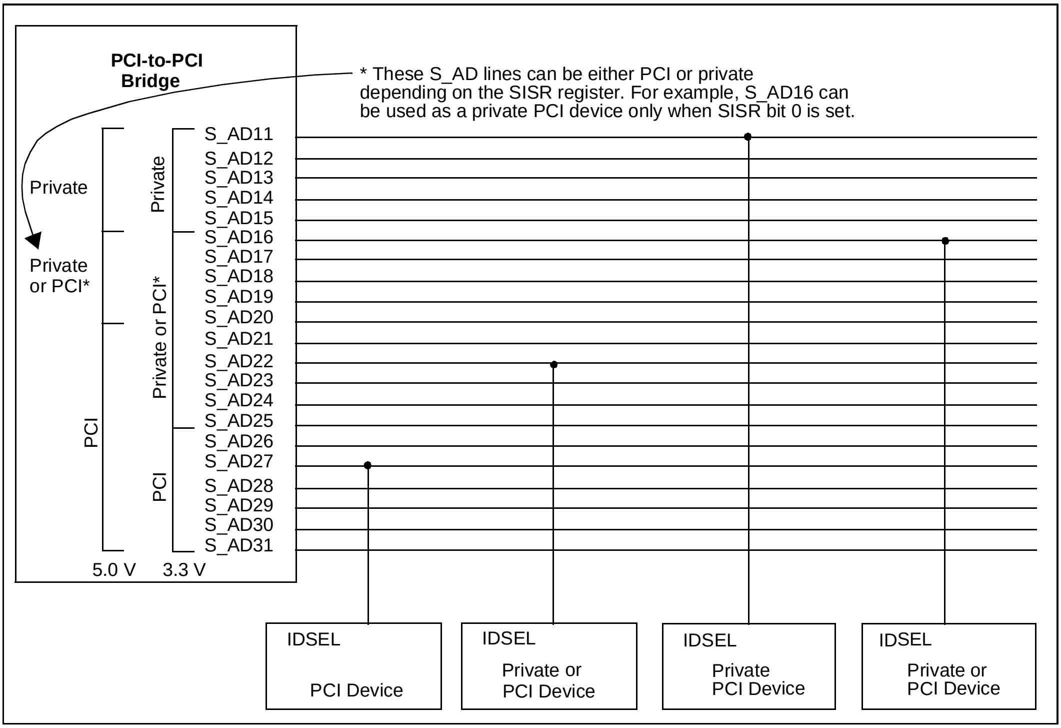


Figure 15-4. Secondary IDSEL Example

15.5 ADDRESS DECODING

The i960 Rx I/O processor provides three separate address ranges for determining which memory and I/O addresses are forwarded in either direction across the i960 Rx I/O processor’s bridge: two address ranges for memory transactions, one address range for I/O transactions. The bridge uses a base address register and limit register to implement an address range.

In addition to the memory and I/O space, the bridge unit supports an ISA compatibility mode and support for VGA graphics devices on the secondary interface.

The Secondary Decode Enable Register (SDER) can also modify standard bridge unit address decoding. It can enable the secondary bridge interface to use positive address decoding and disable the standard inverse address decoding that the PCI-to-PCI bridges use. This is known as the *Special Mode* of operation.

15.5.1 I/O Address Space

The PCI-to-PCI bridge unit implements one programmable address range for PCI I/O transactions. A continuous I/O address space is defined by the I/O Base Register (IOBR) and the I/O Limit Register (IOLR) in the bridge configuration space.

The bridge unit forwards from the primary to secondary interface an I/O transaction that has an address within the address range defined (inclusively) by the IOBR and the IOLR.

When an I/O read or write transaction is present on the secondary bus, the bridge unit forwards it to the primary interface when the address is outside the address range defined by IOBR and IOLR.

The i960 Rx I/O processor only supports 16-bit addresses for I/O transactions and therefore any I/O transaction with an address greater than 64 Kbytes is not forwarded over either interface.

The following registers and configuration bits control bridge response to I/O transactions:

- Primary Command Register (PCMDR)
  - VGA Palette Snoop Enable bit
  - Bus Master Enable bit
  - I/O Space Enable bit
- Bridge Control Register (BCR)
  - VGA Enable bit
  - ISA Enable bit
- Secondary Decode Enable Register (SDER)
  - Secondary Positive I/O Decode Enable bit

15.5.1.1 Disabling the I/O Address Range

The I/O address range can be disabled for primary to secondary transactions by using either the I/O Enable bit or the I/O Base and Limit Registers.

- When the I/O Limit Register (IOLR) is programmed to a value less than or equal to the I/O Base Register (IOBR), the i960 Rx I/O processor does not forward any transactions from the primary to the secondary. All I/O transactions from the secondary to the primary are forwarded upstream through the bridge.
- When positive I/O decoding is enabled on the secondary interface, setting the SIOLR less than or equal to the SIBLR or setting the Secondary Positive I/O Decode Enable bit disables the I/O address range in the same manner as the primary interface. Disabling the secondary I/O address range in this way (with positive I/O decoding from the secondary interface enabled) results in no I/O transactions being forwarded over either PCI interface.

- When the I/O Enable bit is cleared (0), neither the primary or the secondary PCI Interface claims I/O transactions.
- When the I/O Enable bit is set (1), neither the primary or the secondary PCI Interface behaves as described earlier.

15.5.1.2 ISA Mode

The Bridge Control Register (BCR) ISA Enable bit provides ISA-”awareness” for ISA I/O cards on subordinate PCI buses. ISA Mode only affects I/O addresses within the address range which the IOBR and IOLR registers define. When ISA Mode is enabled (ISA Enable bit is set), the bridge filters out and does not forward I/O transactions with addresses in the upper 768 bytes (300H) of each naturally aligned 1 Kbyte block. I/O transactions on the secondary bus inversely decode the ISA addresses and therefore forward I/O transactions with addresses in the upper 768 bytes of each naturally aligned 1 Kbyte block.

ISA Mode addressing is not supported in conjunction with positive decoding on the secondary interface. I/O address decoding with the ISA Enable bit set in the BCR and the Secondary Positive I/O Decode Enable bit set in the SDER is indeterminate.

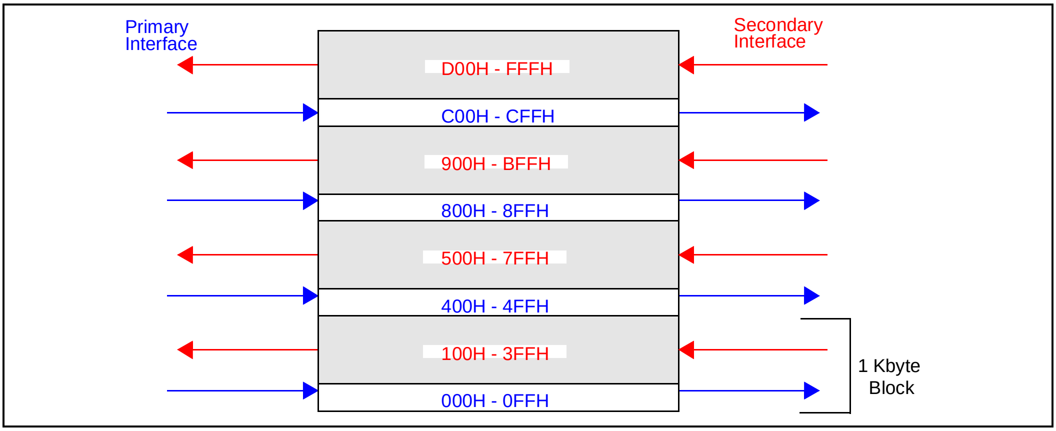


Figure 15-5. ISA Mode Address Decode

15.5.2 Memory Address Space

The mechanism for claiming and forwarding all PCI memory transactions complies with the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0.

The PCI-to-PCI bridge unit supports two separate address ranges for forwarding memory accesses downstream. The Memory Base Register (MBR) and the Memory Limit Register (MLR) define one address range and the Prefetchable Memory Base Register (PMBR) and the Prefetchable Memory Limit Register (PMLR) define the other address range. When the two register pairs overlap, one address range results that is the summation of both registers combined (Figure 15-6) with the prefetchable range having priority over bridge read transaction response.

The prefetchable address range maps memory address ranges of devices that are prefetchable. Both register pairs determine when the bridge forwards *Memory Read*, *Memory Read Line*, *Memory Read Multiple*, *Memory Write*, *and Memory Write and Invalidate* transactions across the bridge.

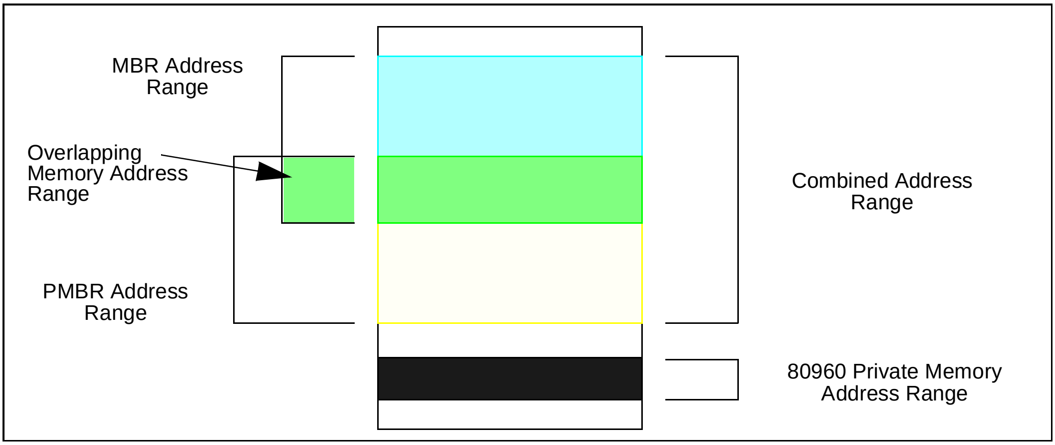

MBR Address Range
Overlapping Memory Address Range
PMBR Address Range
Combined Address Range
80960 Private Memory Address Range

Figure 15-6. Overlapping Memory Address Ranges

The bridge's response to memory transactions on either interface may be modified by the following register bits from the bridge configuration space:

- Primary Command Register (PCMD)
  - Bus Master Enable bit
  - Memory Enable bit
- Bridge Control Register (BCR)
  - VGA Enable bit
- Secondary Decode Enable Register (SDER)
  - Secondary Positive Memory Decode Enable bit

15.5.2.1 Disabling the Memory Address Range

The Memory address range can be disabled for primary to secondary transactions by using either the Memory Enable bit or the MBR-MLR and PMBR-PMLR register pairs.

- When the Memory Limit Register (MLR) is programmed to a value less than the Memory Base Register (MBR) and the Prefetchable Memory Limit Register (PMLR) is programmed to a value less than the Prefetchable Memory Base Register (PMBR), the i960 Rx I/O processor does not forward any transactions from the primary to the secondary. All Memory transactions from the secondary to the primary are forwarded upstream through the bridge.
- When positive Memory decoding is enabled on the secondary interface, setting the SMLR less than or equal to the SMBR or setting the Secondary Positive Memory Decode Enable bit disables the Memory address range in the same manner as the primary interface.
- When the Memory Enable bit is cleared (0), neither the primary or the secondary PCI Interface claims I/O transactions.
- When the Memory Enable bit is set (1), neither the primary or the secondary PCI Interface behaves as described in previous bullets.

15.5.3 VGA Address Support

To support a VGA device on a downstream bus from the i960 Rx I/O processor, the bridge can recognize and forward VGA addresses on the primary interface to the secondary interface. The i960 Rx I/O processor's bridge unit also supports the downstream graphics device need to snoop VGA palette accesses on the primary bus.

15.5.3.1 VGA Compatible Addressing

VGA addressing support allows the i960 Rx I/O processor to support VGA frame buffer addressing and VGA register addressing. When the VGA Enable bit is set in the Bridge Command Register, the i960 Rx I/O processor bridge unit positively decodes memory accesses to a VGA frame buffer and I/O accesses to VGA registers on a secondary bus. The following addresses are positively decoded on the primary interface when the VGA Enable bit is set:

- VGA memory accesses - 0A0000H - 0BFFFFH
- VGA I/O accesses - AD9:0 = 3B0H - 3BBH and 3C0H - 3DFH. These addresses are inclusive of ISA aliasing since AD15:10 are not decoded for VGA I/O accesses

When the VGA Enable bit is set, VGA compatible addressing is not dependent on the address ranges programmed into the MBR/MLR and PMBR/PMLR register pairs for memory or the IOBR/IOLR register pair. Regardless of the defined address ranges, addresses are forwarded from primary to secondary and blocked from secondary to primary. In addition, VGA compatible addressing is not dependent on the ISA enable bit or the VGA Palette Snoop bit.

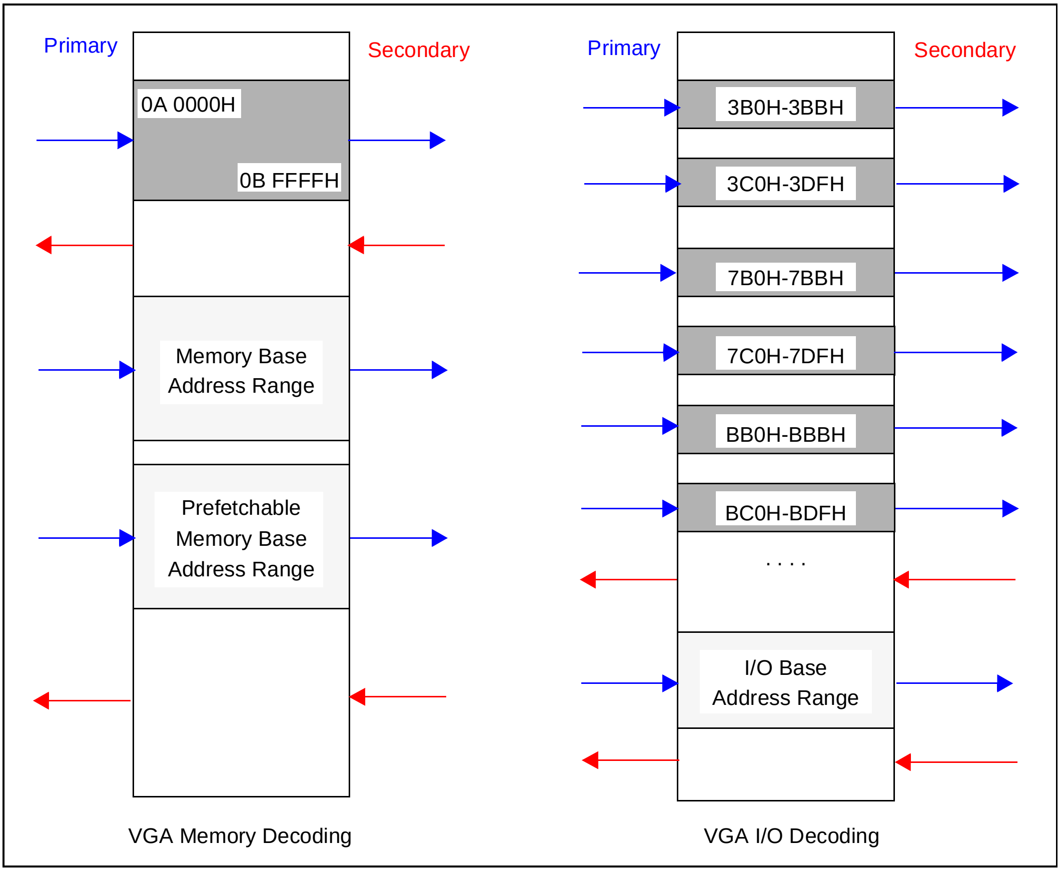


Figure 15-7. VGA Compatible Addressing

15.5.3.2 VGA Palette Snooping

VGA palette snooping mechanism is defined for:

- VGA compatible devices
- subtractive decoding bridges (expansion bridges, such as PCI to EISA)
- PCI-to-PCI bridge
- non-VGA compatible graphics devices

VGA palette addresses are defined as the following addresses:

- AD9:0 = 3C6H, 3C8H, and 3C9H (inclusive of ISA aliases since AD15:10 are not decoded).

The full VGA snooping mechanism for a PCI VGA devices is described on the *PCI Local Bus Specification*, revision 2.1 and the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0.

As shown in Table 15-2, the bridge unit supports three modes of palette accesses:

1. Ignore palette accesses - when there are no graphics devices downstream of the i960 Rx I/O processor's primary interface that need to snoop or respond to VGA palette access cycles.
2. Positively decode and forward palette writes - when graphics agents downstream of the bridge need to respond or snoop palette writes.
3. Positively decode and forward palette reads and writes - when VGA-compatible graphics agents downstream are being used.

The Bridge Control Register (BCR) VGA Enable bit and the Primary Command Register (PCMDR) VGA Snoop Enable bit control the bridge unit's response to palette accesses.

Table 15-2. VGA Palette Configurations

VGA Enable Bit	VGA Snoop Enable Bit	Bridge's Response to Palette Accesses
0	0	ignore all palette accesses
0	1	positively decode palette writes and ignore palette reads
1	X	positively decode palette writes and reads

Many restrictions apply to VGA Palette Snooping in a PCI system. Refer to the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0 for complete details.

15.5.4 64-Bit Address Decoding - Dual Address Cycles

The bridge unit supports the dual address cycle command for 64-bit addressing on the secondary interface only. The bridge unit uses medium decode timing (assert DEVSEL# on the second clock after the second address in a DAC cycle) for claiming dual address cycle commands.

The bridge unit decodes and forwards all dual address cycles from the secondary to the primary interface regardless of the address ranges defined in the MBR/MLR and PMBR/PMLR register pairs.

The PCMDR's Master Enable bit controls the use of DAC cycles on the secondary interface. This bit must be set for the primary interface to master PCI transactions. Claiming of DAC cycles can be disabled by modifying the Extended Bridge Control Register's DAC cycle disable bit.

15.5.5 Private Address Space

The bridge supports private address space by not claiming and forwarding private Memory addresses on the secondary PCI bus. Private addresses are only supported on the secondary PCI bus. The bridge does not claim the following transactions:

- Inbound transactions from private secondary PCI devices to the secondary ATU
- Outbound transactions from the secondary ATU or DMA channel 2 to private secondary PCI devices
- Transactions between two private devices

For transactions between private devices, application code must use the Secondary Memory Base Register and Secondary Memory Limit Register to define a private address range. Also, the Private Memory Space Enable bit in the Secondary Decode Enable Register must be set. See section 15.13.34, Secondary Decode Enable Register - SDER (pg. 15-73).

15.5.6 Address Decode Summary

Tables in this section summarize the address decode options. Each pair of tables is divided into one Memory transaction table and one I/O transaction table. The tables list the various control bits and the potential address ranges.

The response for the address, noted in each table entry, is determined by the control bits and the address range into which the address falls, and may be one of:

- Forward the transaction across the Bridge
- Ignore the transaction and do not forward across the Bridge
- Range is not valid; the response is determined by another address range (denoted as “-”)

It is assumed that the Memory and I/O Base and Limit address ranges are only valid when the Limit is greater than or equal to the Base.

Table 15-3 is a summary of the Memory address decoding rules for Primary to Secondary Memory transactions. Note that the VGA Memory range is independent of the MBR/MLR and PMBR/PMLR ranges and is only valid when the VGA Enable bit is set and the Memory Enable bit is set.

Table 15-3. Primary to Secondary Memory Address Decoding Summary

Memory Enable bit	VGA Enable bit	Primary to Secondary			
		In MBR/MLR range	In PMBR/PMLR range	In VGA Memory range	Outside all valid ranges
0	0	Ignore	Ignore	-	Ignore
0	1	Ignore	Ignore	-	Ignore
1	0	Forward	Forward	-	Ignore
1	1	Forward	Forward	Forward	Ignore

NOTES:

1. Usage of “Forward”, “Ignore”, and “dash” are defined as follows:
   - Forward the transaction across the Bridge
   - Ignore the transaction and do not forward across the Bridge
   - Range is not valid; the response is determined by another address range (denoted as “-”)

Table 15-4 is a summary of the I/O address decoding rules for Primary to Secondary I/O transactions.

The I/O Enable bit must be set to forward any I/O transactions. To be in the ISA range or the VGA Palette Snoop range, the address must also fall in the IOBR/IOLR range. The ISA range covers the complete IOBR/IOLR range. The VGA I/O range is independent of the IOBR/IOLR range.

When the Secondary Positive Memory Decode Enable bit is clear, then the secondary interface must ignore the SMBR/SMLR range. When either the Secondary Positive Memory Decode Enable bit or the Secondary Positive I/O Decode Enable bit is set, then inverse decoding is disabled (except when the Private Address Space Enable bit is set; see next paragraph).

The SDER Private Address Space Enable bit can disable forwarding of the SMBR/SMLR range and override the Secondary Positive Memory Decode Enable and Secondary Positive I/O Decode Enable bits. When the Private Address Space Enable bit is set, the Secondary Positive Memory Decode Enable and Secondary Positive I/O Decode Enable bits are ignored.

Table 15-4. Primary to Secondary I/O Address Decoding Summary

I/O Enable bit	ISA Mode bit	VGA Enable bit	VGA Palette Snoop bit	Primary to Secondary[1]					
				In IOBR/IOLR range	**In VGA I/O range**	**In VGA Palette range**	**In ISA range** (Lower 256 bytes)	**In ISA range** (Upper 768 bytes)	**Outside all valid ranges**
0	X	X	X	Ignore					
1	0	0	0	Forward	-	-	-	-	Ignore
1	0	0	1	Forward	-	Forward Writes; Ignore Reads	-	-	Ignore
1	0	1	0	Forward	Forward	Forward	-	-	Ignore
1	0	1	1	Forward	Forward	Forward	-	-	Ignore
1	1	0	0	-	-	-	Forward	Ignore	Ignore
1	1	0	1	-	-	Forward Writes; Ignore Reads	Forward	Ignore	Ignore
1	1	1	0	-	Forward	Forward	Forward	Ignore	Ignore
1	1	1	1	-	Forward	Forward	Forward	Ignore	Ignore

NOTES:

1. Usage of “Forward”, “Ignore”, and “dash” are defined as follows:
   - Forward the transaction across the Bridge
   - Ignore the transaction and do not forward across the Bridge
   - Range is not valid; the response is determined by another address range (denoted as “-”)

Table 15-3 is a summary of the address decoding rules for Secondary to Primary Memory transactions.

Table 15-5. Secondary to Primary Memory Address Decoding Summary

Master Enable bit	Private Address Space Enable bit	Secondary Positive I/O Decode Enable bit	Secondary Positive Memory Decode Enable bit	Mem Enable bit	VGA Enable bit	Secondary to Primary					
						In MBR /MLR range	In PMBR/ PMLR range	In VGA Memory range	In ATU Inbound Address range	In SMBR/ SMLR range	Outside all valid ranges
0	X	X	X	X	X	Ignore					
1	0	0	0	0	0	-	-	-	Ignore	-	Forward
1	0	0	0	0	1	-	-	-	Ignore	-	Forward
1	0	0	0	1	0	Ignore	Ignore	-	Ignore	-	Forward
1	0	0	0	1	1	Ignore	Ignore	Ignore	Ignore	-	Forward
1	0	X	1	0	0	-	-	-	Ignore	Forward	Ignore
1	0	X	1	0	1	-	-	-	Ignore	Forward	Ignore
1	0	X	1	1	0	-	-	-	Ignore	Forward	Ignore
1	0	X	1	1	1	-	-	Ignore	Ignore	Forward	Ignore
1	0	1	0	0	0	-	-	-	Ignore	-	Ignore
1	0	1	0	0	1	-	-	-	Ignore	-	Ignore
1	0	1	0	1	0	-	-	-	Ignore	-	Ignore
1	0	1	0	1	1	-	-	-	Ignore	-	Ignore
1	1	X	X	0	0	-	-	-	Ignore	Ignore	Forward
1	1	X	X	0	1	-	-	-	Ignore	Ignore	Forward
1	1	X	X	1	0	Ignore	Ignore	-	Ignore	Ignore	Forward
1	1	X	X	1	1	Ignore	Ignore	Ignore	Ignore	Ignore	Forward

1. Usage of “Forward”, “Ignore”, and “dash” are defined as follows:
   - Forward the transaction across the Bridge
   - Ignore the transaction and do not forward across the Bridge
   - Range is not valid; the response is determined by another address range (denoted as “-”)

Table 15-4 summarizes the I/O address decoding rules for Secondary to Primary I/O transactions. The ISA Enable only pertains to the IOBR/IOLR range and never the SIOBR/SIOLR range. Also, when the IOBR/IOLR range overlaps with the SIOBR/SIOLR range and a conflict exists, the bridge ignores the transaction.

The bridge ignores all VGA palette accesses on the secondary PCI bus when the VGA Palette Snoop Enable bit is set.

Table 15-6. Secondary to Primary I/O Address Decoding Summary (Sheet 1 of 2)

Master Enable	Secondary Positive Memory Decode	Secondary Positive I/O Decode	I/O Enable bit	ISA Mode bit	VGA Enable	Secondary to Primary					
						In IOBR/ IOLR range	In ISA range (Lower 256 bytes)	In ISA range (Upper 768 bytes)	In VGA I/O range	In SIOBR/ SIOLR range	Outside all valid ranges
0	X	X	X	X	X	Ignore					
1	0	0	0	0	0	-	-	-	-	-	Forward
1	0	0	0	0	1	-	-	-	-	-	Forward
1	0	0	0	1	0	-	-	-	-	-	Forward
1	0	0	0	1	1	-	-	-	-	-	Forward
1	0	0	1	0	0	Ignore	-	-	-	-	Forward
1	0	0	1	0	1	Ignore	-	-	Ignore	-	Forward
1	0	0	1	1	0	-	Ignore	Forward	-	-	Forward
1	0	0	1	1	1	-	Ignore	Forward	Ignore	-	Forward
1	X	1	0	0	0	-	-	-	-	Forward	Ignore
1	X	1	0	0	1	-	-	-	-	Forward	Ignore
1	X	1	0	1	0	-	-	-	-	Forward	Ignore
1	X	1	0	1	1	-	-	-	-	Forward	Ignore
1	X	1	1	0	0	Ignore	-	-	-	Forward	Ignore
1	X	1	1	0	1	Ignore	-	-	Ignore	Forward	Ignore

NOTES:

1. Usage of “Forward”, “Ignore”, and “dash” are defined as follows:
   - Forward the transaction across the Bridge
   - Ignore the transaction and do not forward across the Bridge
   - Range is not valid; the response is determined by another address range (denoted as “-”)

Table 15-6. Secondary to Primary I/O Address Decoding Summary (Sheet 2 of 2)

Master Enable	Secondary Positive Memory Decode	Secondary Positive I/O Decode	I/O Enable bit	ISA Mode bit	VGA Enable	Secondary to Primary					
						In IOBR/ IOLR range	In ISA range (Lower 256 bytes)	In ISA range (Upper 768 bytes)	In VGA I/O range	In SIOBR/ SIOLR range	Outside all valid ranges
1	X	1	1	1	0	-	Ignore	Forward	-	Forward	Ignore
1	X	1	1	1	1	-	Ignore	Forward	Ignore	Forward	Ignore
1	1	0	0	0	0	-	-	-	-	-	Ignore
1	1	0	0	0	1	-	-	-	-	-	Ignore
1	1	0	0	1	0	-	-	-	-	-	Ignore
1	1	0	0	1	1	-	-	-	-	-	Ignore
1	1	0	1	0	0	Ignore	-	-	-	-	Ignore
1	1	0	1	0	1	Ignore	-	-	Ignore	-	Ignore
1	1	0	1	1	0	-	Ignore	Forward	-	-	Ignore
1	1	0	1	1	1	-	Ignore	Forward	Ignore	-	Ignore

NOTES:

1. Usage of “Forward”, “Ignore”, and “dash” are defined as follows:
   - Forward the transaction across the Bridge
   - Ignore the transaction and do not forward across the Bridge
   - Range is not valid; the response is determined by another address range (denoted as “-”)

15.6 BRIDGE OPERATION

The i960 Rx I/O processor's bridge unit can forward various memory, I/O and configuration commands from one PCI interface to the other PCI interface. Table 15-7 defines the PCI commands which the PCI-to-PCI bridge unit, and its two PCI interfaces, support. To prevent deadlock, the bridge gives priority to the primary interface for simultaneous transactions on both interfaces.

Table 15-7. PCI Commands

C/BE#	PCI Command	Initiator: Primary Bus Target: Secondary Bus	Initiator: Secondary Bus Target: Primary Bus
0000_2	Interrupt Acknowledge	Ignore	Ignore
0001_2	Special Cycle	Ignore	Ignore
0010_2	I/O Read	Forward	Forward
0011_2	I/O Write	Forward	Forward
0100_2	Reserved	Ignore	Ignore
0101_2	Reserved	Ignore	Ignore
0110_2	Memory Read	Forward	Forward
0111_2	Memory Write	Forward	Forward
1000_2	Reserved	Ignore	Ignore
1001_2	Reserved	Ignore	Ignore
1010_2	Configuration Read	Forward	Ignore
1011_2	Configuration Write	Forward	Forward
1100_2	Memory Read Multiple	Forward	Forward
1101_2	Dual Address Cycle	Ignore	Forward
1110_2	Memory Read Line	Forward	Forward
1111_2	Memory Write and Invalidate	Forward	Forward

15.6.1 PCI Interfaces

The bridge unit consists of a primary and secondary PCI interface between the two PCI buses which the i960 Rx I/O processor interconnects:

- The primary interface connects to the lower numbered PCI bus
- The secondary interface connects to the higher numbered PCI bus

When the primary bus initiates a transaction and the bridge claims the transaction, the primary interface becomes the PCI target device; the secondary interface becomes the initiating device for the true PCI target on the secondary bus. The primary is the initiating bus; the secondary is the target bus. The sequence is reversed for transactions initiated on the secondary bus.

15.6.2 Claiming a PCI Transaction

The PCI-to-PCI bridge unit complies with *PCI Local Bus Specification*, revision 2.1 and the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0. Refer to the *PCI Local Bus Specification*, revision 2.1 for full details on transaction claiming.

15.6.2.1 Master Latency Timers

Two Master latency timers (MLT) are programmed to limit one master's ownership of a PCI bus in the presence of other bus masters. These timers are controlled via the Primary Latency Timer Register (PLTR) and the Secondary Latency Timer Register (SLTR).

These registers define the maximum count and granularity of the primary and secondary latency timers. Each register is 8 bits wide; this allows each interface to "own" its PCI bus for a time slice of up to 248 PCI clocks. The PLTR and SLTR lower three bits (02 through 00) are hardwired to 000_2 which forces a minimum timer granularity of 8 PCI clocks. The register's upper five bits are programmable; each PCI interface can be independently programmed to a value between 11111000_2 and 00000000_2 which results in a timer count of 0 to 248.

15.6.2.2 Delayed Transactions

A delayed transaction is a mechanism for processing PCI transactions in high initial latency PCI targets, such as PCI-to-PCI bridges, thereby improving overall bus efficiency. All PCI transactions (except for Special Cycle) can complete as a Delayed transaction.

The bridge processes all transactions as Delayed transactions, except for the transactions Memory Write and Memory Write and Invalidate. These two transactions are processed as either Delayed transactions or as Posted transactions, based on the Extended Bridge Control Register Posting Disable bit:

- When the Posting Disable bit is clear, Memory Write and Memory Write and Invalidate transactions are processed as Posted transactions.
- When the Posting Disable bit is set, Memory Write and Memory Write and Invalidate transactions are processed as Delayed transactions.

In a Delayed transaction, the bridge latches the address, command, and byte enable information required to complete the transaction and the initiator is signaled a Retry. For writes, the latched information includes the data to be written. The bridge performs the request on the target bus on behalf of the initiator. For reads, the returning data and the target response is stored in the bridge posting buffers. For writes, only the target response is recorded. The original initiator must then repeat the original request to complete the transaction.

15.6.2.3 Posted Transactions

In a Posted transaction, the bridge stores the data in a posting buffer and signals a termination other than Retry. Once the bridge acquires the target bus, it completes the request.

Table 15-8 summarizes the differences between Delayed and Posted transactions.

Table 15-8. Delayed Transactions vs. Posted Transactions

Delayed Transaction	Posted Transaction
For all PCI commands (except Special Cycle)	For Memory Write, Memory Write and Invalidate commands only
Requires repeated request	Does not require repeated request
Completes on target bus before initiating bus	Completes on initiating bus before target bus
Less efficient for writes	More efficient for writes

15.6.3 PCI Read Transactions

The i960 Rx I/O processor supports memory read and I/O read transactions from both sides of the bridge. The bridge implements Delayed read transactions to meet initial transaction latency requirements. Delayed Requests are accepted as new requests when all the following conditions apply:

- New Read requests are accepted 4 cycles after the read buffer becomes available. The read buffer is available if there are no delayed read requests, or delayed read completion currently in the queue for a given direction.
- Reads are completed 4 cycles after (4 or 8 DWORDS) are in the read buffer and a match on the address command Command/Byte Enable lines.

The amount of data that the bridge reads on the target bus and stores in the buffer depends on PCI command type and whether the memory address space is prefetchable or not.

Whether the memory address space is prefetchable or not depends on the command type and whether the transaction is being forwarded downstream or upstream. See Table 15-9 for a summary. For downstream transactions, the address range (MBR-MBLR or PMBR-PMLR) used to claim the address determines whether the memory is prefetchable or not. See Section 15.5.2, Memory Address Space. For DAC commands on the secondary PCI bus, the bridge treats the memory as prefetchable.

For Memory Read commands on the secondary PCI bus, the bridge treats the memory as prefetchable or non-prefetchable depending on the Extended Bridge Control Register Upstream Prefetchable Memory Enable bit: when set, upstream memory is prefetchable. This bit controls the type of memory for Memory Read commands only.

For Memory Read Line and Memory Read Multiple commands on the secondary PCI bus, the bridge treats the memory as prefetchable.

Table 15-9. Prefetchable Memory Summary

PCI Command	Downstream	Upstream
Memory Read	Depends on Address Range	Depends on Upstream Prefetchable Memory Enable bit in EBCR
Memory Read Line	Prefetchable	Prefetchable
Memory Read Multiple	Prefetchable	Prefetchable
DAC	N/A	Prefetchable

When the memory space is prefetchable, the bridge reads and stores data up to an 8 DWORD boundary for a Memory Read or Memory Read Line command and attempts to fill the complete buffer for a Memory Read Multiple command. For Memory Read commands, the bridge acquires the target bus to stream data through the read buffers. See Table 15-10 for the amount of data read for Memory commands. Streaming is more likely to occur with proper use of the Memory Read Multiple command.

The bridge does not prefetch past a 4 Kbyte address boundary. This prevents a prefetchable access from crossing the boundary from a prefetchable range into a non-prefetchable range. When the 4 Kbyte address boundary is reached, the bridge signals a Disconnect on the target bus.

I/O Read commands and Configuration Read commands are limited to one PCI data phase. The bridge reads and stores up to 1 DWORD for I/O Read commands and Configuration Read commands. The bridge signals a Disconnect to the initiator when it requests more than one DWORD for an I/O Read or Configuration Read command.

Table 15-10. Memory Read Prefetch Size

Command	Prefetchable Memory Address Space	Non-Prefetchable Memory Address Space
Memory Read	Up to 8 DWORDs	1 DWORD
Memory Read Line	Up to 8 DWORDs	Up to 8 DWORDs
Memory Read Multiple	Fill the buffer	Fill the buffer

The bridge terminates the Delayed Completion transaction with:

- Completion termination when the transaction on the target bus terminated normally.
- Master-abort termination or Completion termination when the transaction on the target bus terminated with Master-abort. See section 15.12.1, Address Parity Errors (pg. 15-34).

- Target-abort termination when the transaction on the target bus terminated with Target-abort.
- Disconnect termination when the transaction on the target bus terminated with Disconnect.

Any additional data words the bridge reads from the target — which are not ultimately requested by the initiator — are discarded. The bridge does *not* follow the termination rules above when it reads more data than is requested. The bridge terminates with Completion termination when the initiator requests less data words than the bridge read from the target.

When the expected number of data transfers are not received from the target for a Memory Read Line command or a Memory Read Multiple command, the bridge performs the same number of data transfers to the initiator during the Delayed Read Completion transaction as it receives from the target. For example, if the Cacheline Size Register is programmed to 8 DWORDs and the target only delivers 4 DWORDs, the bridge delivers 4 DWORDs to the initiator during the Delayed Read Completion transaction.

When the read transaction is a Memory Read Multiple command to a prefetchable memory address space, the bridge attempts to acquire the target bus during the Delayed Completion transaction to stream data across the bridge. When successful in acquiring the target bus, the bridge transfers additional data words to the initiator. This continues until the target initiates termination (Disconnect or Target-abort), the bridge initiates termination on the target bus (Time-out) or the initiator terminates the transaction.

The bridge only supports the linear incrementing burst mode for Memory commands and signals a Disconnect to the initiator after the transfer of the first DWORD when the burst mode is *not* linear incrementing.

NOTE: For Reads, the initiator must repeat the transaction with exactly the same address, byte enables, and command. Otherwise, the bridge treats the transaction as a new request which results in a livelock condition.

15.6.4 PCI Write Transactions

The i960 Rx I/O processor supports memory write and I/O write transactions from both sides of the bridge unit. Memory write transactions are claimed when they are within the MBR/MLR or PMBR/PMLR address pairs on the primary bus and outside the register pairs on the secondary bus. I/O Write transactions are claimed when they are within the IOBR/IOLR write transactions on the primary bus and outside the address pair on the secondary bus.

The bridge supports both posted and delayed write transactions for memory transactions. I/O write and configuration write transactions are always delayed transactions. The Extended Bridge Control Register (EBCR) Posting Disable bit determines the transaction:

- When cleared, posting can occur from either bridge interface.
- When set, all write transactions are processed as delayed transactions.

15.6.4.1 Delayed Write Transactions

Delayed write transactions are limited to one data cycle and are used for:

- I/O Writes
- Configuration Writes
- All memory writes when the Extended Bridge Control Register (EBCR) Posting Disable bit is set; the bridge limits all write commands to one PCI data phase.

When the target bus is obtained, the bridge propagates the write data from the initiating bus to the target bus. The bridge keeps the request information in a posting buffer. The request information is the address, command, byte enables, parity (when enabled), and data.

Once the write data is successfully transferred to the target, the bridge can now accept the repeated write command from the original initiator. The bridge must match the address, command, byte enables, parity (when parity is enabled), and data to signal a termination other than Retry to the initiator. It uses:

- Completion termination when the transaction on the target bus terminates normally.
- Master-abort termination when the transaction on the target bus terminated with Master-abort.
- Target-abort termination when the transaction on the target bus terminated with Target-abort.
- Disconnect termination when the transaction on the target bus terminated with Disconnect.

NOTE: For Writes, the initiator must repeat the transaction with exactly the same address, byte enables, command, parity, and data. Otherwise, the bridge treats the transaction as a new request which results in a livelock condition.

When a parity error occurs during the Delayed Write Request transaction, the bridge causes a deadlock unless the repeated transaction has the same parity error.

15.6.4.2 Posted Write Transactions

In a posted write transaction, the bridge accepts the write data and asserts TRDY# to the initiating bus before the data is transferred to target interface for writing to the target bus.

For posted write transactions, the initiator can transfer a maximum of 16 DWORDs to the bridge unit while the bridge attempts to obtain access to the target bus. When the bridge has not yet acquired the target bus and received a TRDY# from the target, and the bridge posted write buffer becomes full, the bridge signals a Disconnect to the initiator on the last word that fills the posting buffer.

When the bridge unit can transfer data before the buffers fill, the initiator continues to transfer write data. This continues until:

- The target initiates termination (Disconnect or Target-abort)
- The bridge initiates termination on the target bus (Time-out)
- The initiator completes the required number of write transfers

When the target bus transaction terminates while the initiator is still transferring data, the buffers fill and a Disconnect would occur in the same situation as when the initiator started the original transaction. In addition, the posted write buffers must signal a Disconnect when the initiating interface signals a TRDY# on the initiating interface and is unable to signal the next TRDY# within 8 PCI clocks on the initiating interface. The Disconnect is signaled after the seventh clock after the last TRDY#, unless the bridge determines it cannot meet the eight-clock TRDY# to TRDY# specification. In this case, the Disconnect is signaled earlier.

The bridge unit supports simultaneous write posting in both directions across the bridge. Emptying the posting buffers has priority over initiating a new transaction on a target bus.

15.6.4.3 Memory Write Command

The Extended Bridge Control Register’s Posting Disable bit determines whether Memory write transactions are either posted or delayed transactions. Delayed Memory Write commands only transfer one PCI data phase. This means FRAME# is only asserted for one clock on the target interface and that the initiating interface signals a target Disconnect after the first data transfer.

15.6.4.4 Memory Write and Invalidate Command

The Memory Write and Invalidate (MWI) command is identical to the Memory Write command except it guarantees a minimum transfer of at least one cacheline, as defined by the Cacheline Size Register (CLSR).

When the bridge accepts a MWI command which is terminated with a Disconnect by the target before the entire cacheline transfers, the bridge uses a Memory Write command to complete the transaction. When the transaction is still in progress, the bridge is free to disconnect the initiator with a target Disconnect on the initiating bus. The bridge unit takes no further action; no error is reported.

The bridge unit also converts a MWI command to a Memory Write command when:

- the CLSR is programmed to a value of zero and the cacheline size is unknown
- the Cacheline Size Register is programmed to a value greater than 16 DWORDs (size of posting buffer)

Refer to the *PCI Local Bus Specification*, revision 2.1 for the full details of a Memory Write and Invalidate command.

When posting is disabled, the bridge prohibits the MWI command to appear on the target bus. The bridge converts the MWI to a Memory Write and only allows one PCI data phase on the target bus.

15.6.4.5 I/O Write Command

All I/O Write transactions are processed as Delayed transactions. The i960 Rx I/O processor is restricted to 16-bit addressing for I/O transactions.

- On the primary bus, the bridge claims any transaction inside the 16-bit address range which the I/O Base and I/O Limit registers define.
- On the secondary bus, the bridge claims any transaction outside the address range.

15.6.4.6 Write Boundaries

The PCI-to-PCI bridge unit imposes a naturally-aligned 4096 byte write boundary for posted write transactions only. When the bridge unit detects a write boundary, the initiating interface signals a Disconnect to the initiator and completes delivery of the write data still retained in the internal posting buffers to the target interface.

15.6.4.7 Fast Back to Back Transactions

The i960 Rx I/O processor bridge unit does not generate fast back to back transactions. The fast back to back enable bits in the BCR and PCMDR, when set, are ignored.

15.7 BUFFERS

The PCI to PCI bridge unit has six buffers that are used for both Delayed transactions and Posted transactions. The downstream and upstream posting buffers are:

- Posted Write Buffer
- Delayed Write Buffer
- Delayed Read Buffer

The upstream and downstream buffers can be used simultaneously. Hence, transactions to opposite interfaces can occur on both PCI interfaces at the same time. Extra PCI clocks are not required to move data from one entry in the buffer to the next.

NOTE: The i960 Rx I/O processor has no special logic to flush the buffers before PCI interrupts are delivered.

15.7.1 Buffer Organization

Each Posted Write Buffer can hold one Posted Write transaction with up to 64 bytes of data. The buffer is organized as 16 entries of 4 bytes each (16 DWORDs).

Each Delayed Write Buffer can hold one Delayed Write Request or one Delayed Write Completion transaction with up to 4 bytes of data.

Each Delayed Read Buffer can hold one Delayed Read Request or one Delayed Read Completion with up to 64 bytes of data.

Associated with each buffer is an address register and a set of tag bits and valid bits.

15.7.2 Buffer Operation

The buffers help the bridge achieve the full PCI bandwidth and “hide” the latency of acquiring two PCI buses for every transaction crossing the bridge. The EBCR Posting Disable bit must be clear for Posted Write Buffers to post write transactions.

As a default reset state, the buffers are marked invalid. Any subsequent PCI reset event forces all buffers to be cleared by being marked invalid.

15.7.3 Transaction Ordering Rules

Since the bridge can process multiple transactions, it maintains proper ordering to avoid deadlock conditions and improve throughput. Given the buffer organization, the bridge supports the following transactions in each direction:

- Up to one Posted Memory Write, and/or
- Up to one Delayed Read Request or one Delayed Read Completion (but not both at one time), and/or
- Up to one Delayed Write Request or one Delayed Write Completion (but not both at one time)

The bridge completes request transactions in the order received. The mechanism complies with the PCI transaction ordering rules in the *PCI Local Bus Specification*, revision 2.1.

15.8 BRIDGE DATA FLOW

The PCI-to-PCI Bridge of the i960 Rx I/O processor supports transactions from both PCI buses. This section identifies variations of upstream and downstream transactions:

- *downstream* transactions are initiated on the primary PCI bus and targeted at an agent on the secondary PCI bus

- *upstream* transactions are initiated on the secondary PCI bus and targeted at an agent on the primary PCI bus

The following sections describe:

- Downstream Delayed Read transaction
- Downstream Delayed Write transaction
- Downstream Posted Write transaction

15.8.1 Downstream Delayed Read Transaction

A downstream delayed read transaction is initiated by a PCI master on the primary PCI bus and is targeted at a PCI agent on the secondary PCI bus. All downstream read transactions are processed as delayed read transactions. The bridge's PCI interface claims the read transaction and forwards the read request through to the secondary PCI bus, then returns the read data to the primary PCI bus. The Delayed Read Buffer (DRB) contains the downstream PCI address and the read data.

15.8.2 Downstream Delayed Write Transaction

A downstream delayed write transaction is initiated by an agent on the primary PCI bus and is targeted at a PCI agent on the secondary PCI bus. All write transactions are processed as delayed write transactions when posting is disabled. The downstream write address and write data are propagated from the primary PCI bus to the secondary PCI bus through the Delayed Write Buffer (DWB). The Bridge claims any PCI write transaction when the PCI address is within the address window defined by a Base/Limit register pair.

15.8.3 Downstream Posted Write Transaction

A downstream posted write transaction is initiated by a PCI master on the primary PCI bus and is targeted at a PCI agent on the secondary PCI bus. Write transactions are processed as posted transactions when posting is enabled. The downstream write address and write data are propagated from the primary PCI bus to the secondary PCI bus through the Posted Write Buffer (PWB). The Bridge claims any PCI write transaction when the PCI address is within the address window defined by a Base/Limit register pair.

15.8.4 Definitions

The Bridge data flow uses the following terms and abbreviations:

- **PWB** - Posted Write Buffer. Each buffer holds up to 16 DWORDs (64 bytes) and the associated write address. The downstream Posted Write Buffer holds data moving downstream and the upstream Post Write Buffer holds data moving upstream.

- **DRB** - Delayed Read Buffer. Each buffer holds up to 16 DWORDs (64 bytes) and the associated read address.
- **DWB** - Delayed Write Buffer. Each buffer holds up to 1 DWORD (4 bytes) and the associated write address.
- **Inside Window** - This refers to whether or not an address on the PCI bus is within the range window defined by a Base and Limit register pair. A PCI address inside the window is claimed by the bridge PCI interface.
- **PCI Cycle Complete** - Refers to the normal termination of a PCI transaction on the PCI bus.

15.9 EXCLUSIVE ACCESS

The bridge unit supports the PCI exclusive access mechanism (initiated on the primary interface) using the PCI LOCK# signal. The bridge lock mechanism works with 4 states:

- *Free*: No lock transactions from a primary master have been detected on the primary bus. Transactions flow upstream and downstream freely through the bridge.
- *To Be Locked*: Entered into when a downstream read request (DRR) is accepted on the primary interface with P_LOCK# active. All new requests on both sides of the bridge are retried while the bridge attempts to complete all pending transactions (unlocked) enqueued prior to the locked read request.
- *Locked*: Entered into once all unlocked transactions have cleared the bridge queues and the secondary bridge interface has established a lock (with S_LOCK#) with a secondary slave. All upstream traffic is retried during this state and only downstream transactions (read or write) from the lock master on the primary interface are accepted.
- *To be Unlocked*: Entered into when the lock master on the primary interface has released P_LOCK# (or during error states). The bridge does not accept any new requests on either interface while waiting to complete any pending locked downstream PMW transactions. After completing the locked transaction, the bridge releases S_LOCK# and moves to the *Free* state.

See Figure 15-8 for a state diagram of the bridge lock mechanism.

The bridge establishes itself as a locked target during a DRR when P_LOCK# is de-asserted in the address phase and asserted in the clock cycle after the address phase. The PCI master must release P_LOCK# since it was signaled a Retry by the bridge. The bridge attempts to establish a master-target lock with the target on the secondary bus (according to the states defined above).

When locked, the bridge signals a Retry to any other master (besides the lock master) on the primary bus and does not accept transactions moving upstream from secondary masters.

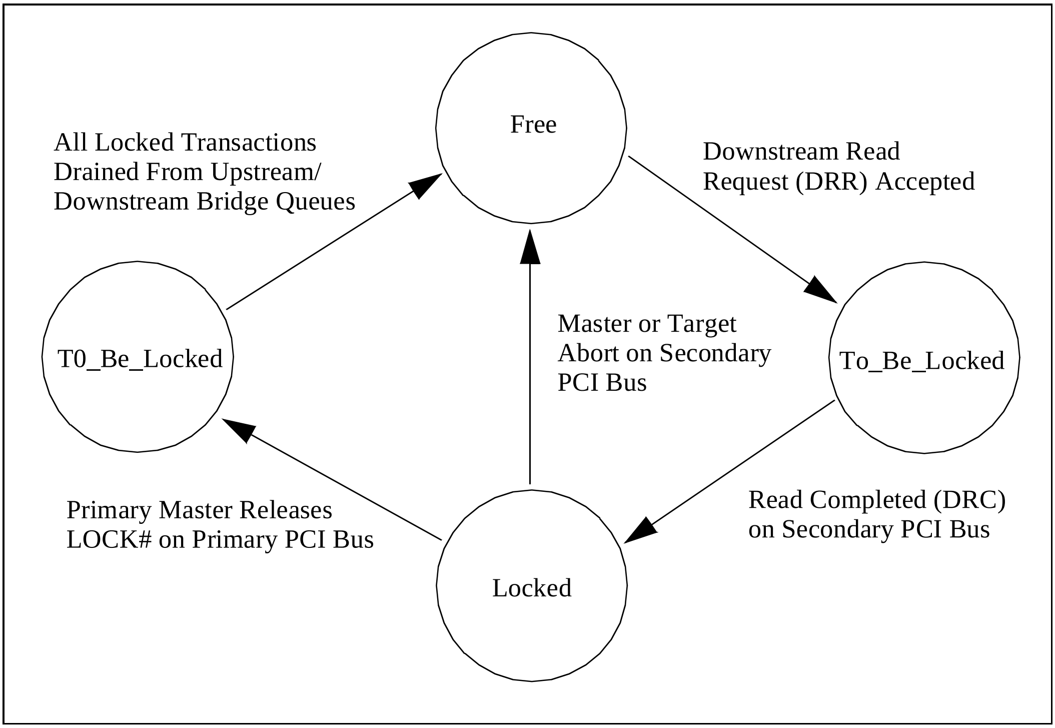

Free
All Locked Transactions Drained From Upstream/ Downstream Bridge Queues
Downstream Read Request (DRR) Accepted
T0_Be_Locked
Master or Target Abort on Secondary PCI Bus
To_Be_Locked
Primary Master Releases LOCK# on Primary PCI Bus
Read Completed (DRC) on Secondary PCI Bus
Locked

Figure 15-8. Bridge Lock Mechanism

15.10 SYNCHRONIZATION EVENTS

Bridge synchronization is enforced for the following events:

- I/O read
- I/O write
- configuration read
- configuration write
- memory read following a memory write
- read of bridge configuration registers through the 80960 memory-mapped registers
- write of bridge configuration register through the 80960 memory-mapped registers

When either event occurs, the bridge ensures that the posting buffers in both directions are clear before the event continues. For PCI transactions, the bridge signals a Retry on the initiating interface while the bridge clears the posting buffers by completing the current transaction within the buffers. When the posting buffers are clear, transactions can be accepted on the bridge unit's initiating interface.

When a bridge configuration register is being read from or written to, the bridge does not accept new transactions. The bridge unit signals a Retry to any PCI master attempting to initiate a transaction on either PCI bus.

15.11 PCI TRANSACTION TERMINATION

As a PCI master (initiator), a device can terminate a transaction when it is complete or when an error condition occurs. As a slave (target), a PCI device can only terminate when an error condition occurs. While transaction termination can be initiated by either a master or a target, ultimately it is up to the master to bring a PCI transaction to an orderly conclusion. All PCI transaction termination mechanisms are consistent with the *PCI Local Bus Specification*, revision 2.1.

15.12 ERROR CONDITIONS

The PCI-to-PCI bridge unit implements parity generation and parity error detection on both the primary and secondary PCI interfaces and passes that information to the primary interface. This enables the parity error recovery mechanisms outlined in the *PCI Local Bus Specification*, revision 2.1 without special considerations for a bridge. The following sections detail the bridge unit response to parity errors on both interfaces.

15.12.1 Address Parity Errors

When the bridge — as a target on the initiating interface — detects a parity error before claiming a cycle, the bridge does not claim the cycle (not assert DEVSEL#) and terminates the transaction with the Master-abort mechanism. When the bridge detects a parity error during a transaction the primary and secondary interfaces handle the error in different manners.

15.12.1.1 Address Parity Errors on Primary Interface

When an address parity error occurs on the primary interface, the i960 Rx I/O processor:

- asserts P_SERR# on the primary interface (when the PCMDR P_SERR# Enable bit and the Parity Error Response Enable bit are enabled)
- sets the Signaled SERR# bit in the PSR (when the PCMDR P_SERR# Enable bit is enabled)
- sets the Detected Parity Error bit in the PSR

15.12.1.2 Address Parity Errors on Secondary Interface

When an address parity error occurs on the secondary interface, the i960 Rx I/O processor:

- asserts P_SERR# on the primary interface (when enabled by the P_SERR# Enable bit, Parity Error Response Enable bit in the PCMDR and the S_SERR# enable bit in the BCR)
- sets the Signaled SERR# bit in the PSR (when P_SERR# Enable bit in the PCMDR is enabled)
- sets the Detected Parity Error bit in the SSR

15.12.2 Data Parity Errors

When the bridge unit detects a data parity error, the bad data and bad parity is passed to the opposite interface. This enables the parity error recovery mechanisms outlined in the *PCI Local Bus Specification*, revision 2.1 without special consideration for the bridge in the datapath.

15.12.2.1 Read Data Parity

The PCI-to-PCI bridge unit passes bad data and bad parity to the initiating interface during a delayed read transaction that crosses the bridge. The following situations are possible during read transactions that contain data parity errors.

Parity error during delayed read request (DRR) on the initiating bus: This is not possible since there is no data transferred when the initiating interface of the bridge latches the address/command to initiate a delayed read transaction.

Parity error during delayed read completion (DRC) on the target bus: The target interface of the bridge detects a data parity error when data is being driven from the PCI slave. If parity is enabled (by the PCMDR Parity Checking Enable bit for the primary interface or the BCR Parity Response Enable for the secondary interface) the interface drives PERR# in response to the bad parity. The bad data and parity are delivered to the initiating interface during the DRC on the initiating bus. The target interface sets the Detected Parity Error and the Data Parity Detected bits in the PSR if the target interface is the primary, or the SSR if the target interface is the secondary. If parity is disabled for the target interface, the bridge does not assert PERR#, but continues to set the Detected Parity Error bit in the corresponding status register.

Parity error during the delayed completion (DRC) on the initiating bus: This can occur in response to a parity error on the target bus or independent of the target bus transaction. If the PCI master asserts PERR#, the bridge sets the Detected Parity error bit in the corresponding interface status register (PSR for primary or SSR for secondary).

15.12.2.2 Delayed Write Data Parity

Table 15-11 summarizes the bridge response to a data parity error on Delayed Write transactions.

Table 15-11. Delayed Write Parity Error Summary

Transaction	Parity Error Detected Error Reporting Disabled	Parity Error Detected Error Reporting Enabled
Initial Delayed Write Request on the Initiating Bus	Signal Retry to initiator	Assert TRDY#, Report Parity Error; Assert PERR#, Discard write data.
Repeated Delayed Write Request on the Initiating Bus	Signal Retry to initiator	Assert TRDY#, Report Parity Error; Assert PERR#, Discard write data.
Target Bus Data Transfer	Store the write data; Do not report Parity Error	Discard write data.
Delayed Write Completion on the Initiating Bus	Assert TRDY#, Complete the transaction	Not Valid.

The bridge only reports parity errors on Delayed Write transactions on the initiating bus. It sets the Detected Parity Error bit in the PSR when the primary bus is the initiating bus or the SSR when the secondary bus is the initiating bus.

When PERR# is detected on the target bus, the bridge unit sets the Data Parity Detected bit (when enabled) in the status register corresponding to the target interface (PSR or SSR).

15.12.2.3 Posted Write Data Parity

When a data parity error is detected by the bridge's initiating interface during a posted write transaction that crosses the bridge, it asserts PERR# on the initiating bus and retains the bad data and parity in its posting buffers. The bridge sets the Detected Parity Error bit (when enabled). When the write data is transferred on the target bus, the transaction target asserts PERR# on the target bus. When the target asserts PERR#, the bridge sets the Data Parity Detected bit and the Detected Parity Error bit in the status register corresponding to the target interface (PSR or SSR).

When the bridge does not detect a data parity error on the initiating bus and a data parity error is detected on the target bus, the transaction master cannot determine when a data parity error occurred. However, when the SERR# Enable bit is set, the bridge propagates the error upstream to the primary interface, asserting P_SERR# on the primary bus. The bridge also sets the PSR Signaled System Error bit. The Detected Parity Error in the status register corresponding to the initiating bus (PSR or SSR) is not set since the initiating interface did not detect the parity error.

15.12.3 Master-abort

A Master-abort occurs when no target responds with a DEVSEL# within five clocks after the assertion of FRAME#**.** The i960 Rx I/O processor bridge unit has two mechanisms for handling Master-aborts.

When a read transaction crosses the bridge in the Master Abort Mode bit clear mode and the target interface signals a Master-abort, the bridge terminates normally (with TRDY#) on the initiating interface.

When a write transaction crosses the bridge and the target interface signals a Master-abort, the bridge completes the transaction normally on the initiating interface and discards the write data on the target interface. In both cases, the bridge:

- Sets the Primary Status Register (PSR) Received Master Abort bit when the Master-abort occurred on the primary interface
- Sets the Secondary Status Register (SSR) Received Master Abort bit when the Master-abort occurred on the secondary interface

When the Master Abort Mode bit is set, the bridge signals a Master-abort to the initiator of a delayed read or write transaction when that transaction causes a Master-abort on the target bus. The bridge sets the corresponding Received Master Abort bit as in the previous case. When the transaction that caused the Master-abort on the target interface was a posted write transaction, the bridge asserts P_SERR# on the primary interface (when enabled). The bridge terminates the posted write transaction on the initiating interface with a Disconnect (assuming the write is still occurring) on the target interface.

A Master-abort is not signaled during a Special Cycle transaction from either interface.

15.12.4 Target-abort

A Target-abort occurs when STOP# is asserted and DEVSEL# is deasserted.

For all transactions crossing the bridge (except posted writes; see next paragraph) the bridge signals a Target-abort to the initiator on the initiating bus when one is received by the bridge on the target bus. The bridge sets the Target Abort (target) bit in the target bus’s status register (PSR or SSR) and the Target Abort (master) bit in the initiating bus’s status register.

When the bridge detects a Target-abort during a posted write transaction on the target bus and the write is still in progress on the initiating bus, the bridge signals a Target-abort to the initiator on the initiating bus. The bridge sets the Target Abort (target) bit in the target bus’s status register (PSR or SSR) and the Target Abort (master) bit in the initiating bus’s status register.

In instances where the posted write transaction on the initiating interface is complete, the bridge asserts P_SERR# (when enabled) on the primary interface, which indicates a system error. The bridge also sets the Target Abort (target) bit in the target bus’s status register (PSR or SSR) and the Target Abort (master) bit in the initiating bus’s status register.

15.12.5 SERR# Assertion

When S_SERR# is asserted on the secondary interface, the bridge asserts P_SERR# on the primary interface (when enabled by SERR# Enable bit) to propagate the error upstream. The bridge also sets the Received System Error bit in the SSR.

15.13 REGISTER DEFINITIONS

The configuration space consists of 8, 16, 24, and 32-bit registers arranged in a predefined format. Configuration registers are accessed through Type 0 Configuration Read and Write commands on the primary side and through i960 Rx I/O processor local bus commands.

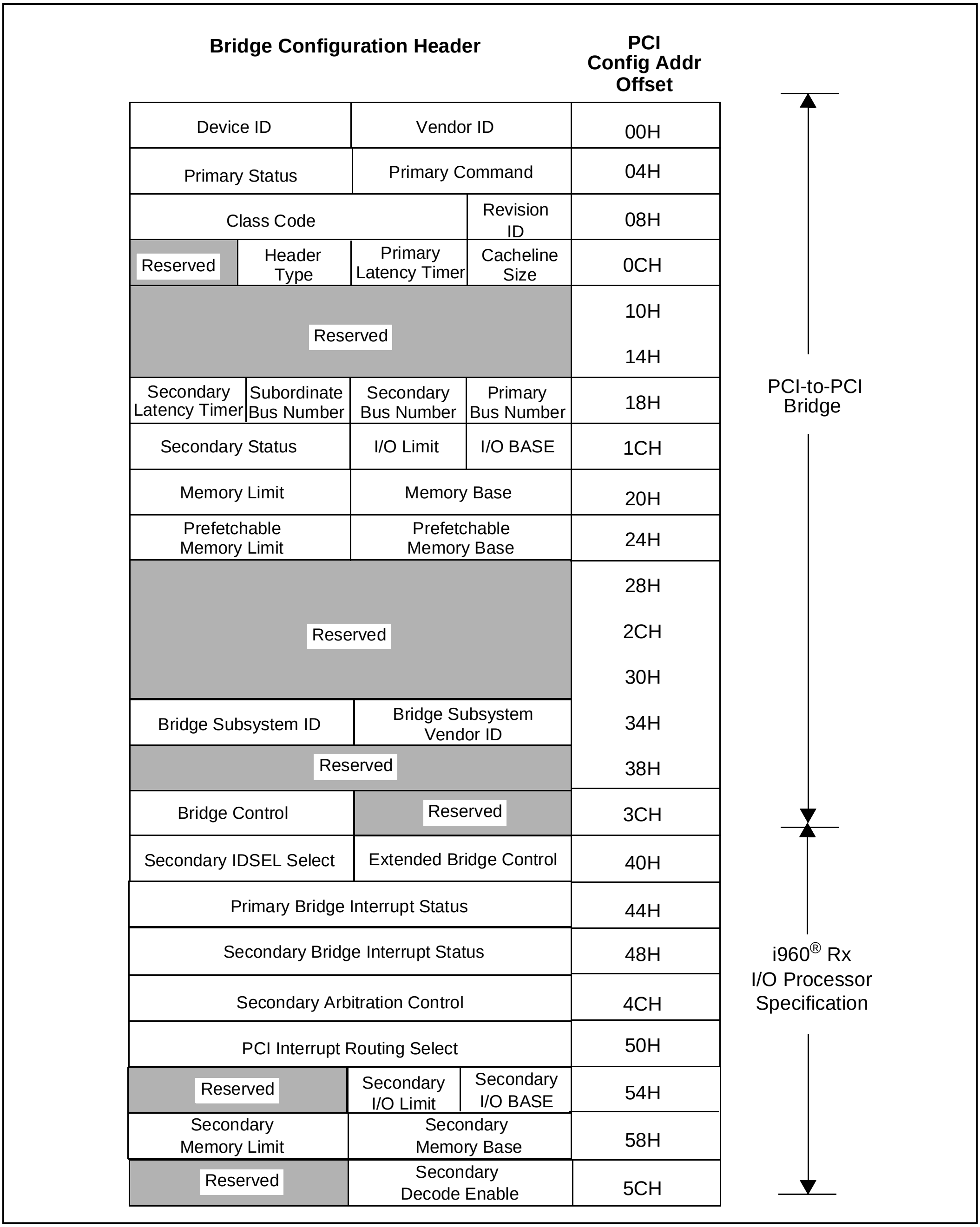
The i960 Rx I/O processor is a multifunction PCI device. The PCI-to-PCI bridge unit is function zero; the Address Translation Unit is function one. Both functions have separate configuration space. Refer to CHAPTER 16, ADDRESS TRANSLATION UNIT, for definition of the ATU (function one) configuration register.

Figure 15-9 describes the entire bridge PCI configuration space. As stated, a Type 0 configuration command on the primary side with an active IDSEL or a memory-mapped i960 core processor access must read or write these registers. The format for registers with offsets up to 3FH are defined in the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0. Registers with offsets greater than 3FH are implementation-specific to the i960 Rx I/O processor.

Unless otherwise noted, all registers adhere to the definitions found in the *PCI Local Bus Specification*, revision 2.1 and the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0.

The i960 core processor also has access to the bridge configuration space. As a result, certain configuration registers can be initialized before PCI configuration begins. The i960 core processor reads and writes the bridge configuration space as memory-mapped registers. Read and Write access capabilities from the PCI Type 0 configuration command and the i960 core processor are detailed in the figure for each register. Refer to the individual register legends for definition of access rights.

P_RST# signal assertion on the primary side affects the state of **most** registers contained within the bridge configuration space. Unless otherwise noted, all bits and registers return to their stated default state value upon primary reset. The secondary S_RST# output’s reset state does not affect the bridge register state, unless otherwise noted.



Bridge Configuration Header				PCI Config Addr Offset	
Device ID		Vendor ID		00H	PCI-to-PCI Bridge
Primary Status		Primary Command		04H	
Class Code			Revision ID	08H	
Reserved	Header Type	Primary Latency Timer	Cacheline Size	0CH	
Reserved				10H	
				14H	
Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number	18H	
Secondary Status		I/O Limit	I/O BASE	1CH	
Memory Limit		Memory Base		20H	
Prefetchable Memory Limit		Prefetchable Memory Base		24H	
Reserved				28H	
				2CH	
				30H	
Bridge Subsystem ID		Bridge Subsystem Vendor ID		34H	
Reserved				38H	
Bridge Control		Reserved		3CH	
Secondary IDSEL Select		Extended Bridge Control		40H	i960® Rx I/O Processor Specification
Primary Bridge Interrupt Status				44H	
Secondary Bridge Interrupt Status				48H	
Secondary Arbitration Control				4CH	
PCI Interrupt Routing Select				50H	
Reserved		Secondary I/O Limit	Secondary I/O BASE	54H	
Secondary Memory Limit		Secondary Memory Base		58H	
Reserved		Secondary Decode Enable		5CH	

Figure 15-9. Bridge Configuration Register Space

Table 15-12. PCI to PCI Bridge Unit Register Summary (Sheet 1 of 2)

Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
15.13.1	Vendor ID Register - VIDR	15-41	16	0000 1000H	00H
15.13.2	Device ID Register - DIDR	15-42	16	0000 1002H	02H
15.13.3	Primary Command Register - PCMDR	15-42	16	0000 1004H	04H
15.13.4	Primary Status Register - PSR	15-44	16	0000 1006H	06H
15.13.5	Revision ID Register - RIDR	15-46	8	0000 1008H	08H
15.13.6	Class Code Register - CCR	15-46	24	0000 1009H	09H
15.13.7	Cacheline Size Register - CLSR	15-47	8	0000 100CH	0CH
15.13.8	Primary Latency Timer Register - PLTR	15-48	8	0000 100DH	0DH
15.13.9	Header Type Register - HTR	15-49	8	0000 100EH	0EH
	Reserved		x	0000 100FH through 0000 1017H	0FH through 17H
15.13.10	Primary Bus Number Register - PBNR	15-50	8	0000 1018H	18H
15.13.11	Secondary Bus Number Register - SBNR	15-50	8	0000 1019H	19H
15.13.12	Subordinate Bus Number Register - SubBNR	15-51	8	0000 101AH	1AH
15.13.13	Secondary Latency Timer Register - SLTR	15-52	8	0000 101BH	1BH
15.13.14	I/O Base Register - IOBR	15-52	8	0000 101CH	1CH
15.13.15	I/O Limit Register - IOLR	15-53	8	0000 101DH	1DH
15.13.16	Secondary Status Register - SSR	15-54	16	0000 101EH	1EH
15.13.17	Memory Base Register - MBR	15-56	16	0000 1020H	20H
15.13.18	Memory Limit Register - MLR	15-57	16	0000 1022H	22H
15.13.19	Prefetchable Memory Base Register - PMBR	15-58	16	0000 1024H	24H
15.13.20	Prefetchable Memory Limit Register - PMLR	15-59	16	0000 1026H	26H
	Reserved		x	0000 1028H through 0000 1033H	28H through 33H
15.13.21	Bridge Subsystem Vendor ID Register - BSVIR	15-60	16	0000 1034H	34H
15.13.22	Bridge Subsystem ID Register - BSIR	15-60	16	0000 1036H	36H
	Reserved		x	0000 1038H through 0000 103DH	38H through 3DH
15.13.23	Bridge Control Register - BCR	15-61	16	0000 103EH	3EH
15.13.24	Extended Bridge Control Register - EBCR	15-64	16	0000 1040H	40H
15.13.25	Secondary IDSEL Select Register - SISR	15-66	16	0000 1042H	42H
15.13.26	Primary Bridge Interrupt Status Register - PBISR	15-68	32	0000 1044H	44H
15.13.27	Secondary Bridge Interrupt Status Register - SBISR	15-69	32	0000 1048H	48H

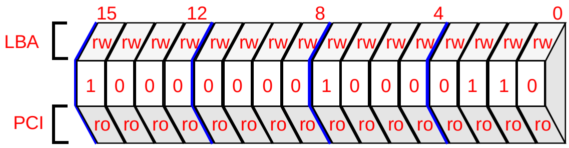
Table 15-12. PCI to PCI Bridge Unit Register Summary (Sheet 2 of 2)

Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
15.13.28	Secondary Arbitration Control Register - SACR	15-69	32	0000 104CH	4CH
15.13.29	PCI Interrupt Routing Select Register - PIRSR	15-70	32	0000 1050H	50H
15.13.30	Secondary I/O Base Register - SIOBR	15-70	8	0000 1054H	54H
15.13.31	Secondary I/O Limit Register - SIOLR	15-71	8	0000 1055H	55H
	Reserved		x	0000 1056H through 0000 1057H	56H through 57H
15.13.32	Secondary Memory Base Register - SMBR	15-72	16	0000 1058H	58H
15.13.33	Secondary Memory Limit Register - SMLR	15-72	16	0000 105AH	5AH
15.13.34	Secondary Decode Enable Register - SDER	15-73	16	0000 105CH	5CH
	Reserved		x	0000 105EH through 0000 105FH	5EH through 5FH

15.13.1 Vendor ID Register - VIDR

Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1.

Table 15-13. Vendor ID Register - VIDR


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

LBA: 1000H
PCI: 00H

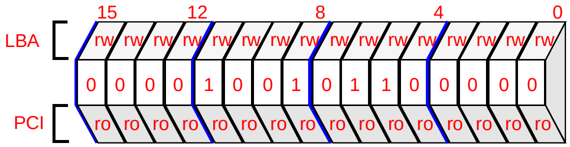
Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 local bus address
PCI = PCI Configuration Address Offset

Bit	Default	Description
15:0	8086H	Vendor ID - This is a 16-bit value assigned to Intel. This register, combined with the DIDR, uniquely identify the PCI device.

15.13.2 Device ID Register - DIDR

Device ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1.

Table 15-14. Device ID Register - DIDR


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

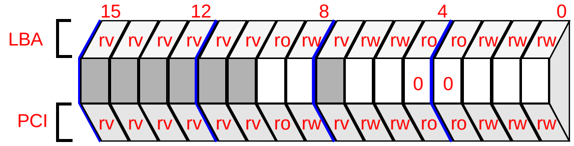
LBA: 1002H
PCI: 02H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15:00	0960H	Device ID - This is a 16-bit value assigned to the i960 Rx I/O processor.

15.13.3 Primary Command Register - PCMDR

Primary Command Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1 and, in most cases, affects PCI-to-PCI bridge's primary interface behavior.

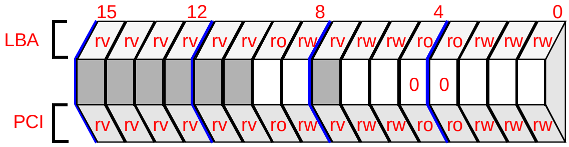
Table 15-15. Primary Command Register - PCMDR (Sheet 1 of 2)


15
12
8
4
0
LBA
rv rv rv rv rv rv ro rw rv rw rw ro ro rw rw rw
0 0
PCI
rv rv rv rv rv rv ro rw rv rw rw ro ro rw rw rw

LBA: 1004H
PCI: 04H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15:10	00H	Reserved
09	0_2	Fast Back to Back Enable - This bit is ignored.
08	0_2	P_SERR# Enable - When cleared, the i960 Rx I/O processor cannot assert P_SERR# on its primary interface. 0= Prevent P_SERR# assertion 1= Allow P_SERR# assertion

Table 15-15. Primary Command Register - PCMDR (Sheet 2 of 2)


15
12
8
4
0
LBA
rv rv rv rv rv rv ro rw rv rw rw ro ro rw rw rw
0
0
PCI
rv rv rv rv rv rv ro rw rv rw rw ro ro rw rw rw

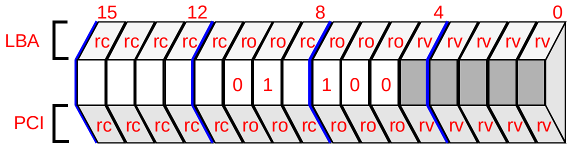
LBA: 1004H
PCI: 04H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
07	0_2	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field. Reserved. Always cleared.
06	0_2	Parity Checking Enable - When set, the bridge must take normal action when a parity error is detected. 0=disable 1=enable
05	0_2	VGA Palette Snoop Enable - When set, I/O writes with address bits AD9:0 = 3C6H, 3C8H, and 3C9H are positively decoded on the primary interface and forwarded to the secondary interface and must not be forwarded from secondary to primary. VGA palette snooping is independent of the address range programmed into the I/O base and limit registers and the ISA Enable bit. It is dependent on the I/O Enable bit in the PCMD. Refer to VGA section for further details.
04	0_2	Memory Write and Invalidate Enable - Not applicable. A PCI-to-PCI bridge does not initiate MWI commands, only forwards them on behalf of a PCI master. The PCI master has control to determine which type of write command to use. Setting or learning this bit does not effect the functionality of MWI commands across the bridge.
03	0_2	Special Cycle Enable - Read only; always cleared. 0=SC transactions are not supported.
02	0_2	Bus Master Enable - Must be set for the primary interface to act as a PCI bus master on behalf of the secondary interface. This bit does not affect the bridge’s ability to forward or convert configuration commands. When cleared, all secondary transactions must be disabled.
01	0_2	Memory Space Enable - Controls the bridges response to both memory-mapped I/O and prefetchable memory accesses. When cleared, the bridge does not respond to any memory access on the primary side.
00	0_2	I/O Space Enable - Controls the bridges response to I/O transactions on the primary side. When cleared, the bridge does not respond to any I/O transaction on the primary side.

15.13.4 Primary Status Register - PSR

Primary Status Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1 but only apply to the primary interface. The Read/Clear bits can only be set by the internal hardware and are cleared by either writing a 1_2 to the register or by a reset condition.

Table 15-16. Primary Status Register - PSR (Sheet 1 of 2)

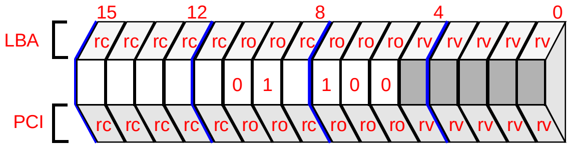

15
12
8
4
0
LBA
rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv
0 1 1 0 0
PCI
rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv

LBA: 1006H
PCI: 06H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15	0_2	Detected Parity Error 0 = No Detected Parity Error 1 = The bridge sets this bit when a parity error is detected on the primary bus even when parity handling is disabled.
14	0_2	Signaled System Error - 0 = No System Error 1 = The bridge sets this bit when P_SERR# is asserted on the primary bus.
13	0_2	Master Abort - 0 = No Master Abort 1 = The bridge sets this bit when a transaction initiated by the bridge on the primary bus (except Special Cycles) terminates with a Master-abort.
12	0_2	Target Abort (master) - 0 = No target abort 1 = The bridge sets this bit when a transaction initiated by the primary interface terminates with a Target-abort.
11	0_2	Target Abort (target) - 0 = No target abort 1 = The bridge sets this bit when the bridge, acting as a target, terminates the transaction on the primary bus with a Target-abort.
10:09	01_2	DEVSEL# Timing - 01 = Primary interface uses Medium Decode timing.

Table 15-16. Primary Status Register - PSR (Sheet 2 of 2)


15
12
8
4
0
LBA
rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv
0 1 1 0 0
PCI
rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv

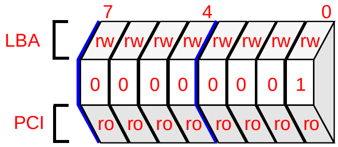
LBA: 1006H
PCI: 06H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

08	0_2	Data Parity Error Detected - 0 = No data parity error 1 = The bridge sets this bit when: • the bridge asserted P_PERR# (or saw asserted) on the primary bus • the bridge was transaction master when error occurred • the Parity Checking Enable bit is set
07	1_2	Fast Back-to-Back Capable - Read only; always set. 1 = The primary interface can accept Fast Back-to-Back transactions as a target.
06	0_2	User-Definable Features (UDF) Support - Read only; always cleared. 0= User Definable Features are not supported
05	0_2	66 MHz Capable - Read only; always cleared. 0= 33 MHz operation is supported
04:00	00H	Reserved

15.13.5 Revision ID Register - RIDR

Revision ID Register bits adhere to definitions in the *PCI Local Bus Specification*, revision 2.1.

Table 15-17. Revision ID Register - RIDR



LBA: 1008H
PCI: 08H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

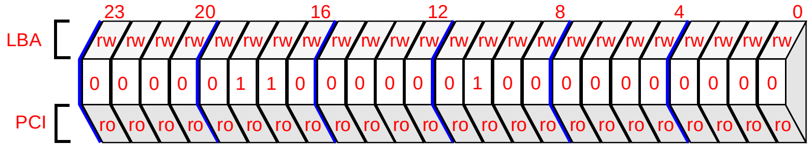
Bit	**Default**	**Description**
07:00	01H	Revision ID - Identifies the i960 Rx I/O processor's revision number.[1]

[1] These are the currently specified devices in stepping, refer to the data sheet and specification update for the latest valid values.

15.13.6 Class Code Register - CCR

The Class Code Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. Auto configuration software uses this register to determine the function type present in the PCI device.

Table 15-18. Class Code Register - CCR


23
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

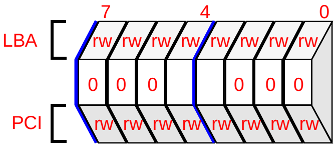
LBA: 1009H
PCI: 09H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
23:16	06H	Base Class - Bridge Device
15:08	04H	Sub Class - PCI-to-PCI Bridge Device
07:00	00H	Programming Interface - Consistent with *PCI-to-PCI Bridge Architecture Specification*, revision 1.0.

15.13.7 Cacheline Size Register - CLSR

Cacheline Size Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1 and apply to both sides of the bridge. It is programmed with the system cacheline size in DWORDs (32-bit quantities). The Cacheline Size is restricted to either 8 or 16 DWORDs. When a value other than 8 or 16 is written to the Cacheline Size Register, the Bridge returns a “0”.

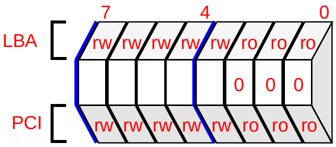
Table 15-19. Cacheline Size Register - CLSR

<table>
<tr><td colspan="3"> </td></tr>
<tr><td colspan="2"><b>LBA:</b> 100CH
<b>PCI:</b> 0CH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>07:05</td><td>000_2</td><td>Must be clear (0).</td></tr>
<tr><td>04:03</td><td>00_2</td><td>Cacheline size in DWORDs. Cacheline size is restricted to either 8 or 16 DWORDs.
00 = 0 DWORDS
01 = 8 DWORDS
10 = 16 DWORDS
11 = treated as 0 DWORDS</td></tr>
<tr><td>02:00</td><td>000_2</td><td>Cacheline Size Granularity - Giving a programmable granularity of DWORDs for the Cacheline Size. Must be clear (0).</td></tr>
</table>

15.13.8 Primary Latency Timer Register - PLTR

Primary Latency Timer Register (PLTR) bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1 and apply to the primary side only. When the timer counts down to zero, the bridge must terminate the transaction when GNT# is deasserted.

Table 15-20. Primary Latency Timer Register - PLTR



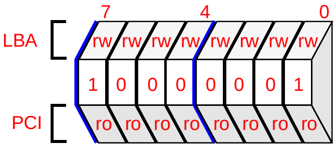
LBA: 100DH
PCI: 0DH
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
07:03	00H	Programmable Latency Timer - Varies the latency timer for the primary interface from 0 to 248 clocks.
02:00	000_2	Latency Timer Granularity - Read only; giving a programmable granularity of 8 clocks for the Latency Timer.

15.13.9 Header Type Register - HTR

Header Type Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. This register indicates the layout of bytes 10H to 3FH in the bridge configuration space. The MSB indicates whether or not the device is multi-function; defined as a 1 for multi-function in the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0.

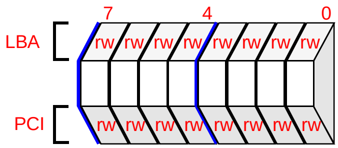
Table 15-21. Header Type Register - HTR

<table>
<tr><td colspan="3">
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 100EH
<b>PCI:</b> 0EH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset</td></tr>
<tr><th>Bit</th><th>Default</th><th>Description</th></tr>
<tr><td>07</td><td>1_2</td><td>Single-Function/Multi-Function Device - This bit identifies the i960 Rx I/O processor as a multi-function PCI device.
0= Single-Function Device
1= Multi-Function Device</td></tr>
<tr><td>06:00</td><td>1H</td><td>PCI Header Type - System initialization code reads this bit field to determine PCI header type. The i960 Rx I/O processor has a PCI-to-PCI bridge header as defined in <i>PCI-to-PCI Bridge Architecture Specification</i>, revision 1.0.</td></tr>
</table>

15.13.10 Primary Bus Number Register - PBNR

Primary Bus Number Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. Use this register to record the primary interface's bus number. This register decodes Type 1 configuration transactions on the secondary interface that are converted to Special Cycle transactions on the primary interface.

Table 15-22. Primary Bus Number Register - PBNR



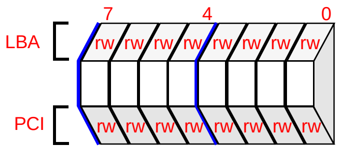
LBA: 1018H
PCI: 18H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
07:00	00H	Primary Bus Number - Programmed with the bridge's primary interface PCI bus number.

15.13.11 Secondary Bus Number Register - SBNR

Secondary Bus Number Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. Use this register to record the secondary interface's bus number. This register determines when to respond to Type 1 configuration commands on the primary interface and convert them to Type 0 commands on the secondary interface.

Table 15-23. Secondary Bus Number Register - SBNR



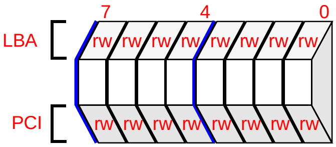
LBA: 1019H
PCI: 19H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
07:00	00H	Secondary Bus Number - This field is programmed with the bridge's secondary interface PCI bus number.

15.13.12 Subordinate Bus Number Register - SubBNR

Subordinate Bus Number Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. Use this register to record the highest numbered PCI bus on the bridge's secondary interface. This register is used in conjunction with the secondary bus number to determine when to respond to Type 1 configuration commands on the primary bus and pass them on to the secondary interface.

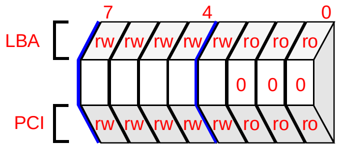
Table 15-24. Subordinate Bus Number Register - SubBNR

<table>
<tr><td colspan="3">
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 101AH
<b>PCI:</b> 1AH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>07:00</td><td>00H</td><td>Subordinate Bus Number - This field is programmed with the highest numbered PCI bus which exists on the bridge's secondary interface.</td></tr>
</table>

15.13.13 Secondary Latency Timer Register - SLTR

Secondary Latency Timer Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1 and apply to the secondary interface only. When the timer counts down to zero, the bridge must terminate the transaction when the GNT# signal is deasserted.

Table 15-25. Secondary Latency Timer Register - SLTR



LBA: 101BH
PCI: 1BH
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
07:03	00H	Programmable Latency Timer - Varies the latency timer for the secondary interface from 0 to 248 clocks.
02:00	000_2	Latency Timer Granularity - Read only; giving a programmable granularity of 8 clocks for the Latency Timer.

15.13.14 I/O Base Register - IOBR

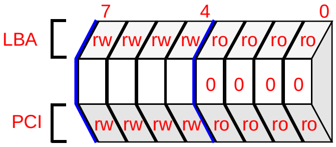
The I/O Base Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The I/O Base Register defines the bottom address (inclusive) of an address range that is used to determine when to forward I/O transactions from one bridge side to the other. It is programmed with a valid value before the Bridge Command Register’s I/O Space Enable bit is set. The bridge only supports 16-bit addressing; this is indicated by a value of 0H in the register’s four least significant bits. The upper four bits are programmed with AD15:12 for the bottom of the address range. The base address’ AD11:0 is always 000H, which forces the I/O address range to be 4 Kbyte aligned.

For address decoding, the bridge assumes that AD31:16, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per *PCI Local Bus Specification*, revision 2.1 and check that the upper 16 bits are equal to 0000H.

The I/O address range (defined by the IOBR in conjunction with the IOLR) is modified by the Bridge Control Register (BCR) ISA Enable bit. When set, the primary side does not accept I/O addresses in the range X400H - XFFFH, even when the address falls within the defined I/O address range.

The I/O address range defined by the IOBR/IOLR register pair does not perform inverse decoding of the register pair on the secondary interface when the Secondary Positive I/O Decode Enable bit is set in the Secondary Decode Enable Register (SDER).

Table 15-26. I/O Base Register - IOBR

 7 4 0 LBA rw rw rw rw ro ro ro ro 0 0 0 0 PCI rw rw rw rw ro ro ro ro		

LBA: 101CH
PCI: 1CH
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
07:04	0H	I/O Base Address - These four bits correspond to address bits AD15:12.
03:00	0H	I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16-bit I/O addressing.

15.13.15 I/O Limit Register - IOLR

The I/O Limit Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The I/O Limit Register defines the upper address (inclusive) of an address range that the bridge uses to determine when to forward I/O transactions from one side to the other.

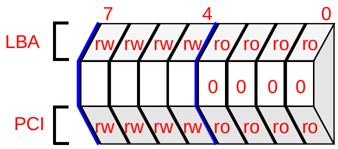
The application program must specify a valid value greater than or equal to the IOBR before the Bridge Command Register I/O Space Enable bit is set.

When the IOBR value is greater than the IOLR value, I/O cycles on either side of the bridge are indeterminate. The bridge only supports 16 bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper four bits are programmed with AD15:12 for the top of the address range. AD11:0 of the base address is always FFFH, which forces a 4 Kbyte I/O range granularity.

For address decoding, the bridge assumes that AD31:16, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per *PCI Local Bus Specification*, revision 2.1 and check that the upper 16 bits are equal to 0000H.

The I/O address range (defined by the IOBR in conjunction with the IOLR) is modified by the Bridge Control Register ISA Enable bit. When set, the primary side does not accept I/O addresses in the range X400H - XFFFH, even when the address falls within the defined I/O address range.

Table 15-27. I/O Limit Register - IOLR

 7 4 0 LBA rw rw rw rw ro ro ro ro 0 0 0 0 PCI rw rw rw rw ro ro ro ro		

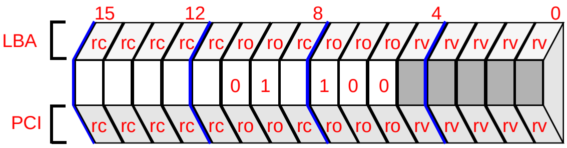
LBA: 101DH
PCI: 1DH
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
07:04	0H	I/O Limit Address - This field is programmed with AD15:12 of the top of the I/O address range to be passed down the hierarchy by the bridge.
03:00	0H	I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16 bit I/O addressing.

15.13.16 Secondary Status Register - SSR

Secondary Status Register bits adhere to the definitions in the *PCI-to-PCI Bridge Architecture Specification*, revision 1.0 and apply to the bridge's secondary interface only. The Read/Clear bits can only be set by the internal hardware and are cleared by either writing a 1_2 to the register or by a reset condition.

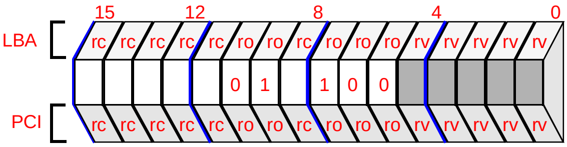
Table 15-28. Secondary Status Register - SSR (Sheet 1 of 2)

 15 12 8 4 0 LBA rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv 0 1 1 0 0 PCI rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv		

LBA: 101EH
PCI: 1EH
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15	0_2	Detected Parity Error - The bridge sets this bit when a parity error is detected on the secondary bus even when parity handling is disabled.

Table 15-28. Secondary Status Register - SSR (Sheet 2 of 2)


15
12
8
4
0
LBA
rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv
0 1 1 0 0
PCI
rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv

LBA: 101EH
PCI: 1EH
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

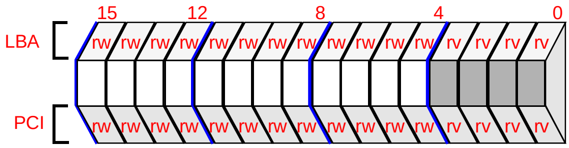
Bit	Default	Description
14	0_2	Received SERR# - When set indicates that the bridge detected S_SERR# on the secondary interface.
13	0_2	Master Abort - The bridge sets this bit when a transaction initiated by the secondary interface (except Special Cycles) ends in Master-abort
12	0_2	Target Abort (master) - The bridge sets this bit when a transaction initiated by the secondary interface ends in a Target-abort.
11	0_2	Target Abort (target) - The bridge sets this bit when the secondary interface, acting as a target, terminates a transaction with a Target-abort.
10:09	01_2	DEVSEL# Timing - Medium Decode Timing for the secondary interface.
08	0_2	Data Parity Error Detected - 0 = No data parity error 1 = The bridge sets this bit when: • the bridge asserted S_PERR# (or saw asserted) on the secondary bus • the bridge was transaction master when error occurred • the Parity Checking Enable bit is set in the BCR
07	1_2	Fast Back-to-Back Capable - Read only; always set. 1 = The secondary interface can accept Fast Back-to-Back transactions as a target.
06	0_2	User-Definable Features (UDF) Support - Read only; always cleared. 0= User Definable Features are not supported.
05	0_2	66 MHz Capable - Read only; always cleared. 0= 33 MHz operation is supported
04:00	00H	Reserved

15.13.17 Memory Base Register - MBR

Memory Base Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The Memory Base Register defines the bottom address (inclusive) of a memory-mapped I/O address range that is used to determine when to forward memory transactions from one bridge side to the other. The Memory Base Register must be programmed before the Bridge Command Register Memory Space Enable bit is set. The upper 12 bits correspond to AD31:20 of 32 bit addresses. For address decoding, the bridge assumes that AD19:0, the lower 20 address bits of the memory base address, are zero. This means that the bottom of the defined address range is aligned on a 1 Mbyte boundary.

The memory address range defined by the MBR/MLR register pair does not perform inverse decoding of the register pair on the secondary interface when the Secondary Positive I/O Decode Enable bit is set in the Secondary Decode Enable Register (SDER).

Table 15-29. Memory Base Register - MBR

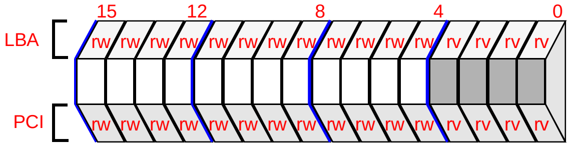
<table>
<tr><td colspan="3">
15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv

</td></tr>
<tr><td colspan="2"><b>LBA:</b> 1020H
<b>PCI:</b> 20H</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>15:04</td><td>000H</td><td>Memory Base Address - This field is programmed with AD31:20 of the bottom of the memory address range to be passed down the hierarchy by the bridge.</td></tr>
<tr><td>03:00</td><td>0H</td><td>Reserved</td></tr>
</table>

15.13.18 Memory Limit Register - MLR

The Memory Limit Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The Memory Limit Register defines the upper address (inclusive) of a memory-mapped I/O address range that is used to determine when to forward memory transactions from one bridge side to the other. The Memory Limit Register must be programmed to a value greater than or equal to the MBR before the Bridge Command Register Memory Space Enable bit is set. When the MLR value is not greater than or equal to the value of the MBR once the Memory Space Enable bit is set, memory transactions on either side of the bridge are indeterminate. The upper 12 bits correspond to AD31:20 of 32 bit addresses. For address decoding, the bridge assumes that AD19:0 **(**lower 20 bits of the memory base address) are FFFFFH. This forces a 1 Mbyte granularity on the memory address range.

Table 15-30. Memory Limit Register - MLR


15
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv

LBA: 1022H
PCI: 22H

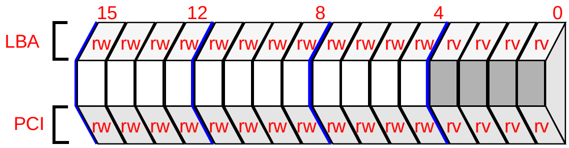
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:04	000H	Memory Limit Address - This field is programmed with AD31:20 of the top of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved

15.13.19 Prefetchable Memory Base Register - PMBR

The Prefetchable Memory Base Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The Prefetchable Memory Base Register defines the bottom address (inclusive) of a memory-mapped I/O address range that is used to determine when to forward memory transactions from one bridge side to the other. The Prefetchable Memory Base Register must be programmed before the Bridge Command Register Memory Space Enable bit is set. The upper 12 bits correspond to AD31:20 of 32 bit addresses. For address decoding, the bridge assumes that AD19:0, the lower 20 address bits of the memory base address, are zero. This means that the bottom of the defined address range is aligned on a 1 Mbyte boundary.

Table 15-31. Prefetchable Memory Base Register - PMBR


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv

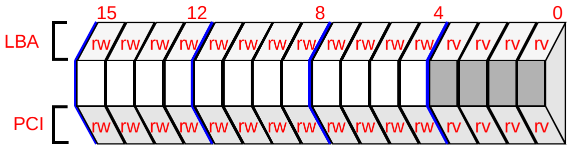
LBA: 1024H
PCI: 24H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:04	000H	Prefetchable Memory Base Address - This field is programmed with AD31:20 of the bottom of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved

15.13.20 Prefetchable Memory Limit Register - PMLR

The Prefetchable Memory Limit Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The Prefetchable Memory Limit Register defines the upper address (inclusive) of a memory-mapped I/O address range that is used to determine when to forward memory transactions from one bridge side to the other. The Prefetchable Memory Limit Register must be programmed to a value greater than or equal to the PMBR before the Bridge Command Register Memory Space Enable bit is set. When the PMLR value is not greater than or equal to the value of the PMBR once the Memory Space Enable bit is set, memory transactions on either bridge side are indeterminate. The upper 12 bits correspond to AD31:20 of 32 bit addresses. For address decoding, the bridge assumes that AD19:0, the lower 20 bits of the memory base address, are FFFFFH. This forces a 1 Mbyte granularity on the memory address range.

Table 15-32. Prefetchable Memory Limit Register - PMLR


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv

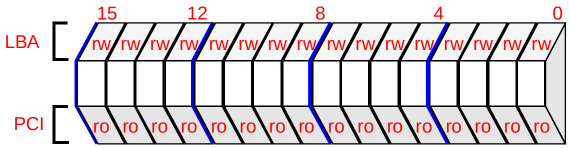
LBA: 1026H	**Legend:**	NA = Not Accessible	RO = Read Only	
PCI: 26H	RV = Reserved	PR = Preserved	RW = Read/Write	
	RS = Read/Set	RC = Read Clear		
	LBA = 80960 local bus address		PCI = PCI Configuration Address Offset	

Bit	Default	Description
15:04	000H	Prefetchable Memory Limit Address - This field is programmed with AD31:20 of the top of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved

15.13.21 Bridge Subsystem Vendor ID Register - BSVIR

The Bridge Subsystem Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1.

Table 15-33. Bridge Subsystem Vendor ID Register - BSVIR


15
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

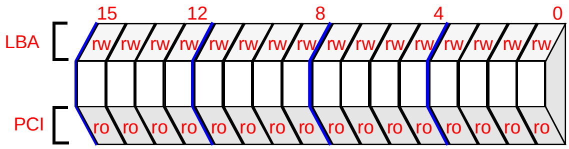
LBA: 1034H
PCI: 34H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15:00	0000H	Bridge Subsystem Vendor ID - This register is used to uniquely identify the vendor of the add-in board or subsystem

15.13.22 Bridge Subsystem ID Register - BSIR

The Bridge Subsystem ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1.

Table 15-34. Bridge Subsystem ID Register - BSIR


15
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

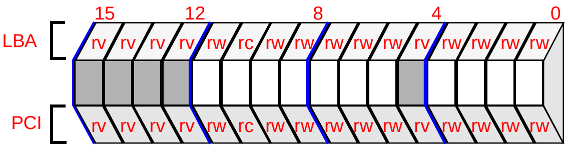
LBA: 1036H
PCI: 36H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15:00	0000H	Bridge Subsystem ID - This register is used to uniquely identify the add-in board or subsystem

15.13.23 Bridge Control Register - BCR

The Bridge Control Register bits adhere to the definitions in the *PCI Local Bus Specification*, revision 2.1. The Bridge Control Register bits provide extensions to the Command Register that are specific to PCI-to-PCI bridges. The Bridge Control Register provides many of the same controls for the secondary interface that are provided by the Command register for the primary interface. Some bits affect the operation of both bridge interfaces.

Table 15-35. Bridge Control Register - BCR (Sheet 1 of 3)

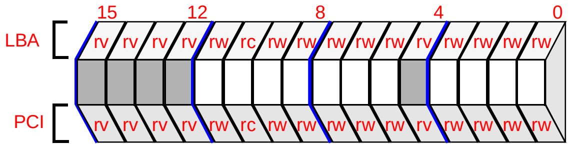

15
12
8
4
0
LBA
rv rv rv rv rw rc rw rw rw rw rw rv rw rw rw rw
PCI
rv rv rv rv rw rc rw rw rw rw rw rv rw rw rw rw

LBA: 103EH
PCI: 3EH

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:12	0H	Reserved
11[1]	0_2	Discard Timer SERR# Enable - This bit enables the assertion of SERR# for all discard timers. 0 - SERR# is not asserted when any discard timer expires. 1 - SERR# is asserted on the bus where the delayed request was initiated when the discard timer expires.
10[1]	0_2	Discard Timer Status - This bit indicates the status of the four discard timers. 0 - no discard timers have expired. 1 - at least one of the four discard timers has expired.
09[1]	0_2	Secondary Discard Timer Value - This bit controls the timeout value for the secondary delayed read and delayed write discard timers. 0 - the timeout value is 2**15 clocks. 1 - the timeout value is 2**10 clocks.
08[1]	0_2	Primary Discard Timer Value - This bit controls the timeout value for the secondary delayed read and delayed write discard timers. 0 - the timeout value is 2**15 clocks. 1 - the timeout value is 2**10 clocks.
07	0_2	Fast Back to Back Enable - This bit is ignored.

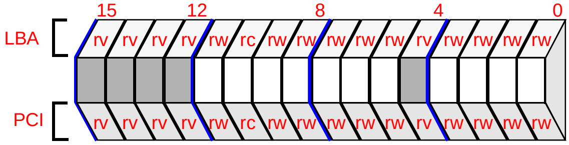
Table 15-35. Bridge Control Register - BCR (Sheet 2 of 3)


15
12
8
4
0
LBA
PCI
rv rv rv rv rw rc rw rw rw rw rw rv rw rw rw rw
rv rv rv rv rw rc rw rw rw rw rw rv rw rw rw rw

LBA: 103EH
PCI: 3EH
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
06	0_2	Secondary Bus Reset - This bit controls the secondary bus S_RST# signal. When set, • The PCI to PCI Bridge Unit resets all upstream and downstream buffers and address queues as well as the secondary PCI bus interface. The Bridge PCI configuration registers are not reset. The primary PCI bus interface retries all transactions — except Type 0 configuration transactions — until this bit is cleared. • DMA Channel 2 immediately halts any PCI transactions and returns to an idle state. DMA Channel 2 does not begin any new transfers until the Secondary Bus Reset bit is cleared. • Secondary ATU immediately halts all PCI transactions and completes all local bus transactions and its registers are reset. The 80960 core processor is released from bus backoff, if necessary. The Secondary ATU does not accept new 80960 core processor requests until the Secondary Bus Reset bit is cleared. When this bit is cleared, the S_RST# signal is deasserted. Software must clear this bit.
05[1]	0_2	Master Abort Mode - This bit controls the PCI-to-PCI bridge when a Master-abort termination occurs on either interface when the bridge is the master. When cleared, reads return all ones and write data is accepted by the bridge and dropped. When set, the bridge signals Master-abort to the requesting master when the corresponding transaction on the other side of the bridge terminates with a Master-abort and the transaction is not yet concluded (reads and non-posted writes). When the bit is set and the transaction on the requesting interface has completed (posted writes) the bridge must assert SERR# on the primary interface (providing the function is enabled).
04	0_2	Reserved

Table 15-35. Bridge Control Register - BCR (Sheet 3 of 3)


15
12
8
4
0
LBA
PCI
rv rv rv rv rw rc rw rw rw rw rw rv rw rw rw rw
rv rv rv rv rw rc rw rw rw rw rw rv rw rw rw rw

LBA: 103EH
PCI: 3EH

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

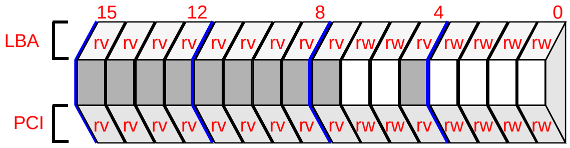
Bit	Default	Description
03	0_2	VGA Enable - This bit modifies the response by the bridge to VGA compatible addresses. When set, the bridge will positively decode and forward the following accesses from the primary to secondary interface (and block the forwarding of accesses from the secondary to primary interface): - Memory accesses where the range is 0A0000H - 0BFFFFH - I/O accesses where AD9:0 are in the range 3B0H - 3BBH and 3C0H - 3DFH (inclusive of ISA addresses - AD15:10 not decoded). VGA address forwarding (and blocking) is independent of the address ranges defined in the memory base registers and the I/O base register. It is also independent of the ISA Enable bit and the VGA Palette Snoop Enable bit. However, VGA address forwarding (and blocking) is dependent on the state of the I/O Enable bit and Memory Enable bit. When cleared, the bridge will not forward (or block) any VGA addresses.
02	Varies with external state of RETRY pin at primary PCI bus reset	ISA Enable - This bit modifies the bridges response to ISA I/O addresses. This only applies to I/O addresses that are defined by the bridge in IOBR and IOLR and are also in the first 64 Kbytes of PCI address space (0000 0000H - 0000 FFFFH) When set, the bridge will not forward from primary to secondary and I/O transactions addressing the last 768 bytes in each 1 Kbyte block. In the opposite direction, I/O transactions will be forwarded up the bridge if the address the last 768 bytes in each 1 Kbyte block.
01	Varies with external state of RST_MODE# pin at primary PCI bus reset	SERR# Enable - This bit controls the forwarding of secondary interface S_SERR# assertions to the primary interface. When the SERR# Enable bit in the PCMDR register is set and the bridge detects the assertion of S_SERR# on the secondary bus, it asserts P_SERR# on the primary interface.
00	0_2	Parity Error Response Enable - This bit controls the response to parity errors on the secondary interface. When this bit is clear, all address and data parity errors on the secondary interface are ignored. When this bit is set, detection and reporting of all parity errors on the secondary interface is enabled. Correct parity must be generated even when parity error reporting is disabled.

1. Bits 11-8 and 5 are not part of *PCI Local Bus Specification*, revision 2.1 and *PCI-to-PCI Bridge Architecture Specification*, revision 1.0.

15.13.24 Extended Bridge Control Register - EBCR

The Extended Bridge Control Register is used to control the extended functionality the bridge implements over the base *PCI-to-PCI Bridge Architecture Specification*, revision 1.0. It has enable/disable bits for the bridge's extended functionality.

Table 15-36. Extended Bridge Control Register - EBCR (Sheet 1 of 2)

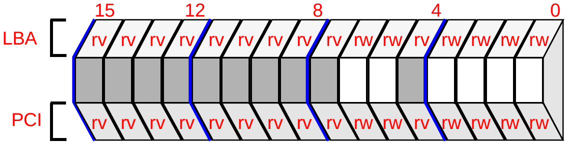

15
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rw rw rv rw rw rw rw
PCI
rv rv rv rv rv rv rv rv rv rw rw rv rw rw rw rw

LBA: 1040H
PCI: 40H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:07	000H	Reserved
06	0_2	DAC Cycle Enable - When set, DAC cycles are disabled. When clear, DAC cycles are enabled on the secondary PCI bridge interface.
05	0_2	Reset Local Bus - When set, the i960 core processor and all units on the local bus are reset. The PCI to PCI Bridge Unit is not reset. The i960 Rx I/O processor hardware clears this bit after the reset operation completes.
04	0_2	Reserved
03	0_2	Upstream Prefetchable Memory Enable - This bit only affects the Memory Read command on the secondary PCI bus. When set, the Bridge assumes that upstream Memory Read commands are to prefetchable memory. When clear, the Bridge assumes that upstream Memory Read commands are to non-prefetchable memory.
02	Varies with external state of RETRY pin at primary PCI bus reset	Configuration Cycle Retry - When set, the primary PCI interface of the i960 Rx I/O processor responds to all configuration cycles with a Retry condition. When clear, the i960 Rx I/O processor responds to the appropriate configuration cycles. This bit's default condition is based on the external state of the RETRY pin at the rising edge of P_RST#. When the external state of the pin is high, the bit is set. When the external state of the pin is low, the pin is cleared.
01	Varies with external state of RST_MODE# pin at primary PCI bus reset	Core Processor Reset - set by the hardware when either P_RST# is asserted or the Reset Local Bus bit in the EBCR is set. When set, the i960 core processor is being held in reset. Software cannot set this bit. Software must clear this bit to deassert 80960 processor reset. This bit's default condition is based on the external state of the RST_MODE# pin at the rising edge of P_RST#. When the external state of the pin is low, the bit is set. When the external state of the pin is high, the bit is cleared.

Table 15-36. Extended Bridge Control Register - EBCR (Sheet 2 of 2)

<table>
<tr><td colspan="3">
15
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rw rw rv rw rw rw rw
PCI
rv rv rv rv rv rv rv rv rv rw rw rv rw rw rw rw

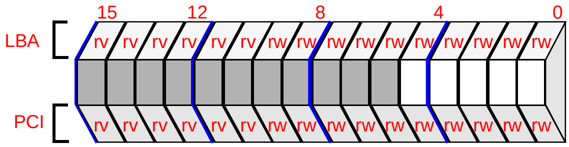
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 1040H
<b>PCI:</b> 40H</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>00</td><td>0_2</td><td>Posting Disable - When set, the bridge cannot post write transactions from either bridge interface. All transactions are processed as Delayed transactions. When clear, the bridge can post write transactions.</td></tr>
</table>

15.13.25 Secondary IDSEL Select Register - SISR

The Secondary IDSEL Select Register controls the usage of S_AD20:16 for 80960RP 33/5.0 and S_AD25:16 for 80960RP 33/3.3 in Type 1 to Type 0 conversions from the primary to secondary interface. In default operation, a unique encoding on primary addresses P_AD15:11 results in the assertion of one bit on the secondary address bus S_AD31:16 during a Type 1 to Type 0 conversion. See section 15.4.2, Special Cycles (pg. 15-8). This is used for the assertion of IDSEL on the device being targeted by the Type 0 configuration command. This register enables the use of secondary address bits (S_AD20:16 for 80960RP 33/5.0 and S_AD25:16 for 80960RP 33/3.3) to configure private PCI devices by forcing secondary address bits (S_AD20:16 for 80960RP 33/5.0 and S_AD25:16 for 80960RP 33/3.3) to all zeros during Type 1 to Type 0 conversions, regardless of the state of primary addresses P_AD15:11 (device number in Type 1 configuration command).

Before any address bit within S_AD20:16 can be used for private secondary PCI devices, the i960 core processor must guarantee that the corresponding bit in the SISR register is set before the host tries to configure the hierarchical PCI buses.

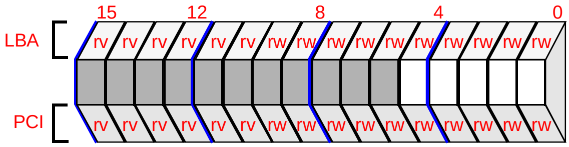
Table 15-37. Secondary IDSEL Select Register - SISR (Sheet 1 of 2)


15
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw
rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw

LBA: 1042H
PCI: 42H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:10	00H	Reserved
09[1]	0_2	AD25 - IDSEL Disable - When set, AD25 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD25 is asserted when primary addresses AD15:11 = 01001_2 during a Type 1 to Type 0 conversion.
08[1]	0_2	AD24 - IDSEL Disable - When set, AD24 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD24 is asserted when primary addresses AD15:11 = 01000_2 during a Type 1 to Type 0 conversion.
07[1]	0_2	AD23 - IDSEL Disable - When set, AD23 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD23 is asserted when primary addresses AD15:11 = 00111_2 during a Type 1 to Type 0 conversion.
06[1]	0_2	AD22 - IDSEL Disable - When set, AD22 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD22 is asserted when primary addresses AD15:11 = 00110_2 during a Type 1 to Type 0 conversion.

Table 15-37. Secondary IDSEL Select Register - SISR (Sheet 2 of 2)


15
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw
rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw

LBA: 1042H
PCI: 42H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
05[1]	0_2	AD21 - IDSEL Disable - When set, AD21 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD21 is asserted when primary addresses AD15:11 = 00101_2 during a Type 1 to Type 0 conversion.
04	0_2	AD20 - IDSEL Disable - When set, AD20 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD20 is asserted when primary addresses AD15:11 = 00100_2 during a Type 1 to Type 0 conversion.
03	0_2	AD19 - IDSEL Disable - When set, AD19 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD19 is asserted when primary addresses AD15:11 = 00011_2 during a Type 1 to Type 0 conversion.
02	0_2	AD18 - IDSEL Disable - When set, AD18 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD18 is asserted when primary addresses AD15:11 = 00010_2 during a Type 1 to Type 0 conversion.
01	0_2	AD17 - IDSEL Disable - When set, AD17 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD17 is asserted when primary addresses AD15:11 = 00001_2 during a Type 1 to Type 0 conversion.
00	0_2	AD16 - IDSEL Disable - When set, AD16 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD16 is asserted when primary addresses AD15:11 = 00000_2 during a Type 1 to Type 0 conversion.

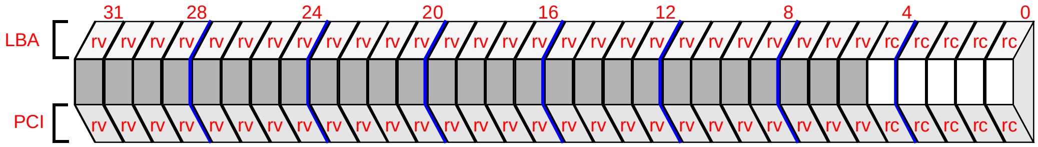
[1] This bit is available on 3.3 V devices only (not on 5.0 Volt devices).

15.13.26 Primary Bridge Interrupt Status Register - PBISR

The Primary Bridge Interrupt Status Register is used to notify the i960 core processor of the source of a Primary Bridge interface interrupt. In addition, this register is written to clear the source of the interrupt to the interrupt unit of the i960 Rx I/O processor.

Bits 4:0 are a direct reflection of bit 8 and bits 14:11 (respectively) of the Primary Status Register (these bits are set at the same time by hardware but need to be cleared independently). The conditions that result in a Primary Bridge interrupt are cleared by writing a 1 to the appropriate bits in this register.

Table 15-38. Primary Bridge Interrupt Status Register - PBISR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc

LBA: 1044H
PCI: 44H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

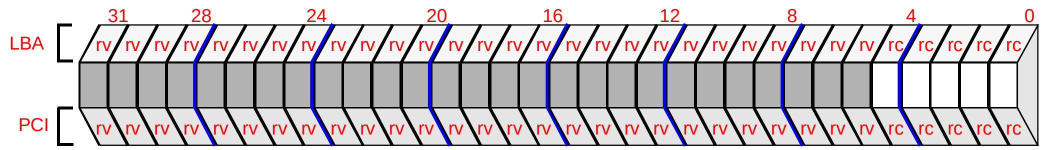
Bit	Default	Description
31:05	0000 000H	Reserved
04	0_2	P_SERR# Detected - set when P_SERR# is detected on the primary PCI bus.
03	0_2	PCI Master Abort - set when a transaction initiated by the primary master interface ends in a Master-abort.
02	0_2	PCI Target Abort (master) - set when a transaction initiated by the primary master interface ends in a Master-abort.
01	0_2	PCI Target Abort (target) - set when the primary interface, acting as a target, terminates the transaction on the PCI bus with a target abort.
00	0_2	PCI Master Parity Error - The primary interface sets this bit when three conditions are met: 1) the bus agent asserted S_PERR# itself or observed S_PERR# asserted 2) the agent setting the bit acted as the bus master for the operation in which the error occurred 3) the parity error response bit (bridge control register) is set

15.13.27 Secondary Bridge Interrupt Status Register - SBISR

The Secondary Bridge Interrupt Status Register is used to notify the i960 core processor of the source of a Secondary Bridge interface interrupt. In addition, this register is written to clear the source of the interrupt to the interrupt unit of the i960 Rx I/O processor.

Bits 4:0 are a direct reflection of bit 8 and bits 14:11 (respectively) of the Secondary Status Register (these bits are set at the same time by hardware but need to be cleared independently). The conditions that result in a Secondary Bridge interrupt are cleared by writing a 1 to the appropriate bits in this register.

Table 15-39. Secondary Bridge Interrupt Status Register - SBISR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc

LBA: 1048H
PCI: 48H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:05	0000 000H	Reserved
04	0_2	S_SERR# Asserted - set when S_SERR# is asserted on the secondary PCI bus.
03	0_2	PCI Master Abort - set when a transaction initiated by the secondary master interface ends in a Master-abort.
02	0_2	PCI Target Abort (master) - set when a transaction initiated by the secondary master interface ends in a Master-abort.
01	0_2	PCI Target Abort (target) - set when the secondary interface, acting as a target, terminates the transaction on the PCI bus with a target abort.
00	0_2	PCI Master Parity Error - The secondary interface sets this bit when three conditions are met: 1) the bus agent asserted P_PERR# itself or observed P_PERR# asserted 2) the agent setting the bit acted as the bus master for the operation in which the error occurred 3) the parity error response bit (command register) is set

15.13.28 Secondary Arbitration Control Register - SACR

See CHAPTER 18, BUS ARBITRATION, for a description of the Secondary Arbitration Control Register.

15.13.29 PCI Interrupt Routing Select Register - PIRSR

Refer to CHAPTER 8, INTERRUPTS for a description of the PCI Interrupt Routing Select Register.

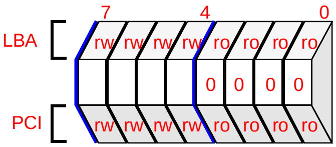
15.13.30 Secondary I/O Base Register - SIOBR

The Secondary I/O Base Register bits are used when the secondary PCI interface is enabled for positive decoding. The Secondary I/O Base Register defines the bottom address (inclusive) of a positively decoded address range that is used to determine when to forward I/O transactions from the secondary interface to the primary interface. It must be programmed with a valid value before the Secondary Decode Enable Register (SDER) is set. The bridge only supports 16-bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper four bits are programmed with S_AD15:12 for the bottom of the address range. S_AD11:0 of the base address is always 000H forcing the secondary I/O address range to be 4 Kbyte aligned.

For address decoding, the bridge assumes that S_AD31:16, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per *PCI Local Bus Specification*, revision 2.1 and check that the upper 16 bits are equal to 0000H.

The positive secondary I/O address range (defined by the SIOBR in conjunction with the SIOLR) is not affected by the state of the ISA Enable bit in the Bridge Control Register (BCR).

Table 15-40. Secondary I/O Base Register - SIOBR



LBA: 1054H
PCI: 54H
		Legend:
		RV = Reserved
		RS = Read/Set
		LBA = 80960 local bus address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
		RO = Read Only
		RW = Read/Write

Bit	**Default**	**Description**
07:04	0H	Secondary I/O Base Address - This field is programmed with S_AD15:12 of the bottom of the positively decoded secondary I/O address range to be passed from the secondary to the primary side.
03:00	0H	I/O Addressing Capability - A value of 0H signifies that the bridge only supports 16 bit I/O addressing.

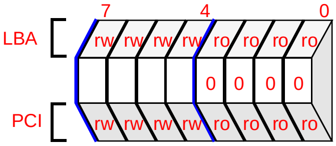
15.13.31 Secondary I/O Limit Register - SIOLR

The Secondary I/O Limit Register bits are used when the secondary PCI interface is enabled for positive decoding. The Secondary I/O Limit Register defines the upper address (inclusive) of a positively decoded secondary address range that is used to determine when to forward I/O transactions from the secondary to primary interface. It must be programmed with a valid value greater than or equal to the SIOBR before the I/O Space Enable bit in the Bridge Command Register and the Secondary Positive I/O Decode Enable bit in the Secondary Decode Enable Register (SDER) are set. When the SIOBR value is greater than the SIOLR value, I/O cycles forwarded from the secondary to primary interface (positively decoded) are undefined. The bridge only supports 16 bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper four bits are programmed with S_AD15:12 for the top of the address range. S_AD11:0 of the base address is always FFFH forcing a 4 Kbyte I/O range granularity.

For address decoding, the bridge assumes that S_AD31:16, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per *PCI Local Bus Specification*, revision 2.1 and check that the upper 16 bits are equal to 0000H.

The Secondary I/O address range (defined by the SIOBR in conjunction with the SIOLR) is not modified by the Bridge Control Register ISA Enable bit.

Table 15-41. Secondary I/O Limit Register - SIOLR



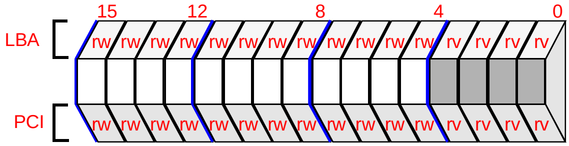
LBA: 1055H
PCI: 55H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
07:04	0H	Secondary I/O Limit Address - This field is programmed with S_AD15:12 of the top of the positively decoded I/O address range to be passed from the secondary to primary interface.
03:00	0H	Secondary I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16-bit I/O addressing.

15.13.32 Secondary Memory Base Register - SMBR

The Secondary Memory Base Register bits are used when the secondary interface is enabled for positive address decoding. They are also used to define a private address space on the secondary PCI bus when the Private Memory Space Enable bit is set in the SDER. The Secondary Memory Base Register defines the bottom address (inclusive) of a memory-mapped address range that is used to determine when to forward transactions from the secondary to primary interface. The Secondary Memory Base Register must be programmed with a valid value before the Secondary Positive Memory Decode Enable bit in the SDER is set. The upper 12 bits correspond to S_AD31:20 of 32 bit addresses. For address decoding, the bridge assumes that S_AD19:0, the lower 20 address bits of the memory base address, are zero. This means that the bottom of the defined address range is aligned on a 1 Mbyte boundary.

Table 15-42. Secondary Memory Base Register - SMBR


15
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv

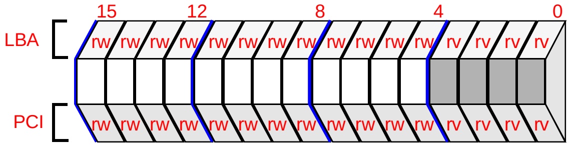
LBA: 1058H
PCI: 58H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:04	000H	Secondary Memory Base Address - This field is programmed with S_AD31:20 of the bottom of the positively decoded secondary memory address range to be passed from the secondary to primary interface.
03:00	0H	Reserved

15.13.33 Secondary Memory Limit Register - SMLR

The Secondary Memory Limit Register bits are used when the secondary interface is enabled for positive address decoding. They are also used to define a private memory space on the secondary PCI bus, when the Private Memory Space Enable bit is set in the SDER. The Secondary Memory Limit Register defines the upper address (inclusive) of a memory-mapped address range that is used to determine when to forward transactions from the secondary to primary interface. The Secondary Memory Limit Register must be programmed to a value greater than or equal to the SMBR before the Private Memory Space Enable bit and the Secondary Positive Memory Decode Enable bit in the SDER are set. When the SMLR value is not greater than or equal to the SMBR value, once the Memory Space Enable bit or Secondary Memory Enable bit are set, positively decoded memory transactions from the secondary to the primary are indeterminate. The upper 12 bits correspond to S_AD31:20 of 32 bit addresses. For address decoding, the bridge assumes that S_AD19:0, the lower 20 address bits of the secondary memory base address, are FFFFFH. This forces a 1 Mbyte granularity on the memory address range.

Table 15-43. Secondary Memory Limit Register - SMLR


15
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv

LBA: 105AH
PCI: 5AH
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:04	000H	Secondary Memory Limit Address - This field is programmed with S_AD31:20 of the top of the secondary memory address range to be passed from the secondary to primary side.
03:00	0H	Reserved

15.13.34 Secondary Decode Enable Register - SDER

The Secondary Decode Enable Register is used to control the address decode functions on the secondary PCI interface of the bridge unit. The Secondary Positive I/O Decode Enable bit, when set, causes the bridge to decode and claim transactions within the address range defined by the SIOBR/SIOLR address pair and forward them through the bridge unit. The Secondary Positive Memory Decode Enable bit has the same function as the Secondary Positive I/O Decode Enable bit but works with the SMBR/SMLR address range. Setting either of these bits disables all inverse decoding on the secondary interface.

Table 15-44. Secondary Address Decode (Sheet 1 of 2)

Private Memory Space Enable	Secondary Positive Memory Enable bit	Secondary Positive I/O Enable bit	Secondary Decode
0	0	0	Inverse Decoding (default)
0	0	1	Secondary Positive I/O Decoding only

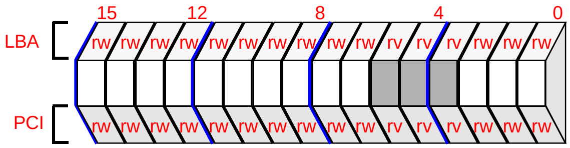
Table 15-44. Secondary Address Decode (Sheet 2 of 2)

Private Memory Space Enable	Secondary Positive Memory Enable bit	Secondary Positive I/O Enable bit	Secondary Decode
0	1	0	Secondary Positive Memory Decoding only
0	1	1	Secondary Positive I/O Decoding and Secondary Positive Memory Decoding
1	X	X	Inverse Decoding and SMBR/SMLR address pair define a private address space

The Private Memory Space Enable bit enables the creation of a private memory space on the secondary PCI bus. This bit is used in conjunction with the SMBR/SMLR registers. When set, the bridge ignores transactions with addresses within the SMBR/SMLR address range. It also disables secondary positive decode.

Bits 15 through 4 of the Secondary Decode Enable Register (SDER) can be used to mask sources of NMI# from the bridge. When set to 1, the source of NMI# is masked. When cleared to 0, the source of NMI# is enabled.

Table 15-45. Secondary Decode Enable Register - SDER (Sheet 1 of 2)

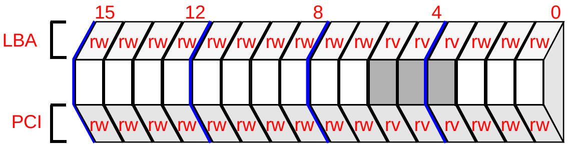

15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rv rv rv rw rw rw
PCI
rw rw rw rw rw rw rw rw rw rw rv rv rv rw rw rw

LBA: 105CH
PCI: 5CH

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
15[1]	1_2	S_SERR# Detected Interrupt Mask - When set, detecting S_SERR# on the secondary interface resulting in bit 14 of the SSR being set will *not* result in bit 4 of the SBISR being set. When clear, an error that sets bit 14 of the SSR will cause bit 4 of the SBISR to be set
14	1_2	Secondary PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the SSR being set will *not* result in bit 3 of the SBISR being set. When clear, an error that sets bit 13 of the SSR will cause bit 3 of the SBISR to be set.
13	1_2	Secondary PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the SSR being set will *not* result in bit 2 of the SBISR being set. When clear, an error that sets bit 12 of the SSR will cause bit 2 of the SBISR to be set.
12	1_2	Secondary PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the SSR being set will *not* result in bit 1 of the SBISR being set. When clear, an error that sets bit 11 of the SSR will cause bit 1 of the SBISR to be set.

Table 15-45. Secondary Decode Enable Register - SDER (Sheet 2 of 2)


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rv rv rv rw rw rw
PCI
rw rw rw rw rw rw rw rw rw rw rv rv rv rw rw rw

LBA: 105CH
PCI: 5CH

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 local bus address
PCI = PCI Configuration Address Offset

Bit	Default	Description
11	1_2	Secondary PCI Master Parity Error Interrupt Mask - When set a parity error resulting in bit 8 of the SSR being set will *not* result in bit 0 of the SBISR being set. When clear, an error that sets bit 8 of the SSR will cause bit 0 of the SBISR to be set.
10	1_2	P_SERR# Asserted Interrupt Mask - When set, detecting or asserting P_SERR# on the primary interface resulting in bit 14 of the PSR being set will *not* result in bit 4 of the PBISR being set. When clear, an error that sets bit 14 of the PSR will cause bit 4 of the PBISR to be set.
09	1_2	Primary PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the PSR being set will *not* result in bit 3 of the PBISR being set. When clear, an error that sets bit 13 of the PSR will cause bit 3 of the PBISR to be set.
08	1_2	Primary PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the PSR being set will *not* result in bit 2 of the PBISR being set. When clear, an error that sets bit 12 of the PSR will cause bit 2 of the PBISR to be set.
07	1_2	Primary PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the PSR being set will *not* result in bit 1 of the PBISR being set. When clear, an error that sets bit 11 of the PSR will cause bit 1 of the PBISR to be set.
06	1_2	Primary PCI Master Parity Error Interrupt Mask - When set a parity error resulting in bit 8 of the PSR being set will *not* result in bit 0 of the PBISR being set. When clear, an error that sets bit 8 of the PSR will cause bit 0 of the PBISR to be set.
05:03	000_2	Reserved.
02	0_2	Private Memory Space Enable - when set, this bit disables Bridge forwarding of addresses in the SMBR/SMLR address range. This creates a private memory space on the secondary PCI bus for peer to peer transactions.
01	0_2	Secondary Positive Memory Decode Enable - when set, this bit enables the secondary interface of the bridge unit to positively decode memory addresses on the secondary bus. Addresses within the SMBR/SMLR address range are forwarded through the bridge. Inverse decoding is disabled.
00	0_2	Secondary Positive I/O Decode Enable - when set, this bit enables the secondary interface of the bridge unit to positively decode I/O addresses on the secondary bus. Addresses within the SIOBR/SIOLR address pair are forwarded through the bridge. Inverse decoding is disabled.

[1] Bits 15:06 do not exist for 5.0 Volt device.

16

ADDRESS TRANSLATION UNIT

CHAPTER 16
ADDRESS TRANSLATION UNIT

The Address Translation Unit (ATU) is the interface between the PCI buses and the 80960 local bus. This chapter describes ATU operation modes, setup, and interface.

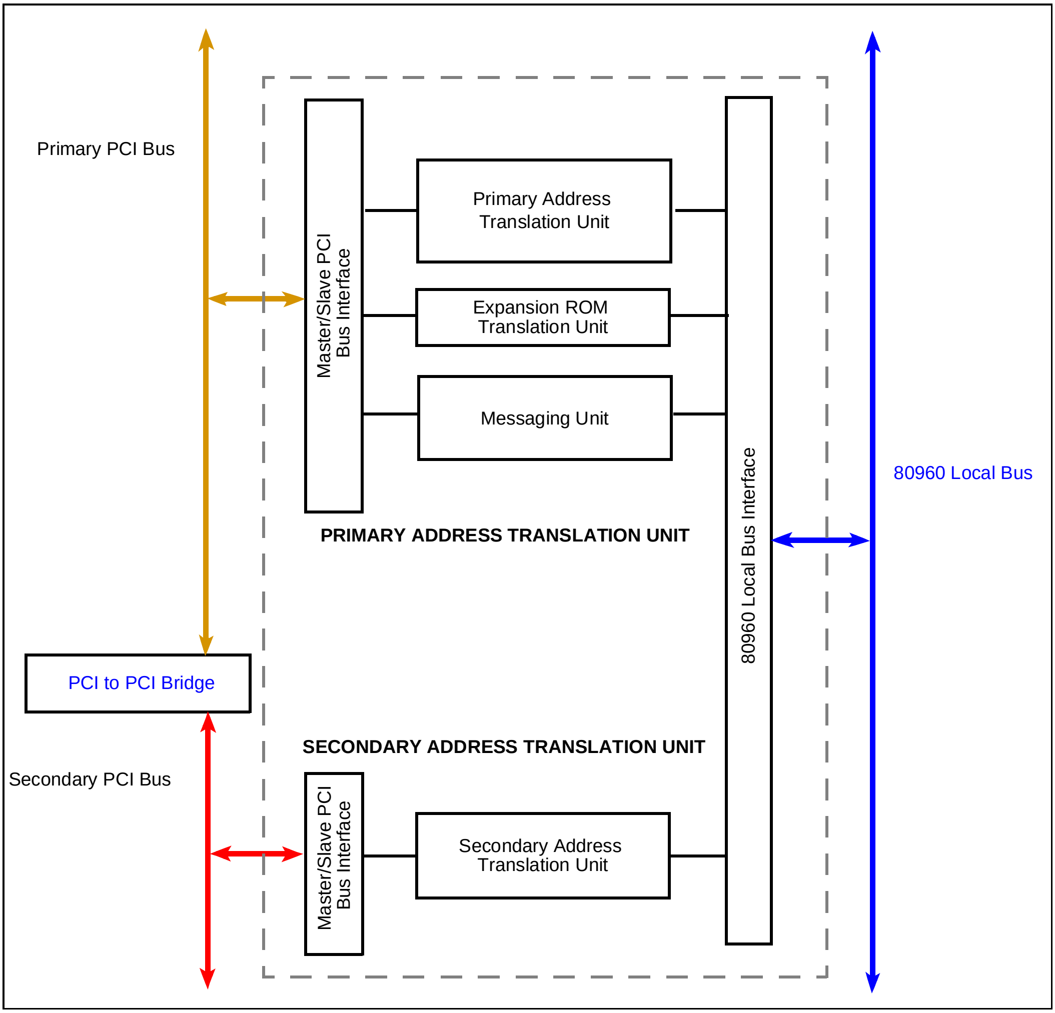

Primary PCI Bus
Master/Slave PCI Bus Interface
Primary Address Translation Unit
Expansion ROM Translation Unit
Messaging Unit
80960 Local Bus Interface
80960 Local Bus
PRIMARY ADDRESS TRANSLATION UNIT
PCI to PCI Bridge
Secondary PCI Bus
SECONDARY ADDRESS TRANSLATION UNIT
Master/Slave PCI Bus Interface
Secondary Address Translation Unit

Figure 16-1. Address Translation Unit (ATU) Block Diagram

16.1 OVERVIEW

As indicated in Figure 16-1, the ATU — the interface between the PCI bus and the on-chip 80960 local bus — consists of two address translation units and the Messaging Unit (MU); described in CHAPTER 17, MESSAGING UNIT. The MU allows the system processor and the i960® Rx I/O processor to transfer control information.

The ATUs support both inbound and outbound address translation. The ATUs are:

- Primary ATU — provides direct access between the primary PCI bus and the 80960 local bus. The primary ATU and MU share PCI address space.
- Secondary ATU — provides direct access between the secondary PCI bus and the 80960 local bus.

Both ATUs and the MU appear as a single PCI device on the primary PCI bus. Collectively, these units are the second PCI function (Function 1) in the multifunction PCI device.

Transactions initiated on a PCI bus and targeted at the 80960 local bus are referred to as *inbound transactions* (PCI to 80960 local bus); transactions initiated on the 80960 local bus and targeted at a PCI bus are referred to as *outbound transactions* (80960 local bus to PCI). The ATUs handle multiple inbound PCI transactions; they can simultaneously process PCI read and write transactions using both primary and secondary ATUs.

During inbound transactions, the ATU converts PCI addresses (initiated by a PCI bus master) to 80960 local bus addresses and initiates the data transfer on the 80960 local bus. During outbound transactions, the ATU converts 80960 local bus addresses to PCI addresses and initiates the data transfer on the respective PCI bus.

The ATUs do not support outbound transactions generated by the DMA controller or the opposite ATU. The ATUs also do not claim PCI transactions from the PCI-to-PCI bridge or the DMA controller, unless the Secondary Messaging Unit access enable (bit 12) is set.

16.2 ATU TRANSACTION QUEUES

ATU operation and performance depends on the queueing mechanism implemented between the local bus interface and PCI bus interface. As indicated in Figure 16-2, the ATU transaction queues consist of three address queues and two data queues; each are described in the following subsections.

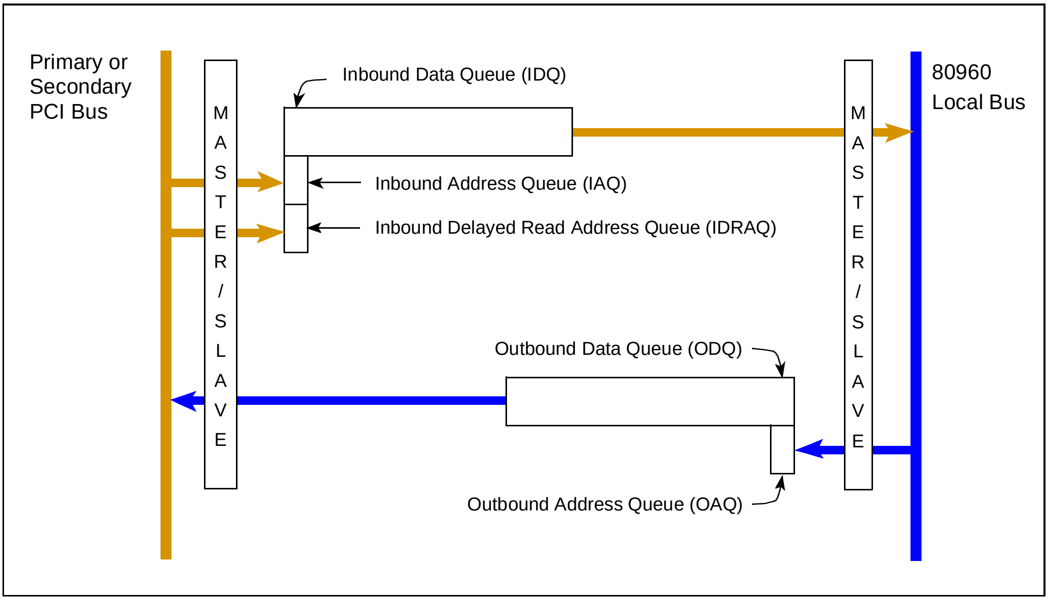

Primary or Secondary PCI Bus
MASTER/SLAVE
Inbound Data Queue (IDQ)
Inbound Address Queue (IAQ)
Inbound Delayed Read Address Queue (IDRAQ)
Outbound Data Queue (ODQ)
Outbound Address Queue (OAQ)
MASTER/SLAVE
80960 Local Bus

Figure 16-2. ATU Transaction Queue Block Diagram

16.2.1 Address Queues

As indicated in Figure 16-2, ATU transaction queues contain three separate address queues:

- Inbound Delayed Read Address Queue (IDRAQ)
- Inbound Address Queue (IAQ)
- Outbound Address Queue (OAQ)

These queues, each of which hold a single 32-bit address, forward transactions from one side of the queue structure to the other.

The ATU PCI interface uses IDRAQ for inbound read operations and IAQ for write operations. The 32-bit PCI address is latched into the inbound address queues and translated to the 80960 local bus address and driven onto the local bus by the ATU local bus interface.

The ATU local bus interface uses OAQ for outbound read and write operations. The 32-bit 80960 local address is latched into the OAQ and translated to a PCI address and driven onto the PCI bus by the ATU PCI interface.

The address queue is always initialized by the initiating bus and cleared by the target bus under normal termination. The address queue is also cleared by a bus when an error has occurred on that bus. This effectively cancels the transaction and clears the queue, allowing a new transaction to be initiated.

16.2.2 Data Queues

The ATU transaction queue contains two separate data queues:

- Inbound Data Queue (IDQ)
- Outbound Data Queue (ODQ)

Each 64-byte queue is arranged in a 16 x 32-bit (1 DWORD) configuration. The ATU PCI interface uses the IDQ to hold inbound write data; the ATU local bus uses IDQ to return outbound read data. The ATU local bus interface uses ODQ for outbound write data and the ATU PCI interface to return inbound read data. Data in the queues is invalidated only on error conditions (see Section 16.6).

16.3 ATU ADDRESS TRANSLATION

The ATUs implement an address windowing scheme to determine which addresses to claim and translate to the appropriate bus.

- The address windowing mechanism for inbound translation is described in Section 16.3.1, Inbound Address Translation (pg. 16-5)
- The address windowing mechanism for outbound translation is described in Section 16.3.6, Outbound Address Translation (pg. 16-11)

The primary ATU contains a data path between the primary PCI bus and 80960 local bus. Connecting the primary ATU in this manner enables data transfers to occur without requiring any resources on the secondary PCI bus. The secondary ATU contains a data path between the secondary PCI bus and the 80960 local bus. The secondary ATU allows secondary PCI bus masters to directly access the 80960 local bus and memory. These transactions are initiated by a secondary bus master and do not require any bandwidth on the primary PCI bus.

The ATU units allow for recognition and generation of multiple PCI cycle types. Table 16-1 shows the PCI commands supported by both inbound and outbound ATUs. The type of operation seen by the inbound ATUs is determined by the PCI master (on either primary or secondary bus) who initiates the transaction. Claiming an inbound transaction depends on the address being within the programmed inbound translation window. The type of transaction used by the outbound ATUs is determined by the 80960 local address and the fixed outbound windowing scheme. See Section 16.3.6, Outbound Address Translation (pg. 16-11) for the full details on outbound PCI cycle selection.

Both ATUs support the 64-bit addressing extension specified by the *PCI Local Bus Specification*, revision 2.1. This 64-bit addressing extension is for outbound data transactions only (i.e., data transfers initiated by the i960 core processor).

Neither ATU supports exclusive access using the PCI LOCK# signal. To achieve exclusive access, use a software protocol or the Messaging Unit.

The ATUs do not guarantee atomicity when performing atomic accesses using 80960 atomic instructions (**atmod**, **atadd**, etc.).

Table 16-1. ATU Command Support

PCI Command Type	Claimed on Inbound Transactions	Generated by Outbound Transactions
Interrupt Acknowledge	No	No
Special Cycle	No	No
I/O Read	No	**Yes**
I/O Write	No	**Yes**
Memory Read	**Yes**	**Yes**
Memory Write	**Yes**	**Yes**
Memory Write and Invalidate	**Yes**	No
Memory Read Line	**Yes**	No
Memory Read Multiple	**Yes**	No
Configuration Read	**Yes**	**Yes**
Configuration Write	**Yes**	**Yes**
Dual Address Cycle	No	**Yes**

16.3.1 Inbound Address Translation

The ATUs allow PCI bus masters to directly access the 80960 local bus. These PCI bus masters can read or write i960 Rx I/O processor memory-mapped registers or 80960 local memory space. The transactions where PCI bus masters are accessing the 80960 local bus are called *inbound transactions*. Inbound translation involves two steps:

1. Address Detection.
   - Determine when the 32-bit PCI address is within the address window defined for the inbound ATU (primary or secondary).
   - Claim the PCI transaction with medium DEVSEL# timing.
2. Address Translation.
   - Translate the 32-bit PCI address to a 32-bit 80960 local bus address.

The ATUs use the following registers in inbound address translation:

- Inbound ATU Base Address Register
- Inbound ATU Limit Register
- Inbound ATU Translate Value Register

See Section 16.7, REGISTER DEFINITIONS (pg. 16-25) for details on inbound translation register definition and programming constraints.

By convention, primary inbound ATU addresses are primary PCI addresses; secondary inbound ATU addresses are secondary PCI addresses. In the event that an address can be claimed by both the ATU and the bridge, the inbound ATU PCI interface has priority.

Inbound address detection is determined from the 32-bit PCI address, the base address register and the limit register. The algorithm for detection is:

> When PCI_Address & Limit_Register == Base_Register
> the PCI Address is claimed by the Inbound ATU

Figure 16-3 shows an example of inbound address detection and inbound translation windows.

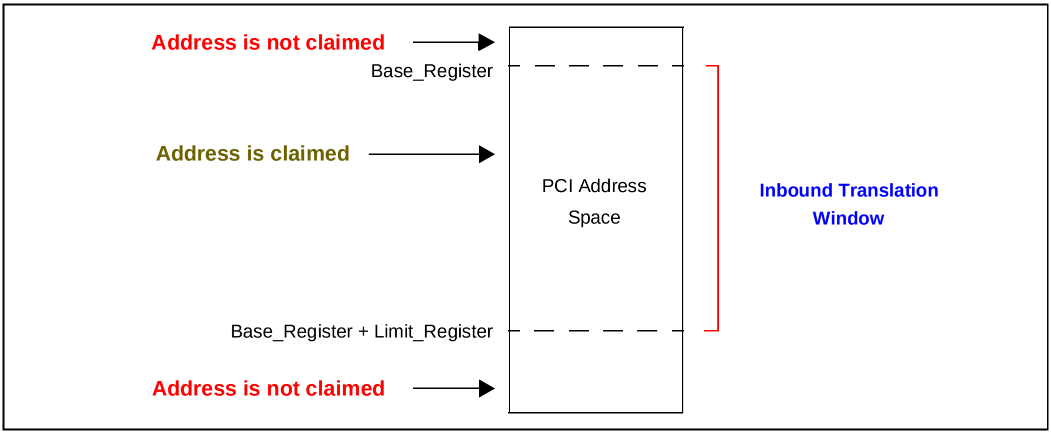


Figure 16-3. Inbound Address Detection

The incoming 32-bit PCI address is bitwise ANDed with the associated inbound limit register. When the result matches the base register, the inbound PCI address is detected as being within the inbound translation window and is claimed by the ATU.

NOTE: The first 4 Kbytes of the primary ATU's inbound address translation window are reserved for the Messaging Unit. See Section 16.4, MESSAGING UNIT (pg. 16-19).

Once the transaction is claimed, the address within the Inbound Address Queue (IAQ) must be translated from a 32-bit PCI address to a 32-bit 80960 local bus address. The algorithm is:

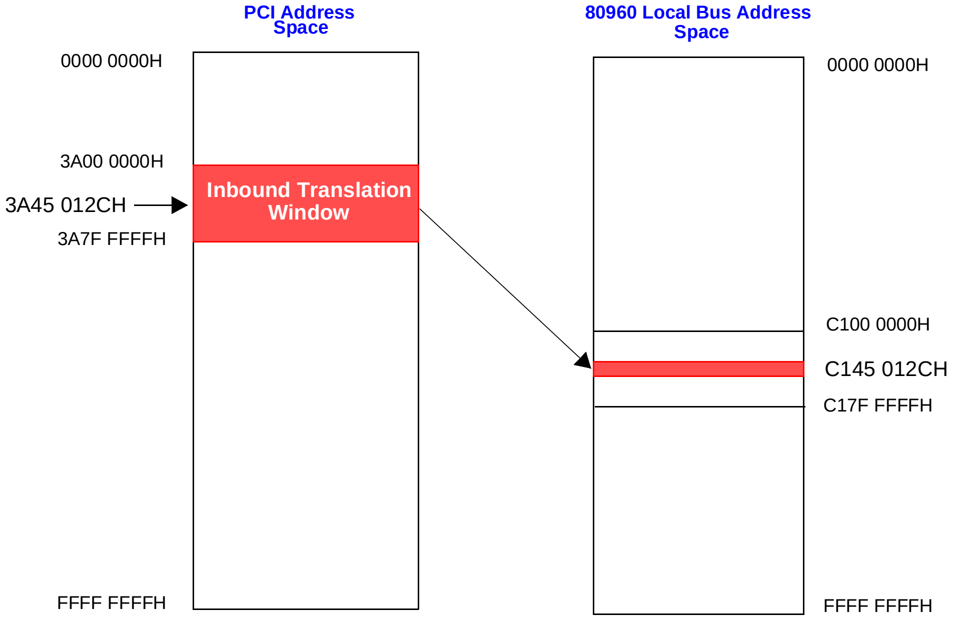
80960_Address = (PCI_Address & ~Limit_Register) | Translate_Register

The incoming 32-bit PCI address is first bitwise ANDed with the bitwise inverse of the limit register. This result is bitwise ORed with the translate value register and the result is the 80960 local address. The translate value register must be aligned on the limit register boundary. For example, if the limit register is 8 Mbytes, the translate value register must point to an 8 Mbyte boundary on the 80960 local bus. This translation mechanism is used for all inbound memory read and write commands excluding inbound configuration read and writes. Inbound configuration cycle translation is described in Section 16.3.4 (pg. 16-11). Address aliasing of multiple PCI addresses to the same physical 80960 local bus address can be prevented by programming the inbound limit register on boundaries matching the associated limit register, but this is only enforced through application programming.

For inbound memory transactions, the only burst order supported is Linear Incrementing. For any other burst order, the ATU signals a Disconnect after the first data phase.

For inbound address translation, the physical memory attribute for the 80960 local bus must be 32-bits wide. See Section 12.1.1 (pg. 12-1). The only exception is the expansion ROM window can be in 8-bit wide memory.

Figure 16-4 shows an inbound translation example. This example would hold true for an inbound transaction from either the primary or secondary PCI bus.



Register Values

Base_Register = 3A00 0000H

Limit_Register = FF80 0000H
(8 Mbyte limit value)

Translate_Register = C100 0000H

Inbound Translation Window ranges from
3A00 0000H to 3A7F FFFFH (8 Mbytes)

Address Detection

PCI_Address & Limit_Register == Base_Register
3A45 012CH & FF80 0000H == 3A00 0000H
PCI_Address is in the Inbound Translation Window

Address Translation

80960_Address = (PCI_Address & ~Limit_Register) | Translate_Register
80960_Address = (3A45 012CH & 007F FFFFH) | C100 0000H
80960_Address =C145 012CH

Figure 16-4. Inbound Translation Example

16.3.2 Inbound Write Transaction

An inbound write transaction is initiated by a PCI master (on either the primary or secondary PCI bus) and is targeted at either 80960 local memory or an 80960 local bus memory-mapped register. Data flow for an inbound write transaction on the PCI bus is summarized as:

- The ATU claims the PCI write transaction when the PCI address is within the inbound translation window defined by the ATU Inbound Base Register and Inbound Limit Register.
- When no transaction is currently in the IAQ or inbound data queue (IDQ), the ATU latches the PCI address into the IAQ. When an inbound write transaction is currently in progress, the ATU does not latch the PCI address and signals a Retry to the initiator.
- Once the PCI address is in the IAQ, the PCI interface can start accepting write data and store it in the IDQ.
- The PCI interface continues to accept write data until one of the following is true:
    - The initiator completes the transaction.
    - The IDQ becomes full. In this case, the PCI interface signals a Disconnect to the initiator.

Once the PCI interface places a PCI address in the IAQ, the ATU's local bus interface becomes aware of the inbound write. The ATU local bus interface completes the inbound write on the 80960 local bus.

Data flow for the inbound write transaction on the 80960 local bus is summarized as:

- The ATU local bus interface requests the 80960 local bus when a PCI address appears in the IAQ.
- When the 80960 local bus is granted, the local bus interface initiates the write transaction by driving the translated address onto the 80960 local bus. For details on inbound address translation, see Section 16.3, ATU ADDRESS TRANSLATION (pg. 16-4).
- Write data is transferred from the IDQ to the 80960 local bus when data is available and the local bus interface retains local bus ownership.
- The local bus interface stops transferring data to the local bus when one of the following conditions becomes true:
    - The local bus interface loses bus ownership and the IDQ still has data. In this case, the local bus interface removes REQ and immediately starts requesting the internal local bus again.
    - The Memory Controller signals a Bus Fault. In this case, the local bus interface aborts the inbound write transaction and clears the IAQ and IDQ.
    - The IDQ becomes empty while the transaction on the PCI bus is in progress, but held in wait states. In this case, the local bus interface goes idle and is requested again when data is received in the IDQ.
    - The IDQ becomes empty and the PCI transaction has completed. The IAQ is cleared, in this case, and the local bus interface goes idle. The IAQ and IDQ are now ready for a new transaction.

16.3.3 Inbound Read Transaction

An inbound read transaction is initiated by a PCI master (on either the primary or secondary PCI bus) and is targeted at either 80960 local memory or an 80960 local bus memory-mapped register. The read transaction is propagated through the inbound delayed read address queue (IDRAQ) and read data is returned through the outbound data queue (ODQ).

Data for all inbound ATU read transactions is implicitly prefetchable as defined in the *PCI Local Bus Specification*, revision 2.1. The Inbound ATU Base Address Register's Bit 3 is hardwired to one (1) defining the memory space as prefetchable. The ATU prefetches on both single- and multi-word read transactions.

All inbound read transactions are processed as delayed read transactions. The ATU's PCI interface claims the read transaction and forwards the read request through to the 80960 local bus and returns the read data to the PCI bus. The IDRAQ contains inbound PCI read address and the read data is stored in the ODQ. Data flow for an inbound read transaction on the PCI bus is summarized in the following statements:

- The ATU claims the PCI read transaction when the PCI address is within the inbound translation window defined by ATU Inbound Base Register and Inbound Limit Register.
- When no transaction is currently in the IDRAQ, the PCI address is latched into IDRAQ and a Retry is signalled to the initiator.
  - When the IDRAQ is full: the PCI address, command, and byte enables match those from a previous transaction, and the ODQ contains read data, start returning read data to the initiator.
  - When the IDRAQ is full and the PCI address, command, and byte enables do not match: signal a Retry to the initiator and do not latch any transaction information.
- Once read data is driven onto the PCI bus from the ODQ, it continues until one of the following is true:
  - The initiator completes the PCI transaction.
  - A local bus error was detected. In this case, a Target-abort is signaled to the initiator.
  - The ODQ becomes empty. In this case, the PCI interface signals a Disconnect to the initiator.

16.3.4 Inbound Configuration Cycle Translation

The ATU only accepts Type 0 configuration cycles with a function number of one (the bridge unit is function 0 in the i960 Rx I/O processor).

Both primary and secondary ATUs share the same PCI configuration space. This configuration space can be accessed using PCI configuration cycles from both the primary and secondary PCI buses using function 1 configuration space. All inbound configuration cycles are processed as delayed transactions.

16.3.5 Discard Timers

The ATUs implement discard timers for inbound delayed transactions. These timers prevent deadlocks when the initiator of a retried delayed transaction fails to complete the transaction within 2^{10} or 2^{15} PCI clock cycles. The timer starts counting when the delayed request becomes a delayed completion by completing on the destination bus. When the originating master on the initiating bus has not completed the transaction before the timer expires, the completion transaction is discarded.

Discard timer values are controlled by the Bridge Control Register’s Primary Discard Timer Value bit (for the primary ATU) and the Secondary Discard Timer Value bit (for the secondary ATU).

16.3.6 Outbound Address Translation

In addition to providing the mechanism for inbound translation, the ATUs translate i960 core processor-initiated cycles to the PCI bus. This is known as *outbound address translation*. Outbound transactions are processor reads or writes targeted at the PCI primary or secondary bus. The ATU local bus slave interface claims 80960 local bus address cycles and completes the cycle on the PCI bus on behalf the i960 core processor. The primary and secondary ATUs support two different outbound translation modes:

- Address Translation Windows
- Direct Addressing Window

Figure 16-5 shows a i960 Rx I/O processor memory map with all reserved address locations highlighted. The outbound translation windows exist from 8000 0000H to 9001 FFFFH. This is a 256 Mbyte window and a 128 Kbyte window which are equally divided between the primary and secondary ATUs. The outbound direct addressing window is from 0000 2000H to 7FFF FFFFH. Both outbound schemes are described in the following subsections.

Outbound address translation is disabled for the Primary ATU when the Bus Master Enable bit in the Primary ATU Command Register is clear and is disabled for the Secondary ATU when the Bus Master Enable bit in the Secondary ATU Command Register is clear. When the Bus Master Enable bit is clear or the Outbound ATU Enable (bits 1:2 of the ATUCR) are clear, the ATUs do not claim any i960 core processor accesses. These unclaimed accesses may cause a Bus Monitor time-out to occur. For outbound memory transactions, the only burst order supported is Linear Incrementing.

16.3.6.1 Outbound Address Translation Windows

Inbound translation involves a programmable inbound translation window consisting of a base and limit register and a value register for PCI to 80960 translation. The outbound address translation windows use a similar methodology except that the outbound translation windows are fixed in 80960 local bus address space; this removes the need for base and limit registers.

Figure 16-6 illustrates the outbound address translation windows. Each ATU has three windows. Two are 64 Mbyte and one is 64 Kbyte. The primary outbound memory and DAC translation windows range from 8000 0000H to 87FF FFFFH (2 x 64 Mbyte) and the secondary outbound memory and DAC translation windows range from 8800 0000H to 8FFF FFFFH (2 x 64 Mbyte). After these four windows, the primary and secondary outbound I/O windows range from 9000 0000H to 9001 FFFFH (2 x 64 Kbyte). When the secondary PCI Boot Mode (bit 11 in ATUCR) is set, the Secondary ATU claims all 80960 local bus accesses with addresses in the range FE00 0000 to FFFF FFFF.

Each memory and DAC window are 64 Mbytes and each I/O window is 64 Kbytes. An 80960 local bus cycle with an address within one outbound window initiates a read or write cycle on the targeted PCI bus. The PCI cycle type depends on which translation window the local bus cycle “hits”. The read or write decision is based on the 80960 local bus cycle type.

Each ATU has a window dedicated to the following outbound PCI transaction types in the outbound address translation window:

- Memory reads and writes - Memory Window
- I/O reads and writes - I/O Window
- Dual Address Cycle reads and writes - DAC Window

Refer to Figure 16-6 for the sub-window addresses involved in primary and secondary outbound translation.

The windowing scheme means:

- a processor read cycle that addresses a Memory Window is a Memory Read on the PCI bus
- a processor write cycle that addresses the I/O Window is an I/O Write on the PCI bus

Memory Write and Invalidate (MWI), Memory Read Line, and Memory Read Multiple commands are not supported in outbound ATU transactions.

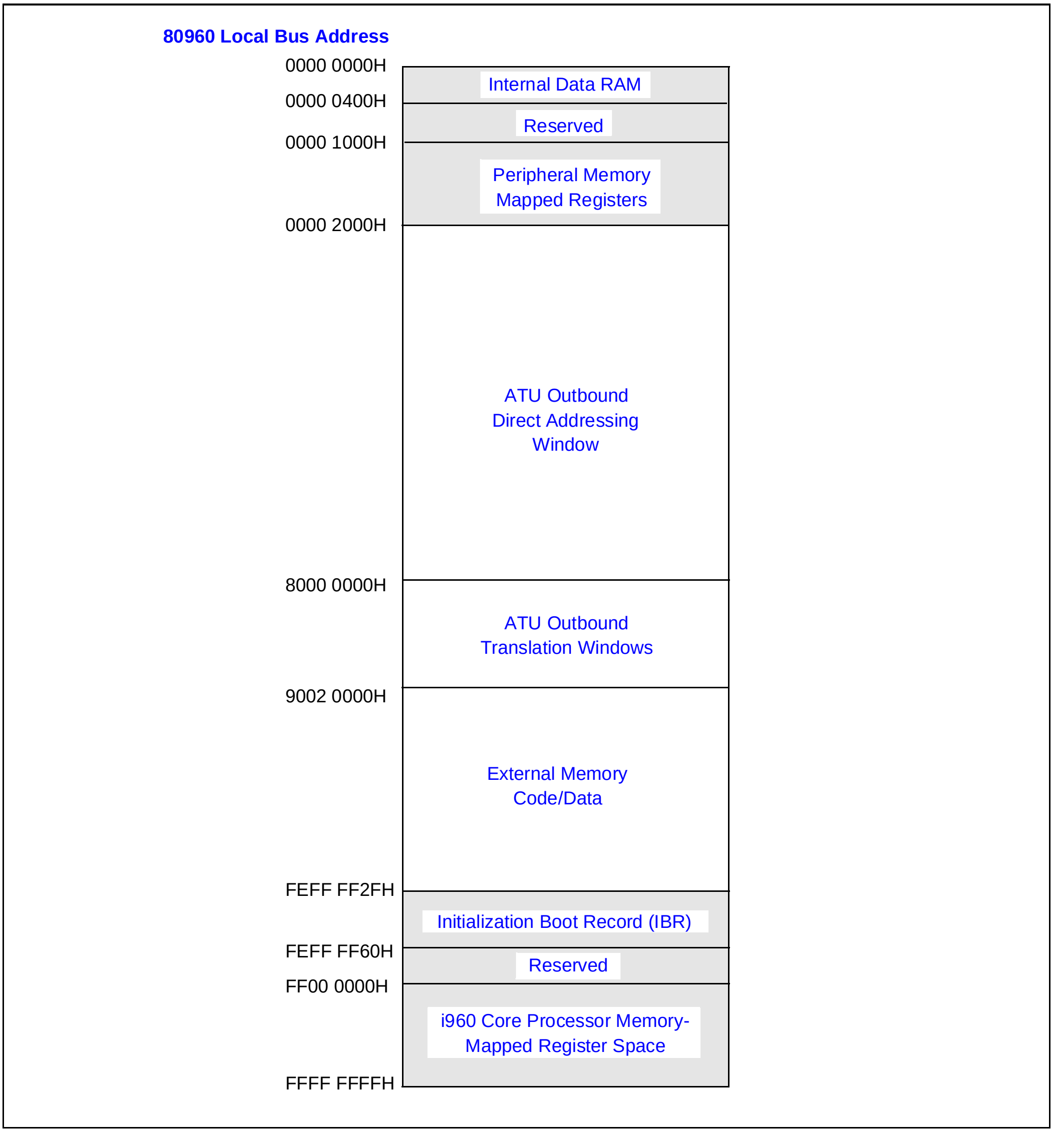

80960 Local Bus Address
0000 0000H
0000 0400H
0000 1000H
0000 2000H
8000 0000H
9002 0000H
FEFF FF2FH
FEFF FF60H
FF00 0000H
FFFF FFFFH
Internal Data RAM
Reserved
Peripheral Memory Mapped Registers
ATU Outbound Direct Addressing Window
ATU Outbound Translation Windows
External Memory Code/Data
Initialization Boot Record (IBR)
Reserved
i960 Core Processor Memory-Mapped Register Space

Figure 16-5. 80960 Local Bus Memory Map - Outbound Translation Window

The translation portion of outbound ATU transactions is accomplished with a value register in the same manner as inbound translations. The outbound upper 64-bit Dual Address Cycle (DAC) registers are for DAC commands and contain the high order 32-bits of a dual cycle 64-bit address directly with no translation. Both ATUs use the following registers in outbound address translation:

- Outbound Memory Window Value Register
- Outbound I/O Window Value Register
- Outbound DAC Window Value Register
- Outbound Upper 64-Bit DAC Register
- Outbound Configuration Cycle Address Register

See Section 16.7, ”REGISTER DEFINITIONS” (pg. 16-25) for details on outbound translation register definition and programming constraints.

The translation algorithm used, as stated, is very similar to inbound translation. For memory and DAC transactions, the algorithm is:

PCI_Address = (80960_Address & 03FF FFFFH) | Translate_Register

For memory and DAC transactions, the 80960 local bus address is bitwise ANDed with the inverse of 64 Mbytes which clears the upper 6 bits of address. The result is bitwise ORed with the outbound window value register to create the primary or secondary low 32-bit PCI address. The upper 32-bit of address for the PCI DAC read/write cycle comes from the Primary Outbound Upper 64-bit DAC Register and the Secondary Outbound Upper 64-bit DAC Register. For I/O transactions, the algorithm is:

PCI_Address = (80960_Address & 0000 FFFFH) | Translate_Register

For I/O transactions, the local address is bitwise ANDed with the inverse of 64 Kbytes which clears the upper 16 bits of address. Address aliasing can be prevented by programming the outbound window value registers on boundaries equivalent to the window’s length, but this is only enforced through application programming. PCI I/O addresses are byte addresses and not word addresses. The PCI I/O address’s two least significant bits are determined by byte enables that the processor issues. For example, when the i960 core processor performs a 2-byte write and generates byte enables of 0011_2, the ATU sets the two least significant bits of PCI I/O address to 10_2.

NOTE: When the i960 core processor’s data cache is enabled for accesses to the Outbound I/O Window, the byte enables generated by the i960 core processor are always 00_2 for Byte and Short accesses.

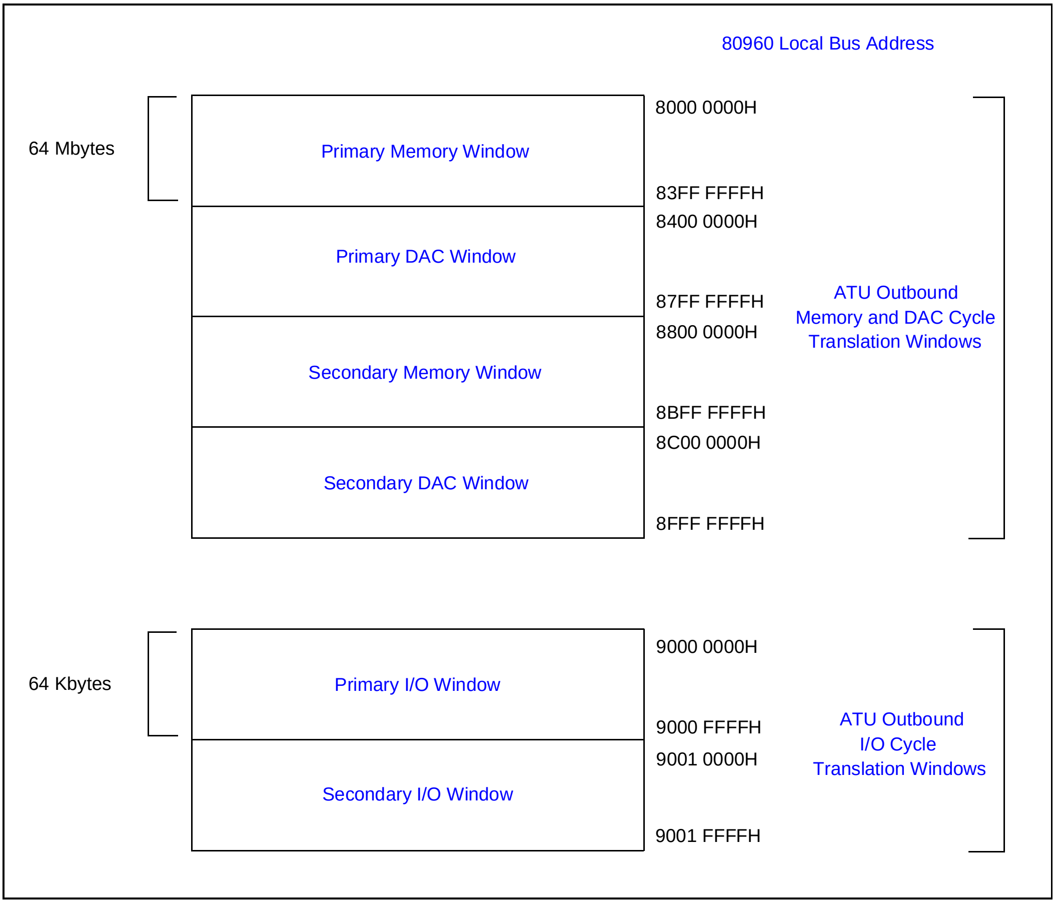


Figure 16-6. Outbound Address Translation Windows

16.3.6.2 Direct Addressing Window

The second method used by outbound cycles from the i960 core processor to the PCI bus is with the direct addressing window. This is a window of addresses in 80960 local bus address space that act in the same manner as the outbound translation windows without the translation. An i960 core processor read or write to a local bus address within the direct addressing window initiates a read or write on the PCI bus with the same address as used on the local bus. Figure 16-7 shows an example of an outbound write that is through the direct addressing window.

Direct Addressing is limited to PCI memory read and writes only. I/O cycles, DAC cycles, MWI, Memory Read Line, and Memory Read Multiple commands are not supported with direct addressing.

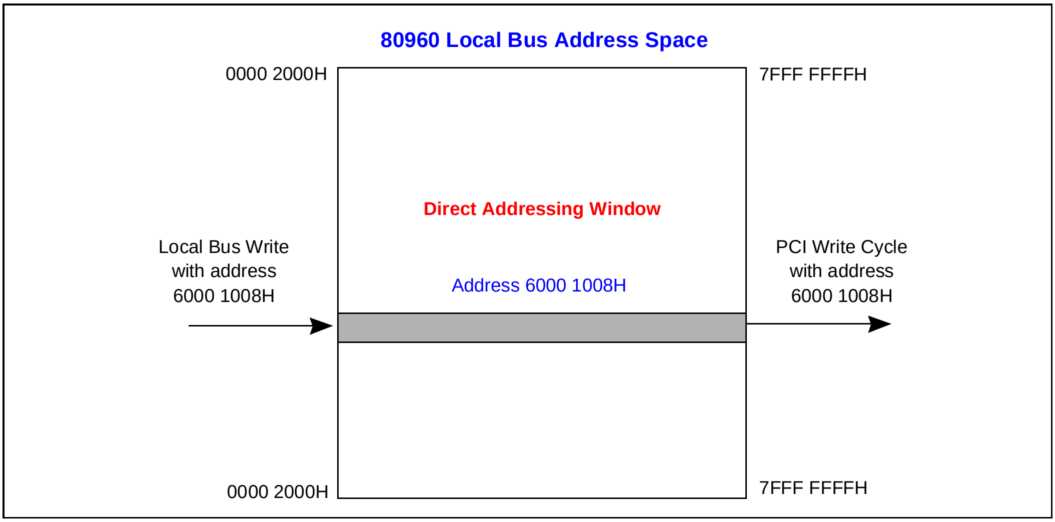


Figure 16-7. Direct Addressing Window

The direct addressing window address range is fixed in the lower 2 Gbytes of the 80960 local bus address space — except for the first 8 Kbytes which is reserved for the i960 core processor's internal data RAM and i960 core processor memory-mapped registers. 80960 local bus cycles with an address from 0000 2000H to 7FFF FFFCH are forwarded to a PCI bus, when enabled. The primary PCI bus is the default bus for direct addressing. The following bits within the Address Translation Unit Configuration Register (ATUCR) affect direct addressing operation:

- ATUCR Direct Addressing Enable bit - when set, enables the direct addressing window. When clear, addresses within the direct addressing window are not claimed by the ATU.
- ATUCR Secondary Direct Addressing Select bit - when clear, all transactions through the direct addressing window are to the primary ATU and primary PCI bus. When set, all transactions through the direct addressing window are to the secondary ATU and secondary PCI bus.

16.3.7 Outbound Write Transaction

An outbound write transaction is initiated by the i960 core processor and is targeted at a PCI slave on either the primary or secondary PCI buses. The outbound write address and write data are propagated from the 80960 local bus to a PCI bus through the OAQ and the ODQ.

The ATU's slave local bus interface claims the write transaction and forwards the write data through to the targeted PCI bus. Data flow for an outbound write transaction on the 80960 local bus is summarized in the following statements:

- The ATU local bus interface latches the address from the 80960 local bus into the OAQ when that address is inside one of the outbound translate windows and the OAQ and ODQ are empty.
- Once the outbound address is latched, the local bus interface stores the write data into the ODQ until the local bus transaction completes.
- When the OAQ or the ODQ are not available, the ATU signals the internal arbitration unit to assert an i960 core processor backoff. Backoff remains active until the OAQ and ODQ become available. When backoff is deasserted, the local bus slave interface returns to idle while the backoff logic re-initiates the local bus transaction.
- 80960Rx software must ensure that the upper 32 bits of a DAC are non-zero during outbound DAC window accesses.

16.3.8 Outbound Read Transaction

An outbound read transaction is initiated by the i960 core processor and is targeted at a PCI slave on either the primary or secondary PCI buses. The read transaction is propagated through the outbound address queue (OAQ) and read data is returned through the inbound data queue (IDQ).

The ATU’s local bus interface claims the read transaction and forwards the read request through to the PCI bus and returns the read data to the 80960 local bus. The data flow for an outbound read transaction on the local bus is summarized in the following statements:

- The ATU local bus interface latches the 80960 local bus address on the bus when the address is inside an outbound address translation window and the OAQ is empty. When the address is inside an outbound translation window but the OAQ is not empty (previous outbound transaction in progress), the local bus interface notifies the internal arbiter, which asserts backoff. The processor stays in backoff until the OAQ becomes empty, at which time backoff is deasserted.
- Once the outbound local address is latched into the OAQ, the i960 core processor is put into backoff to give the delayed read transaction time to complete on the PCI bus. Backoff is deasserted when the PCI interface has completed reading the requested amount of data and has put the data into the IDQ. A PCI error cancels backoff and causes the outbound read request to return FFFF FFFFH to the i960 core processor.
- If the PCI Read transaction is disconnected and an inbound write transaction occurs, then return any data to the local bus and allow the inbound write transaction to complete. The outbound read transaction will resume after the inbound write transaction completes.
- Once the transaction completes on the PCI bus, the local interface starts reading data from the IDQ. This continues until the IDQ is empty and the local bus operation completes.
- 80960Rx software must ensure that the upper 32 bits of a DAC are non-zero during outbound DAC window accesses.

16.3.9 Private PCI Address Space / Outbound Configuration Cycle Translation

The secondary ATU contains special support for creating private address spaces on the secondary PCI bus. A private address space is defined as a range of secondary PCI bus addresses which are not part of the secondary PCI address space as defined by the bridge and are also not part of the primary PCI address space. Private address space can be considered a “hole” in the PCI address space that is only supported on the secondary PCI bus. Private address space generally falls within the primary PCI address space and requires special bridge support so that it does not forward these addresses. The i960 Rx I/O processor has several mechanisms to support private address space:

- Inbound transactions from private devices to the secondary ATU.
- Outbound transactions from the secondary ATU and DMA channel 2 to private devices.
- Outbound configuration cycles to private devices.
- Hiding private devices from PCI Type 0 configuration cycles. (See CHAPTER 15, PCI-TO-PCI BRIDGE UNIT for more details.)

For inbound transactions from private devices, the secondary ATU can be configured outside the valid secondary PCI address space; this creates private address space. The secondary ATU claims private addresses and prevents the bridge from forwarding them upstream to the primary PCI bus.

For outbound transactions from the secondary ATU or DMA channel 2 to a private address space, the PCI to PCI bridge does not claim the transaction unless the ATUCR’s Secondary Bus Messaging Unit Access Enable bit (bit 12) is set. When this bit is set, the PCI-to-PCI Bridge unit can forward a transaction from the secondary PCI interface, through the bridge, and to the Messaging Unit (first 4 Kbytes of the PATU inbound address space) on the primary PCI interface. For correct operation, the transaction must be a valid bridge address (claimed by secondary interface of the bridge and forwarded to the primary interface of the bridge) as well as a valid Messaging Unit address. When clear, the Messaging Unit cannot claim a transaction mastered by the primary interface of the bridge.

Outbound configuration cycles — secondary and primary — can support private PCI devices. Outbound ATUs provide a port programming model for outbound configuration cycles. Performing an outbound configuration cycle to either the primary or secondary PCI bus involves up to two 80960 local bus cycles:

1) Writing the Outbound Configuration Cycle Address Register (primary or secondary) with the PCI address used during the configuration cycle. See the *PCI Local Bus Specification*, revision 2.1 for information regarding configuration address cycle formats. This i960 core processor cycle enables the transaction.

2) Writing or reading the Outbound Configuration Cycle Data Register (primary or secondary). The i960 core processor cycle initiates the transaction. A read causes a configuration cycle read to the primary or secondary PCI bus with the address in the outbound configuration cycle address register. Similarly, a write initiates a configuration cycle write to the primary or secondary PCI bus with the write data from the second processor cycle. Configuration cycles are non-burst and restricted to a single word cycle.

Section 16.7, REGISTER DEFINITIONS (pg. 16-25) describes the outbound configuration cycle address and data register definitions and programming constraints.

NOTE: Outbound configuration cycle data registers are not physical registers. They are an 80960 local bus memory mapped address used to initiate a transaction with the address in the associated address register. Reads/writes to these registers return data from the PCI bus — not from the register. Outbound configuration cycles use address stepping and may delay the assertion of FRAME#.

16.4 MESSAGING UNIT

The Messaging Unit (MU) transfers data between the PCI system and the i960 Rx I/O processor and notifies the respective system when new data arrives. The MU is described in CHAPTER 17, MESSAGING UNIT.

The primary PCI window for messaging transactions is always the *first* 4 Kbytes of the inbound translation window defined by the Primary Inbound ATU Base Address Register (PIABAR) and the Primary Inbound ATU Limit Register (PIALR).

16.5 EXPANSION ROM TRANSLATION UNIT

The primary inbound ATU supports one address range (defined by a base/limit register pair) used for containing the Expansion ROM. Refer to the *PCI Local Bus Specification*, revision 2.1 for details on Expansion ROM format and usage.

During a powerup sequence, initialization code from Expansion ROM is executed once by the host processor to initialize the associated device. The code can be discarded once executed. Expansion ROM registers are described in Section 16.7.15 (pg. 16-41), Section 16.7.31 (pg. 16-57), and Section 16.7.32 (pg. 16-58).

The inbound primary ATU supports an inbound Expansion ROM window which works like the inbound translation window. A read from the expansion ROM window is forwarded to the 80960 local bus and to the Memory Controller. Writes through the Expansion ROM window are not supported. The address translation algorithm is the same as in inbound translation; see Section 16.3.1, Inbound Address Translation (pg. 16-5). Two ROM widths are supported: 8- and 32-bit. When the Expansion ROM is 32 bits wide (Expansion ROM Width bit is set in the ATUCR), the inbound ATU uses standard 32-bit accesses on the local bus as if it was reading from any 32-bit memory.

16.6 ATU DATA FLOW ERROR CONDITIONS

PCI and 80960 local bus error conditions cause the ATU state machines to exit normal operation and return to idle states. Error conditions on one side of the ATU are propagated to the other side of the ATU and have different effects depending on the error. Error conditions and their effects are described in the following sections.

PCI bus error conditions and the action taken on the bus are defined within the *PCI Local Bus Specification*, revision 2.1. The ATU adheres to the error conditions defined within the PCI specification for both master and slave operation. Error conditions on the 80960 local bus are caused by the propagation of an error from the Memory Controller. See CHAPTER 14, MEMORY CONTROLLER for details on memory controller error conditions. All actions on the PCI Bus for error situations are dependent on the error control bits found in the Primary ATU and Secondary ATU Command Registers. See Section 16.7, REGISTER DEFINITIONS (pg. 16-25).

Table 16-2 through Table 16-5 assume that all error reporting is enabled through the appropriate command and status registers (unless otherwise noted). Refer to the *PCI Local Bus Specification*, revision 2.1 for details on the complete action a PCI master and slave interface needs to take for parity error events.

When the ATU detects the assertion of P_SERR# on the primary PCI bus and the Primary SERR Interrupt Enable bit in the ATU Configuration Register (ATUCR) is set, the ATU signals an NMI# interrupt to the i960 core processor. Likewise, when the ATU detects the assertion of the S_SERR# signal on the secondary PCI bus and the Secondary SERR Interrupt Enable bit in the ATUCR is set, the ATU signals an NMI# interrupt to the i960 core processor.

Table 16-2. Inbound Write Error Conditions

Bus & State Machine	Error Condition	Effect on PCI Bus	Effect on 80960 Local Bus
PCI Slave	Address Parity Error	• SERR# asserted • PCI Master Abort	• No effect • Transaction never propagated to local bus
	Data Parity Error	• PERR# asserted • IAQ Cleared • PCI Disconnect	• Data in IDQ completed
Local Bus Master	80960Rx Memory Controller Fault	• PERR# asserted when transaction is still in progress or... SERR# asserted after transaction completes on PCI bus, if not in progress • IAQ cleared	• i960 core processor is interrupted with NMI# • IDQ cleared

Table 16-3. Inbound Read Error Conditions

Bus & State Machine	Error Condition	Effect on PCI Bus	Effect on 80960 Local Bus
PCI Slave	Address Parity Error	• SERR# asserted • PCI Master Abort	• No effect • Transaction never propagated to local bus
Local Bus Master	80960Rx Memory Controller Parity Error	• ATU interface drives bad data, causes bad parity • Error condition determined by PCI master	• i960 core processor is interrupted with NMI#
	80960Rx Memory Controller Fault	• PCI Target Abort	• i960 core processor is interrupted with NMI#

Table 16-4. Outbound Write Error Conditions

Bus & State Machine	Error Condition	Effect on PCI Bus	Effect on 80960 Local Bus
PCI Master	No DEVSEL#	• PCI Master Abort	• i960 core processor is interrupted with NMI# if the ATU PCI Error Interrupt Enable bit is set in the ATUCR. The data in the OWQ is discarded.
	Data Parity Error	• PERR# detected	
	PCI Target Abort	• PCI Target Abort	

Table 16-5. Outbound Read Error Conditions

Bus & State Machine	Error Condition	Effect on PCI Bus	Effect on 80960 Local Bus
PCI Master	No DEVSEL#	• PCI Master Abort	• i960 core processor is interrupted with NMI# if the ATU PCI Error Interrupt Enable bit is set in the ATUCR • A false data value is returned to the processor to allow the cycle to complete. FFH is returned for every byte read on the local bus
	Data Parity Error	• PERR# asserted	
	PCI Target Abort	• PCI Target Abort	

The following two tables (Table 16-6 and Table 16-7) summarize the ATU error reporting for PCI bus errors and local bus errors. The tables assume that all error reporting is enabled through the appropriate command and status registers (unless otherwise noted). The Primary and Secondary ATU Status Registers record PCI bus errors. Note that the SERR# Asserted bit in the Status Register is set only when the SERR# Enable bit in the Command Register is set. The Primary and Secondary ATU Interrupt Status Registers record i960 core processor interrupt status information.

Table 16-6. Primary ATU Error Reporting Summary (Sheet 1 of 2)

Error Condition	Primary ATU Status Register (PATUSR)	Primary ATU Interrupt Status Register (PATUISR)	NMI# Interrupt? (if enabled)
Inbound Write PCI Address Parity Error	Parity Error bit (bit 15) set P_SERR# Asserted bit (bit 14) set	P_SERR# Detected bit (bit 4) set	Yes
Inbound Write PCI Data Parity Error	Parity Error bit (bit 15) set		No
Inbound Write Local Bus Fault	P_SERR# Asserted bit (bit 14) set	P_SERR# Detected bit (bit 4) set 80960 local bus address Fault (bit 5) set	Yes
Inbound Read PCI Address Parity Error	Parity Error bit (bit 15) set P_SERR# Asserted bit (bit 14) set	P_SERR# Detected bit (bit 4) set	Yes
Inbound Read Local Bus Data Parity Error		80960 local bus memory Fault bit (bit 6) set	Yes
Inbound Read Local Bus Fault	Target Abort (Target) (bit 11) set	80960 local bus address Fault (bit 5) set	Yes
Outbound Write PCI Master Abort	Master Abort bit (bit 13) set	PCI Master Abort bit (bit 3) set	Yes

Table 16-6. Primary ATU Error Reporting Summary (Sheet 2 of 2)

Error Condition	Primary ATU Status Register (PATUSR)	Primary ATU Interrupt Status Register (PATUISR)	NMI# Interrupt? (if enabled)
Outbound Write PCI Data Parity Error	Master Parity Error (bit 8) set, Parity Error (bit 15) is set	PCI Master Parity Error bit (bit 0) set	Yes
Outbound Write PCI Target Abort	Target Abort (Master) (bit 12) set	PCI Target Abort (Master) (bit 2) set	Yes
Outbound Read PCI Master Abort	Master Abort bit (bit 13) set	PCI Master Abort bit (bit3) set	Yes
Outbound Read PCI Data Parity Error	Parity Error bit (bit 15) set Master Parity Error (bit 8) set	PCI Master Parity Error (bit 0) set	Yes
Outbound Read PCI Target Abort	Target Abort (Master) (bit 12) set	PCI Target Abort (Master) (bit 2) set	Yes
P_SERR# Detected		P_SERR# Detected bit (bit 4) set	Yes

Table 16-7. Secondary ATU Error Reporting Summary (Sheet 1 of 2)

Error Condition	Secondary ATU Status Register (SATUSR)	Secondary ATU Interrupt Status Register (SATUISR)	NMI# Interrupt? (if enabled)
Inbound Write PCI Address Parity Error	Parity Error bit (bit 15) set S_SERR# Asserted bit (bit 14) set	S_SERR# Detected bit (bit 4) set	Yes
Inbound Write PCI Data Parity Error	Parity Error bit (bit 15) set		No
Inbound Write Local Bus Fault	S_SERR# Asserted bit (bit 14) set	S_SERR# Detected bit (bit 4) set 80960 local bus address Fault (bit 5) set	Yes
Inbound Read PCI Address Parity Error	Parity Error bit (bit 15) set S_SERR# Asserted bit (bit 14) set	S_SERR# Detected bit (bit 4) set	Yes
Inbound Read Local Bus Data Parity Error		80960 local bus memory Fault bit (bit 6) set	Yes
Inbound Read Local Bus Fault	Target Abort (Target) (bit 11) set	80960 local bus address Fault (bit 5) set	Yes
Outbound Write PCI Master Abort	Master Abort bit (bit 13) set	PCI Master Abort bit (bit 3) set	Yes

Table 16-7. Secondary ATU Error Reporting Summary (Sheet 2 of 2)

Error Condition	Secondary ATU Status Register (SATUSR)	Secondary ATU Interrupt Status Register (SATUISR)	NMI# Interrupt? (if enabled)
Outbound Write PCI Data Parity Error	Master Parity Error (bit 8) set, Parity Error (bit 15) is set	PCI Master Parity Error bit (bit 0) set	Yes
Outbound Write PCI Target Abort	Target Abort (Master) (bit 12) set	PCI Target Abort (Master) (bit 2) set	Yes
Outbound Read PCI Master Abort	Master Abort bit (bit 13) set	PCI Master Abort bit (bit3) set	Yes
Outbound Read PCI Data Parity Error	Parity Error bit (bit 15) set Master Parity Error (bit 8) set	PCI Master Parity Error (bit 0) set	Yes
Outbound Read PCI Target Abort	Target Abort (Master) (bit 12) set	PCI Target Abort (Master) (bit 2) set	Yes
S_SERR# Detected		S_SERR# Detected bit (bit 4) set	Yes

16.7 REGISTER DEFINITIONS

Every PCI device implements its own separate configuration address space and configuration registers. The *PCI Local Bus Specification*, revision 2.1 requires that configuration space be 256 bytes, and the first 64 bytes must adhere to a predefined header format.

Figure 16-8 defines the format for the first 64 bytes of the header. The additional 182 bytes of the configuration space is defined as the ATU extended configuration space. ATU configuration space is function number one of the i960 Rx I/O processor multifunction PCI device.

Beyond the required 64 byte header format, ATU configuration space implements extended register space in support of the units functionality. Refer to the *PCI Local Bus Specification*, revision 2.1 for details on accessing and programming configuration register space.

The following sections describe the ATU and Expansion ROM configuration registers. Configuration space consists of 8, 16, 24, and 32-bit registers arranged in a predefined format. Each register is described in functionality, access type (read/write, read/clear, read only) and reset default condition.

See CHAPTER 1, INTRODUCTION for a description of *reserved*, *read only*, and *read/clear*. All registers adhere to the definitions found in the *PCI Local Bus Specification*, revision 2.1 unless otherwise noted.

NOTE: Each configuration register's access type is individually defined for PCI configuration accesses. Some PCI read-only configuration registers have read/write capability from the i960 core processor. See also APPENDIX C, MEMORY-MAPPED REGISTERS.

<table>
<tr><th colspan="4">ATU Configuration Space Header</th><th>PCI Config Addr Offset</th></tr>
<tr><td colspan="2">ATU Device ID</td><td colspan="2">ATU Vendor ID</td><td>00H</td></tr>
<tr><td colspan="2">Primary ATU Status</td><td colspan="2">Primary ATU Command</td><td>04H</td></tr>
<tr><td colspan="3">ATU Class Code</td><td>ATU Revision ID</td><td>08H</td></tr>
<tr><td>ATU BIST</td><td>ATU Header Type</td><td>ATU Latency Timer</td><td>ATU Cacheline Size</td><td>0CH</td></tr>
<tr><td colspan="4">Primary Inbound ATU Base Address</td><td>10H</td></tr>
<tr><td colspan="4" rowspan="2"></td><td>14H</td></tr>
<tr><td>18H</td></tr>
<tr><td rowspan="4"></td><td colspan="2" rowspan="2">Reserved</td><td rowspan="4"></td><td>1CH</td></tr>
<tr><td>20H</td></tr>
<tr><td colspan="2" rowspan="2"></td><td>24H</td></tr>
<tr><td>28H</td></tr>
<tr><td colspan="2">ATU Subsystem ID</td><td colspan="2">ATU Subsystem Vendor ID</td><td>2CH</td></tr>
<tr><td colspan="4">Expansion ROM Base Address</td><td>30H</td></tr>
<tr><td colspan="4"></td><td>34H</td></tr>
<tr><td></td><td colspan="2">Reserved</td><td></td><td>38H</td></tr>
<tr><td>ATU Max. Latency</td><td>ATU Minimum Grant</td><td>ATU Interrupt Pin</td><td>ATU Interrupt Line</td><td>3CH</td></tr>
</table>

Figure 16-8. ATU Configuration Space Header

Table 16-8. ATU Configuration Space Register Summary (Sheet 1 of 3)

Section	Register Name and Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
16.7.1	ATU Vendor ID Register - ATUVID	16-29	16	0000 1200H	00H
16.7.2	ATU Device ID Register - ATUDID	16-29	16	0000 1202H	02H
16.7.3	Primary ATU Command Register - PATUCMD	16-30	16	0000 1204H	04H
16.7.4	Primary ATU Status Register - PATUSR	16-31	16	0000 1206H	06H
16.7.5	ATU Revision ID Register - ATURID	16-32	8	0000 1208H	08H
16.7.6	ATU Class Code Register - ATUCCR	16-32	24	0000 1209H	09H
16.7.7	ATU Cacheline Size Register - ATUCLSR	16-33	8	0000 120CH	0CH
16.7.8	ATU Latency Timer Register - ATULT	16-33	8	0000 120DH	0DH
16.7.9	ATU Header Type Register - ATUHTR	16-34	8	0000 120EH	0EH
16.7.10	ATU BIST Register - ATUBISTR	16-35	8	0000 120FH	0FH

Table 16-8. ATU Configuration Space Register Summary (Sheet 2 of 3)

Section	Register Name and Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
16.7.11	Primary Inbound ATU Base Address Register - PIABAR	16-36	32	0000 1210H	10H
	Reserved		32	0000 1214H	14H
			32	0000 1218H	18H
			32	0000 121CH	1CH
			32	0000 1220H	20H
			32	0000 1224H	24H
			32	0000 1228H	28H
16.7.13	ATU Subsystem Vendor ID Register - ASVIR	16-40	16	0000 122CH	2CH
16.7.14	ATU Subsystem ID Register - ASIR	16-40	16	0000 122EH	2EH
16.7.15	Expansion ROM Base Address Register - ERBAR	16-41	32	0000 1230H	30H
	Reserved		32	0000 1234H	34H
			32	0000 1238H	38H
16.7.16	ATU Interrupt Line Register - ATUILR	16-42	8	0000 123CH	3CH
16.7.17	ATU Interrupt Pin Register - ATUIPR	16-43	8	0000 123DH	3DH
16.7.18	ATU Minimum Grant Register - ATUMGNT	16-44	8	0000 123EH	3EH
16.7.19	ATU Maximum Latency Register - ATUMLAT	16-45	8	0000 123FH	3FH
16.7.20	Primary Inbound ATU Limit Register - PIALR	16-46	32	0000 1240H	40H
16.7.21	Primary Inbound ATU Translate Value Register - PIATVR	16-47	32	0000 1244H	44H
16.7.22	Secondary Inbound ATU Base Address Register - SIABAR	16-48	32	0000 1248H	48H
16.7.23	Secondary Inbound ATU Limit Register - SIALR	16-49	32	0000 124CH	4CH
16.7.24	Secondary Inbound ATU Translate Value Register - SIATVR	16-50	32	0000 1250H	50H
16.7.25	Primary Outbound Memory Window Value Register - POMWVR	16-51	32	0000 1254H	54H
	Reserved		32	0000 1258H	58H
16.7.26	Primary Outbound I/O Window Value Register - POIOWVR	16-52	32	0000 125CH	5CH
16.7.27	Primary Outbound DAC Window Value Register - PODWVR	16-53	32	0000 1260H	60H
16.7.28	Primary Outbound Upper 64-bit DAC Register - POUDR	16-54	32	0000 1264H	64H
16.7.29	Secondary Outbound Memory Window Value Register - SOMWVR	16-55	32	0000 1268H	68H
16.7.30	Secondary Outbound I/O Window Value Register - SOIOWVR	16-56	32	0000 126CH	6CH
	Reserved		32	0000 1270H	70H
16.7.31	Expansion ROM Limit Register - ERLR	16-57	32	0000 1274H	74H
16.7.32	Expansion ROM Translate Value Register - ERTVR	16-58	32	0000 1278H	78H
	Reserved		32	0000 127CH	7CH
			32	0000 1280H	80H
			32	0000 1284H	84H

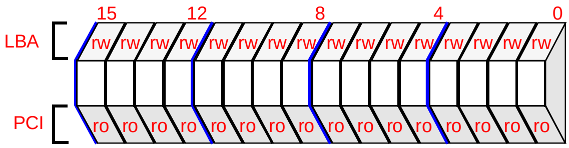
Table 16-8. ATU Configuration Space Register Summary (Sheet 3 of 3)

Section	Register Name and Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
16.7.33	ATU Configuration Register - ATUCR	16-58	32	0000 1288H	88H
	Reserved		32	0000 128CH	8CH
16.7.34	Primary ATU Interrupt Status Register - PATUISR	16-61	32	0000 1290H	90H
16.7.35	Secondary ATU Interrupt Status Register - SATUISR	16-62	32	0000 1294H	94H
16.7.36	Secondary ATU Command Register - SATUCMD	16-64	16	0000 1298H	98H
16.7.37	Secondary ATU Status Register - SATUSR	16-65	16	0000 129AH	9AH
16.7.38	Secondary Outbound DAC Window Value Register - SODWVR	16-66	32	0000 129CH	9CH
16.7.39	Secondary Outbound Upper 64-bit DAC Register - SOUDR	16-67	32	0000 12A0H	A0H
16.7.40	Primary Outbound Configuration Cycle Address Register - POCCAR	16-68	32	0000 12A4H	A4H
16.7.41	Secondary Outbound Configuration Cycle Address Register - SOCCAR	16-69	32	0000 12A8H	A8H
16.7.42	Primary Outbound Configuration Cycle Data Port - POCCDP	16-70	32	0000 12ACH	ACH
16.7.43	Secondary Outbound Configuration Cycle Data Port - SOCCDP	16-70	32	0000 12B0H	B0H
	Reserved			0000 12B4H through 0000 12FFH	

16.7.1 ATU Vendor ID Register - ATUVID

The ATU Vendor ID Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1.

Table 16-9. ATU Vendor ID Register - ATUVID


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

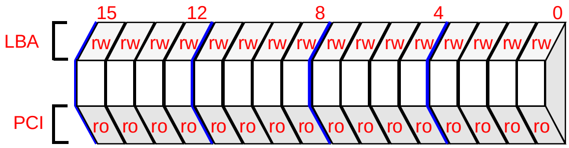
LBA: 1200H
PCI: 00H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15:00	8086H	ATU Vendor ID - This is a 16-bit value assigned to Intel. This register, combined with the DID, uniquely identify the PCI device. Access type is Read/Write to allow the i960 core processor to configure the register as a different vendor ID to simulate the interface of a standard mechanism currently used by existing application software.

16.7.2 ATU Device ID Register - ATUDID

ATU Device ID Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1.

Table 16-10. ATU Device ID Register - ATUDID


15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

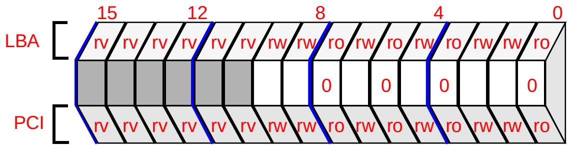
LBA: 1202H
PCI: 02H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15:00	1960H	ATU Device ID - This is a 16-bit value assigned to the ATU and MU. This ID, combined with the ATUVID, uniquely identify the PCI device.

16.7.3 Primary ATU Command Register - PATUCMD

ATU Command Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1 and, in most cases, affect the behavior of the primary ATU.

Table 16-11. Primary ATU Command Register - PATUCMD


15
12
8
4
0
LBA
rv rv rv rv rv rv rw rw ro rw ro rw ro rw rw ro
0 0 0 0
PCI
rv rv rv rv rv rv rw rw ro rw ro rw ro rw rw ro

LBA: 1204H
PCI: 04H

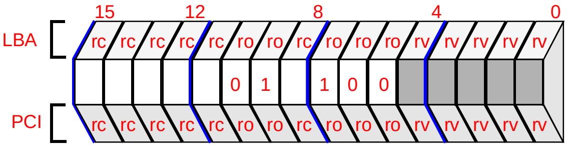
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:10	00H	Reserved.
09	0_2	Fast Back to Back Enable - When cleared, the ATU primary interface is not allowed to generate fast back-to-back cycles on its bus.
08	0_2	P_SERR# Enable - When cleared, the ATU primary interface is not allowed to assert P_SERR# on the PCI interface.
07	0_2	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field.
06	0_2	Parity Checking Enable - When set, the primary ATU and DMA channels 0 and 1 take normal action when a parity error is detected. When cleared, parity checking is disabled.
05	0_2	VGA Palette Snoop Enable - The primary ATU interface does not support I/O writes and therefore, does not perform VGA pallet snooping.
04	0_2	Memory Write and Invalidate Enable - When set, DMA channels 0 and 1 may generate MWI commands. When clear, DMA channels 0 and 1 use Memory Write commands instead of MWI.
03	0_2	Special Cycle Enable - The ATU interface does not respond to special cycle commands in any way. Not applicable. Not implemented and a reserved bit field
02	0_2	Bus Master Enable - The primary ATU interface can act as a master on the PCI bus. When cleared, disables the primary ATU from generating PCI accesses. When set, allows the primary ATU to behave as a PCI bus master. This enable bit also controls DMA channels 0 and 1 master interface. The bit must be set before initiating a DMA transfer on the PCI bus.
01	0_2	Memory Enable - Controls the primary ATU interface’s response to PCI memory addresses. When cleared, the ATU interface does not respond to any memory access on the PCI bus.
00	0_2	I/O Space Enable - Controls the ATU interface response to I/O transactions on the primary side. The primary ATU does not support I/O space.

16.7.4 Primary ATU Status Register - PATUSR

The Primary ATU Status Register bits adhere to the *PCI Local Bus Specification*, revision 2.1 definitions. The *read/clear* bits can only be set by internal hardware and are cleared by either a reset condition or by writing a 1_2 to the bit to be cleared.

Table 16-12. Primary ATU Status Register - PATUSR

 15 12 8 4 0 LBA rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv 0 1 1 0 0 PCI rc rc rc rc rc ro ro rc ro ro ro rv rv rv rv rv		

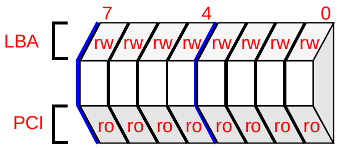
LBA: 1206H
PCI: 06H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
15	0_2	Parity Error - set when a parity error is detected on the primary PCI bus even when the PATUCMD register's Parity Checking Enable bit is cleared.
14	0_2	P_SERR# Asserted - set when P_SERR# is asserted on the PCI bus.
13	0_2	Master Abort - set when a transaction initiated by the primary ATU master interface ends in a Master-abort.
12	0_2	Target Abort (master) - set when a transaction initiated by the primary ATU master interface ends in a target abort.
11	0_2	Target Abort (target) - set when the primary ATU interface, acting as a target, terminates the transaction on the primary PCI bus with a target abort.
10:09	01_2	DEVSEL# Timing - These bits are read-only and define medium DEVSEL# timing for a target device (except configuration accesses).
08	0_2	Master Parity Error - The primary ATU interface sets this bit when three conditions are met: 1) bus agent asserted S_PERR# itself or observed S_PERR# asserted 2) agent setting the bit acted as the bus master for the operation in which the error occurred 3) PATUCMD register's Parity Checking Enable bit is set
07	1_2	Fast Back-to-Back - The ATU/Messaging Unit interface is capable of accepting fast back-to-back transactions when the transactions are not to the same target.
06	0_2	UDF Supported - User Definable Features are not supported.
05	0_2	66 MHz Capable - 66 MHz operation is not supported.
04:00	00H	Reserved.

16.7.5 ATU Revision ID Register - ATURID

Revision ID Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1.

Table 16-13. ATU Revision ID Register - ATURID



LBA: 1208H
PCI: 08H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

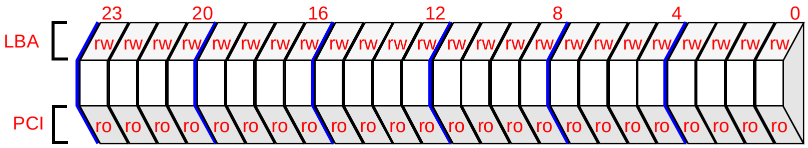
Bit	**Default**	**Description**
07:00	XXH	ATU Revision - Identifies the i960 Rx I/O processor's revision number.[1]

[1] These numbers vary with stepping, refer to the *i960® Rx I/O Processor Specification Update (272918)* for the correct value.

16.7.6 ATU Class Code Register - ATUCCR

Class Code Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. Auto configuration software reads this register to determine the PCI device function.

Table 16-14. ATU Class Code Register - ATUCCR


23
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

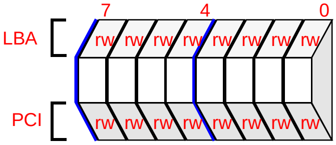
LBA: 1209H
PCI: 09H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
23:16	05H	Base Class - Memory Controller
15:08	80H	Sub Class - Other Memory Controller
07:00	00H	Programming Interface - None defined

16.7.7 ATU Cacheline Size Register - ATUCLSR

Cacheline Size Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register is programmed with the system cacheline size in DWORDs (32-bit words). Cacheline Size is restricted to either 8 or 16 DWORDs; the ATU interprets any other value as “0”.

Table 16-15. ATU Cacheline Size Register - ATUCLSR



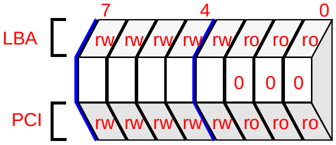
LBA: 120CH
PCI: 0CH
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write
			PCI = PCI Configuration Address Offset

Bit	Default	Description
07:00	00H	ATU Cacheline Size - specifies the system cacheline size in DWORDs. Cacheline size is restricted to either 8 or 16 DWORDs.

16.7.8 ATU Latency Timer Register - ATULT

ATU Latency Timer Register bit definitions apply to both the primary and secondary PCI interfaces.

Table 16-16. ATU Latency Timer Register - ATULT



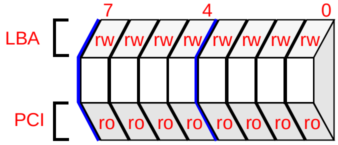
LBA: 120DH
PCI: 0DH
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write
			PCI = PCI Configuration Address Offset

Bit	Default	Description
07:03	0H	Programmable Latency Timer - This field varies the latency timer for the primary interface from 0 to 248 clocks, in increments of eight clocks.
02:00	000_2	Latency Timer Granularity - These Bits are read only giving a programmable granularity of 8 clocks for the latency timer.

16.7.9 ATU Header Type Register - ATUHTR

Header Type Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register indicates the layout of ATU and Messaging Unit register configuration space bytes 10H to 3FH. The MSB indicates whether or not the device is multifunction.

Table 16-17. ATU Header Type Register - ATUHTR

 7 4 0 LBA rw rw rw rw rw rw rw rw PCI ro ro ro ro ro ro ro ro		

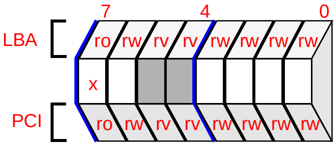
LBA: 120EH
PCI: 0EH
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
07	1_2	Single Function/Multi-Function Device - Identifies the ATU is a multifunction PCI device. 1=multifunction device.
06:00	0H	PCI Header Type - This bit field indicates the type of PCI header implemented. The ATU interface header conforms to *PCI Local Bus Specification*, revision 2.1. Type 00H configuration space header definition.

16.7.10 ATU BIST Register - ATUBISTR

The ATU BIST Register controls the functions the i960 core processor performs when BIST is initiated. This register is the interface between the host processor requesting BIST functions and the i960 core processor replying with the results from the software implementation of BIST functionality.

Table 16-18. ATU BIST Register - ATUBISTR

 7 4 0 LBA ro rw rv rv rw rw rw rw x PCI ro rw rv rv rw rw rw rw		

LBA: 120FH
PCI: 0FH
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
07	X_2	BIST Capable - This bit value is always equal to the ATUCR ATU BIST Interrupt Enable bit. See Section 16.7.33, ATU Configuration Register - ATUCR
06	0_2	Start BIST - When the ATUCR BIST Interrupt Enable bit is set: • Setting this bit generates an interrupt to the i960 core processor to perform a software BIST function. The i960 core processor clears this bit when the BIST software has completed with the BIST results found in ATUBISTR register bits [3:0]. When the ATUCR BIST Interrupt Enable bit is clear: • Setting this bit does not generate an interrupt to the i960 core processor and no BIST functions are performed. The i960 core processor does not clear this bit.
05:04	00_2	Reserved.
03:00	0H	BIST Completion Code - when the ATUCR BIST Interrupt Enable bit is set and the ATUBISTR Start BIST bit is set (bit 6): • The i960 core processor places the results of the software BIST in these bits. A nonzero value indicates a device-specific error.

16

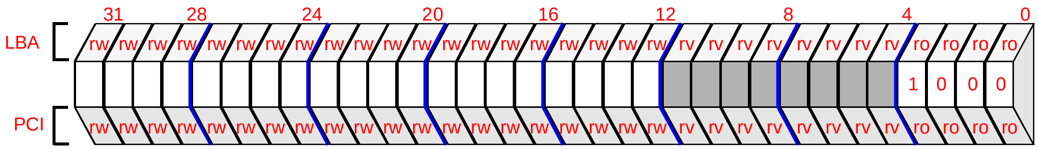
16.7.11 Primary Inbound ATU Base Address Register - PIABAR

The Primary Inbound ATU Base Address Register (PIABAR) defines the block of memory addresses where the primary inbound translation window begins. The inbound ATU decodes and forwards the bus request to the 80960 local bus with a translated address to map into 80960 local memory. The PIABAR defines the base address and describes the required memory block size; see section 16.7.12.

The first 4 Kbytes of memory defined by the PIABAR and the PIALAR is reserved for the Messaging Unit.

The programmed value within the base address register must comply with the PCI programming requirements for address alignment. Refer to the *PCI Local Bus Specification*, revision 2.1 for additional information on programming base address registers.

Table 16-19. Primary Inbound ATU Base Address Register - PIABAR



LBA: 1210H
PCI: 10H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Primary Translation Base Address - These bits define the actual location the Primary translation function is to respond to when addressed from the PCI bus.
11:04	00H	Reserved.
03	1_2	Prefetchable Indicator - Defines the memory spaces as prefetchable.
02:01	00_2	Address Type - These bits define where the block of memory can be located. The base address must be located anywhere in the first 4 Gbyte of address space (lower 32 bits of address).
00	0_2	Memory Space Indicator - This bit field describes memory or I/O space base address. the primary ATU does not occupy I/O space, thus this bit must be zero.

16.7.12 Determining Block Sizes for Base Address Registers

The Primary and Secondary Inbound ATU Base Address Registers and Expansion ROM Base Address Register use their associated limit registers for defining the defining the requested address space size. The requested address size and type can be determined by writing to a base address register, and then reading back from the register. Table 16-20 describes the device specific values used to determine the memory block size. By scanning the returned value from the least-significant bit of the base address register in ascending order, the programmer can determine the required address space size. The binary-weighted value of the first one bit found indicates the required amount of space. Table 16-20 describes the relationship between the values read back and the byte sizes the base address register requires.

Table 16-20. Device Specific Instructions for Base Address Register (Sheet 1 of 2)

Device Part Number	Value Written to the BAR	Effect of writing the value to the Base Address Register
80960RP 33/5.0 A-0	FFFF FFFFH	The first read after a write of FFFF FFFFH to the base address register is directed to the limit register. The data returned on subsequent reads from the base address register is the contents of the base address register — not the contents of the corresponding limit register. If any other value is written to the base address register, that value is programmed into the base address register. When determining block size requirements, reading to or writing from the base address register must be using 32-bit configuration cycles. However, configuration cycles not used to determine block size requirements can be performed as 8-, 16-, or 32-bit accesses.
80960RP 33/5.0 A-1	FFFF FFFEH or FFFF FFFFH	The first read after a write of FFFF FFFEH or FFFF FFFFH to the base address register is directed to the limit register. The data returned on subsequent reads from the register is the contents of the base address register — not the contents of the corresponding limit register. If any other value is written to the base address register, that value is programmed into the base address register. When determining block size requirements, reading to or writing from the base address register must be using 32-bit configuration cycles. However, configuration cycles not used to determine block size requirements can be performed as 8-, 16-, or 32-bit accesses.

Table 16-20. Device Specific Instructions for Base Address Register (Sheet 2 of 2)

Device Part Number	Value Written to the BAR	Effect of writing the value to the Base Address Register
80960RP 33/3.3 A-0 80960RD 66/3.3 A-0	FFFF FFFEH or FFFF FFFFH	The read after a write of FFFF FFFEH or FFFF FFFFH to the base address register is directed to the limit register. The data returned on subsequent reads from the base address register returns the limit register contents until the base address register is rewritten with a value other than FFFF FFFEH or FFFF FFFFH. If any other value is written to the base address register, that value is programmed into the base address register. When determining block size requirements, reading to or writing from the base address register must be using 32-bit configuration cycles. However, configuration cycles not used to determine block size requirements can be performed as 8-, 16-, or 32-bit accesses.
80960RP 33/3.3 B-0 80960RD 66/3.3 B-0	FFFF FFFEH or FFFF FFFFH	The limit register is a bitwise enable of the base address register. When any limit register bits are set to a 1, the corresponding bit in the base register is enabled as read/write. Once the base address register is enabled through the limit register, all 1's can be written to the base register as described in Section 6.2.5.1 of the *PCI Local Bus Specification*, revision 2.1. Reading the base address register after ones are written to the base address register yields the memory block size requirement. Values used for programming the limit register should be similar to those listed in Table 16-21. Any access to the base address register can be performed as on 8-, 16-, or 32-bit access.

Table 16-21. Memory Block Size Read Response (Sheet 1 of 2)

Response After Writing all 1s to the Base Address Register	Block Size
FFFF F000H	4 Kbytes
FFFF E000H	8 Kbytes
FFFF C000H	16 Kbytes
FFFF 8000H	32 Kbytes
FFFF 0000H	64 Kbytes
FFFE 0000H	128 Kbytes
FFFC 0000H	256 Kbytes
FFF8 0000H	512 Kbytes
FFF0 0000H	1 Mbytes
FFE0 0000H	2 Mbytes

Table 16-21. Memory Block Size Read Response (Sheet 2 of 2)

Response After Writing all 1s to the Base Address Register	Block Size
FFC0 0000H	4 Mbytes
FF80 0000H	8 Mbytes
FF00 0000H	16 Mbytes
FE00 0000H	32 Mbytes
FC00 0000H	64 Mbytes
F800 0000H	128 Mbytes
F000 0000H	256 Mbytes
E000 0000H	512 Mbytes
C000 0000H	1 Gbytes
8000 0000H	2 Gbytes
0000 0000H	Register Not Implemented, No address space required.

As an example, assume that FFFF FFFFH is written to the ATU Primary Inbound Base Address Register (PIABAR) and the value read back is FFF0 0004H. Bit zero is a zero, so the device requires memory address space. Bits 2:1 are 002, so the memory can be located anywhere within 32-bit address space (4 Gbytes). Bit three is one, so the memory does support prefetching. Scanning upwards starting at bit four, bit twenty is the first one bit found. The binary-weighted value of this bit is 1,048,576, indicated that the device requires 1 Mbyte of memory space.

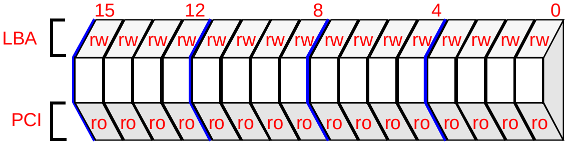
Table 16-22. Base Address and Limit Register Descriptions

Base Address Register	Limit Register	Description
Primary Inbound ATU Base Address Register	Primary Inbound ATU Limit Register	Defines the inbound translation window from the primary PCI bus.
Secondary Inbound ATU Base Address Register	Secondary Inbound ATU Limit Register	Defines the inbound translation window from the secondary PCI bus.
Expansion ROM Base Address Register	Expansion ROM Limit Register	Defines the window of addresses used by a primary bus master for reading from an expansion ROM.

16.7.13 ATU Subsystem Vendor ID Register - ASVIR

ATU Subsystem Vendor ID Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1.

Table 16-23. ATU Subsystem Vendor ID Register - ASVIR

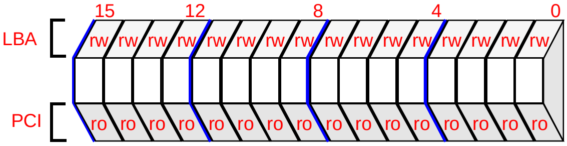
<table>
<tr><td colspan="3">
15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

</td></tr>
<tr><td colspan="2"><b>LBA:</b> 122CH
<b>PCI:</b> 2CH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>15:00</td><td>0000H</td><td>Subsystem Vendor ID - This register uniquely identifies the add-in board or subsystem vendor.</td></tr>
</table>

16.7.14 ATU Subsystem ID Register - ASIR

ATU Subsystem ID Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1.

Table 16-24. ATU Subsystem ID Register - ASIR

<table>
<tr><td colspan="3">
15
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

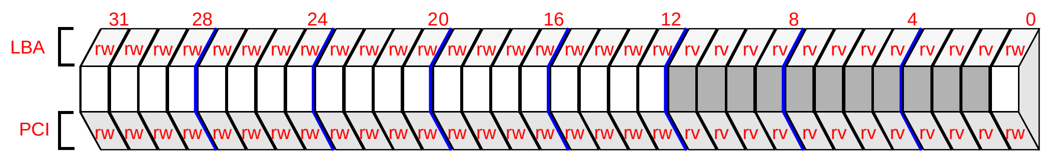
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 122EH
<b>PCI:</b> 2EH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>15:00</td><td>0000H</td><td>Subsystem ID - uniquely identifies the add-in board or subsystem</td></tr>
</table>

16.7.15 Expansion ROM Base Address Register - ERBAR

The Expansion ROM Base Address Register defines the block of memory addresses used for containing the Expansion ROM. It permits the inclusion of multiple code images, allowing the device to be initialized. The code image supplied consists of either executable code or an interpreted code. Each code image must start on a 512 byte boundary and each must contain the PCI Expansion ROM header. Image placement in ROM space depends on the length of code images which precede it within ROM. ERBAR defines the base address and describes the required memory block size; see Section 16.7.12 (pg. 16-37). Expansion ROM address space (limit size) can be a minimum of 4 Kbytes or a maximum of 16 Mbytes.

The Expansion ROM Base Address Register's programmed value must comply with the PCI programming requirements for address alignment. Refer to the *PCI Local Bus Specification*, revision 2.1 for additional information on programming Expansion ROM base address registers.

Table 16-25. Expansion ROM Base Address Register - ERBAR



LBA: 1230H
PCI: 30H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Expansion ROM Base Address - These bits define the actual location where the Expansion ROM address window resides when addressed from the primary PCI bus on any 2 Kbyte boundary.
11:01	000H	Reserved
00	0_2	Address Decode Enable - This bit field shows the ROM address decoder is enabled or disabled. When cleared, indicates the address decoder is disabled.

16.7.16 ATU Interrupt Line Register - ATUILR

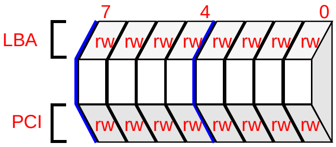
ATU Interrupt Line Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register identifies the system interrupt controller’s interrupt request lines which connect to the device’s PCI interrupt request lines (as specified in the interrupt pin register).

In a PC environment, for example, the register values and corresponding connections are:

- 00H - 0FH: correspond to IRQ0 through IRQ15
- 10H - FEH: reserved
- FFH: “unknown” or “no connection”

The operating system or device driver can examine each device’s interrupt pin and interrupt line register to determine which system interrupt request line the device uses to issue requests for service.

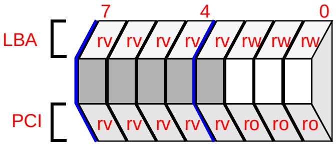
Table 16-26. ATU Interrupt Line Register - ATUILR

<table>
<tr><td colspan="3">
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 123CH
<b>PCI:</b> 3CH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>07:00</td><td>FFH</td><td>Interrupt Assigned - system-assigned value identifies which system interrupt controller’s interrupt request line connects to the device's PCI interrupt request lines (as specified in the interrupt pin register).</td></tr>
</table>

16.7.17 ATU Interrupt Pin Register - ATUIPR

ATU Interrupt Pin Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register identifies the interrupt pin the ATU and Messaging Unit interface uses. The i960 Rx I/O processor is a PCI multifunction device and, as such, can generate more than one interrupt output. The interrupt output is for the Messaging Unit on P_INTA#, P_INTB#, P_INTC#, or P_INTD#. The i960 core processor modifies the pin register to match the PCI interrupts which the Messaging Unit generates.

Table 16-27. ATU Interrupt Pin Register - ATUIPR

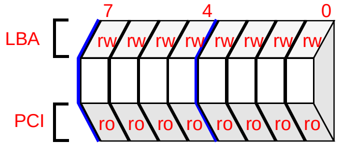
<table>
<tr><td colspan="3">
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 123DH
<b>PCI:</b> 3DH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>07:03</td><td>00H</td><td>Reserved.</td></tr>
<tr><td>02:00</td><td>001_2</td><td>Interrupt Used - Selects the interrupt pin the ATU interface uses.
001 - INTA# used
010 - INTB# used
011 - INTC# used
100 - INTD# used
All other values have the effect of disabling the ATU interface interrupt.</td></tr>
</table>

16.7.18 ATU Minimum Grant Register - ATUMGNT

ATU Minimum Grant Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register specifies the burst period the device requires in increments of 8 PCI clocks.

This register and the ATU Maximum Latency register are information-only registers which the configuration uses to determine frequency (how often) and duration (how long) of a bus master's access to the PCI bus. This information is useful when determining the values to be programmed into the bus master latency timers and in programming the algorithm to be used by the PCI bus arbiter.

Table 16-28. ATU Minimum Grant Register - ATUMGNT



LBA: 123EH
PCI: 3EH

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

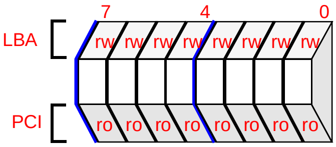
Bit	Default	Description
07:00	00H	This register specifies how long a burst period the device needs in increments of 8 PCI clocks. A zero value indicates the device has no stringent requirement.

16.7.19 ATU Maximum Latency Register - ATUMLAT

ATU Maximum Latency Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register specifies how often the device needs to access the PCI bus in increments of 8 PCI clocks.

This register and the Minimum Grant Register are information-only registers which the configuration uses to determine how often a bus master typically requires access to the PCI bus and the duration of a typical transfer when it does acquire the bus. This information is useful in determining the values to be programmed into the bus master latency timers and in programming the algorithm to be used by the PCI bus arbiter.

Table 16-29. ATU Maximum Latency Register - ATUMLAT

<table>
<tr><td colspan="3">
</td></tr>
<tr><td colspan="2"><b>LBA:</b> 123FH
<b>PCI:</b> 3FH</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</td></tr>
<tr><td><b>Bit</b></td><td><b>Default</b></td><td><b>Description</b></td></tr>
<tr><td>07:00</td><td>00H</td><td>Specifies frequency (how often) the device needs to access the PCI bus in increments of 8 PCI clocks. A zero value indicates the device has no stringent requirement.</td></tr>
</table>

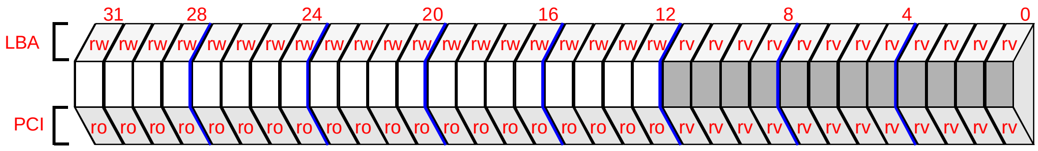
16.7.20 Primary Inbound ATU Limit Register - PIALR

Primary inbound address translation occurs for data transfers occurring from the PCI bus (originated from the primary PCI bus) to the 80960 local bus. The address translation block converts PCI addresses to 80960 local bus addresses.

All data transfers are directly translated; thus, the bus master which initiates the transfer breaks unaligned transfers into multiple data transfers. Byte enables specify valid data paths.

The primary inbound translation base address is specified in Section 16.7.11 (pg. 16-36). When determining block size requirements — as described in Section 16.7.12 (pg. 16-37) — the primary translation limit register provides the block size requirements for the primary base address register. The remaining registers used for performing address translation are discussed in Section 16.3.1 (pg. 16-5).

Table 16-30. Primary Inbound ATU Limit Register - PIALR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv rv rv rv rv rv rv rv rv rv rv

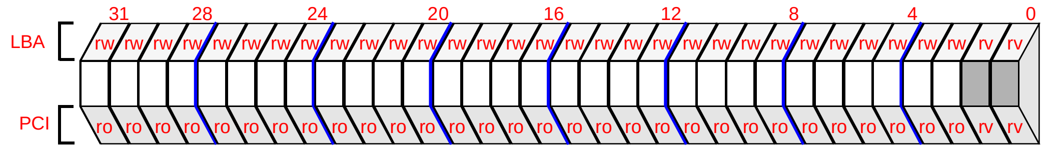
LBA: 1240H
PCI: 40H
		Legend:
		RV = Reserved
		RS = Read/Set
		LBA = 80960 Local Bus Address
			NA = Not Accessible
			PR = Preserved
			RC = Read Clear
				RO = Read Only
				RW = Read/Write
				
				PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	FFFF FH	Primary Inbound Translation Limit - This value (Table 16-22) determines the memory block size required for the primary ATU translation unit.
11:00	000H	Reserved

16.7.21 Primary Inbound ATU Translate Value Register - PIATVR

The Primary Inbound ATU Translate Value Register (PIATVR) contains the local address used to convert primary PCI bus addresses. The converted address is driven on the local bus as a result of primary inbound ATU address translation.

Table 16-31. Primary Inbound ATU Translate Value Register - PIATVR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv

LBA: 1244H
PCI: 44H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

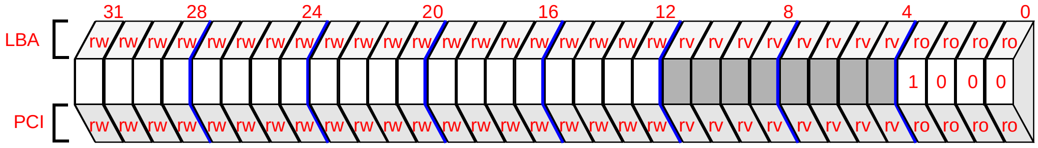
Bit	**Default**	**Description**
31:02	0001 000H	Primary Inbound ATU Translation Value - This value is used to convert the primary PCI address to local addresses. This value must be word-aligned on the 80960 local bus. The default address allows the ATU to access the internal 80960 local bus memory-mapped registers.
01:00	00_2	Reserved.

16.7.22 Secondary Inbound ATU Base Address Register - SIABAR

The Secondary Inbound ATU Base Address Register (SIABAR) defines the block of memory addresses where the secondary inbound translation window begins. The inbound ATU decodes and forwards the bus request to the 80960 local bus with a translated address to map into the 80960 local memory. The SIABAR defines the base address and describes the required memory block size; see Section 16.7.12, Determining Block Sizes for Base Address Registers (pg. 16-37).

The effects on the base address register are that when a value of FFFF FFFFH or FFFF FFFEH is written to the SIABAR, the next read from the register returns data from the Secondary Inbound ATU Limit Register (SIALR) and not the SIABAR.

Table 16-32. Secondary Inbound ATU Base Address Register - SIABAR



LBA: 1248H
PCI: 48H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Secondary Translation Base Address - These bits define the actual location to which the Secondary Translation function responds when addressed from the secondary PCI bus.
11:04	00H	Reserved.
03	1_2	Prefetchable Indicator - Defines the memory spaces as prefetchable.
02:01	00_2	Address Type - These bits define where the block of memory can be located. The base address must be located anywhere in the first 4 Gbyte of address space (lower 32-bits of address).
00	0_2	Memory Space Indicator - This bit field describes memory or I/O space base address. The secondary ATU does not occupy I/O space; thus, this bit must be zero.

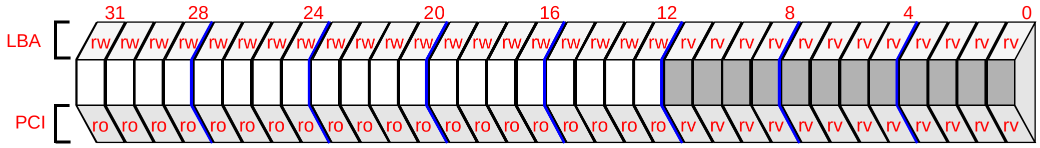
16.7.23 Secondary Inbound ATU Limit Register - SIALR

Secondary inbound address translation occurs for data transfers occurring from the secondary PCI bus to the 80960 local bus. The address translation block converts the PCI addresses to 80960 local bus address. All data transfers are directly translated; thus, the bus master initiating the data transfers breaks unaligned transfers into multiple data transfers. The byte enables specify which data paths are valid.

The secondary translation base address is specified in Section 16.7.22, Secondary Inbound ATU Base Address Register - SIABAR (pg. 16-48). When determining the block size requirements as described in Section 16.7.12, Determining Block Sizes for Base Address Registers (pg. 16-37), the secondary limit register provides the block size requirements for the secondary base address register. The remaining registers used for performing address translation are discussed in Section 16.3.1, Inbound Address Translation (pg. 16-5).

The default value of FFFF F000H is a 4 Kbyte memory block size.

Table 16-33. Secondary Inbound ATU Limit Register - SIALR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv rv rv rv rv rv rv rv rv rv rv

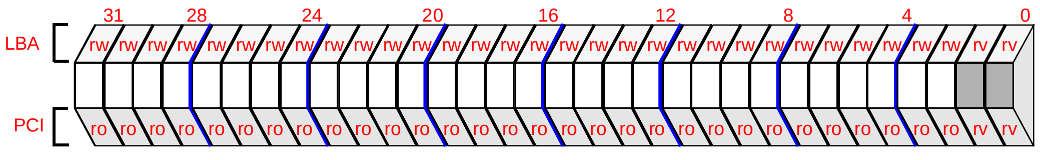
LBA: 124CH
PCI: 4CH
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	FFFF FH	Secondary Inbound ATU Limit - This value (Table 16-22) determines the memory block size required for the secondary ATU translation unit.
11:00	000H	Reserved

16.7.24 Secondary Inbound ATU Translate Value Register - SIATVR

The Secondary Inbound ATU Translate Value Register (SIATVR) contains the 80960 local address used to convert the secondary PCI bus address to a local address. This address is driven on the 80960 local bus as a result of secondary inbound ATU address translation.

Table 16-34. Secondary Inbound ATU Translate Value Register - SIATVR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv

LBA: 1250H
PCI: 50H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

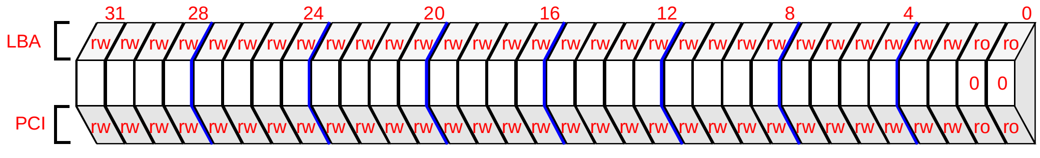
Bit	Default	Description
31:02	0000 1000H	Secondary Inbound ATU Translate Value - Used to convert the secondary PCI address to a local address. The secondary inbound address translation value must be word aligned on the 80960 local bus. The default address allows the translation unit access to the internal i960 Rx I/O processor memory-mapped registers.
01:00	00_2	Reserved.

16.7.25 Primary Outbound Memory Window Value Register - POMWVR

The Primary Outbound Memory Window Value Register (POMWVR) contains the primary PCI address used to convert 80960 local addresses for outbound transactions. This address is driven on the primary PCI bus as a result of primary outbound ATU address translation. See Section 16.3.6 (pg. 16-11) for details on outbound address translation.

Primary memory window 0 is from 80960 local bus address 8000 0000H to 83FF FFFFH with the fixed length of 64 Mbytes.

Table 16-35. Primary Outbound Memory Window Value Register - POMWVR


31 28 24 20 16 12 8 4 0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro
0 0
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro

LBA: 1254H
PCI: 54H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

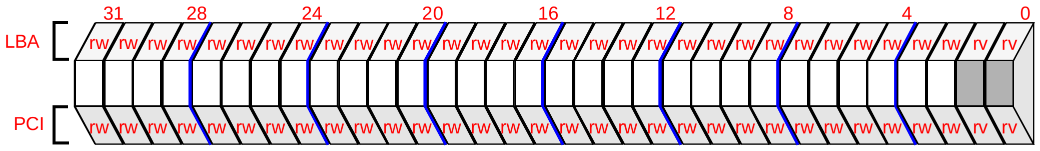
Bit	Default	Description
31:02	0000 0000H	Primary Outbound MW Value - Used to convert 80960 local addresses to PCI addresses.
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

16.7.26 Primary Outbound I/O Window Value Register - POIOWVR

The Primary Outbound I/O Window Value Register (POIOWVR) contains the primary PCI I/O address used to convert the local bus access to a PCI address. This address is driven on the primary PCI bus as a result of primary outbound ATU address translation. See Section 16.3.6, Outbound Address Translation (pg. 16-11) for details on outbound address translation.

The primary I/O window is from 80960 local bus address 9000 0000H to 9000 FFFFH with a fixed length of 64 Kbytes.

Table 16-36. Primary Outbound I/O Window Value Register - POIOWVR


LBA
PCI
31
28
24
20
16
12
8
4
0
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv

LBA: 125CH
PCI: 5CH

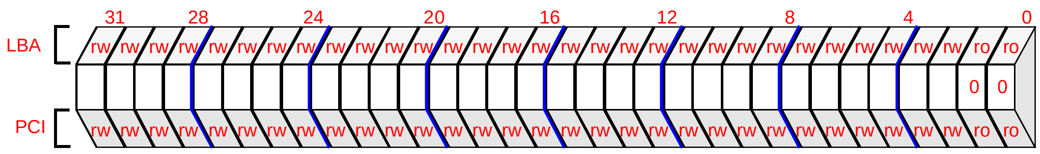
Legend:
NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000H	Primary Outbound I/O Window Value - Used to convert local addresses to PCI addresses.
01:00	00_2	Reserved.

16.7.27 Primary Outbound DAC Window Value Register - PODWVR

The Primary Outbound DAC Window Value Register (PODWVR) contains the primary PCI DAC address used to convert an 80960 local address. This address is driven on the primary PCI bus as a result of primary outbound ATU address translation. See Section 16.3.6, Outbound Address Translation for details on outbound address translation. This register is used in conjunction with the Primary Outbound Upper 64-Bit DAC Register. The primary DAC window is from 80960 local bus address 8400 0000H to 87FF FFFFH with the fixed length of 64 Mbytes.

Table 16-37. Primary Outbound DAC Window Value Register - PODWVR



31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro
0 0
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro

LBA: 1260H
PCI: 60H

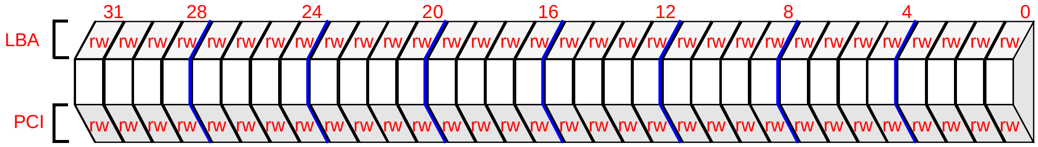
Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 Local Bus Address
PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000H	Primary Outbound DAC Window Value - This value the primary ATU uses to convert 80960 local addresses to PCI addresses.
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

16.7.28 Primary Outbound Upper 64-bit DAC Register - POUDR

The Primary Outbound Upper 64-bit DAC Register (POUDR) defines the upper 32-bits of address used during a dual address cycle. This enables the primary outbound ATU to directly address anywhere within the 64 bit host address space.

Table 16-38. Primary Outbound Upper 64-bit DAC Register - POUDR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw

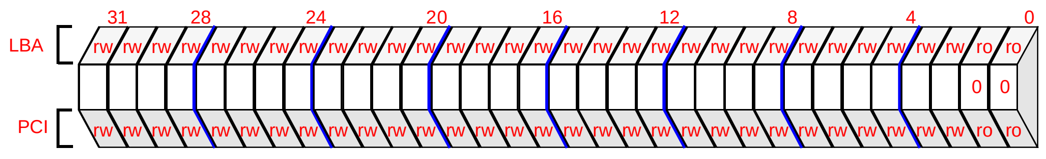
LBA: 1264H
PCI: 64H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	These bits define the upper 32-bits of address driven during the dual address cycle (DAC).

16.7.29 Secondary Outbound Memory Window Value Register - SOMWVR

The Secondary Outbound Memory Window Value Register (SOMWVR) contains the secondary PCI address used to convert 80960 local addresses for outbound transactions. This address is driven on the secondary PCI bus as a result of secondary outbound ATU address translation. See Section 16.3.6, Outbound Address Translation (pg. 16-11) for details on outbound address translation. The secondary memory window is from 80960 local bus address 8800 0000H to 8BFF FFFFH with the fixed length of 64 Mbytes.

Table 16-39. Secondary Outbound Memory Window Value Register - SOMWVR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro
0 0
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro

LBA: 1268H
PCI: 68H
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

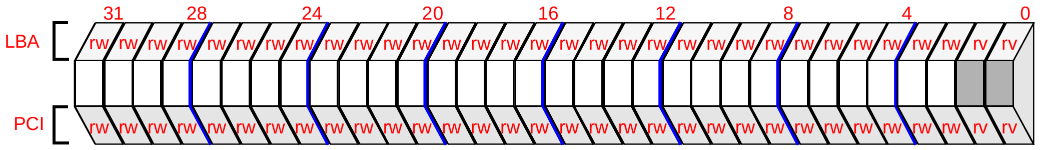
Bit	Default	Description
31:02	0000 0000H	Secondary Outbound Memory Window Value - Used to convert 80960 local addresses to PCI addresses.
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

16.7.30 Secondary Outbound I/O Window Value Register - SOIOWVR

The Secondary Outbound I/O Window Value Register (SOIOWVR) contains the secondary PCI I/O address used to convert 80960 local addresses. This address is driven on the secondary PCI bus as a result of secondary outbound ATU address translation. See Section 16.3.6, Outbound Address Translation for details on outbound address translation.

When the Secondary PCI Boot Mode bit in the ATUCR is set, this register translates local addresses that access the region of FE00 0000H to FFFF FFFFH. When clear, this register translates local addresses that access the secondary I/O window from 9001 0000H to 9001 FFFFH.

Table 16-40. Secondary Outbound I/O Window Value Register - SOIOWVR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv

LBA: 126CH
PCI: 6CH

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

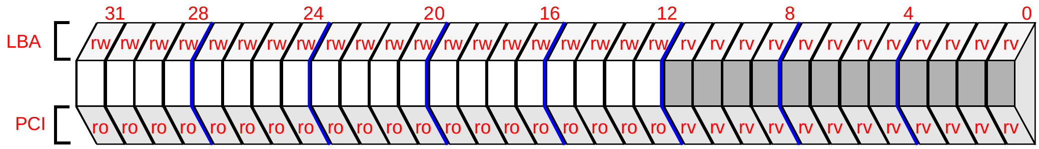
Bit	Default	Description
31:02	0000 0000H	Secondary Outbound I/O Window Value - Used to convert local addresses to PCI addresses.
01:00	00_2	Reserved.

16.7.31 Expansion ROM Limit Register - ERLR

The Expansion ROM Limit Register (ERLR) defines the block size of addresses the primary ATU defines as Expansion ROM address space. The block size is programmed by writing a value into the ERLR from the i960 core processor. The possible programmed values range from 4 Kbytes (FFFF F000H) to 16 Mbytes (FF00 0000H).

The Expansion ROM base address is specified in Section 16.7.15, "Expansion ROM Base Address Register - ERBAR" (pg. 16-41). When determining the block size requirements, the Expansion ROM Limit Register provides the block size requirements for the Expansion ROM Base Address Register.

Table 16-41. Expansion ROM Limit Register - ERLR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv rv rv rv rv rv rv rv rv rv rv

LBA: 1274H
PCI: 74H

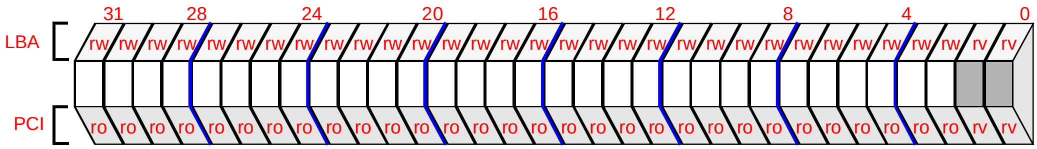
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Expansion ROM Limit - Block size of memory required for the Expansion ROM translation unit. Default value is 0, which indicates no expansion ROM address space.
11:00	000H	Reserved.

16.7.32 Expansion ROM Translate Value Register - ERTVR

The Expansion ROM Translate Value Register contains the 80960 local bus address which the primary ATU converts the primary PCI bus access. This address is driven on the 80960 local bus address as a result of primary Expansion ROM address translation.

Table 16-42. Expansion ROM Translate Value Register - ERTVR


31 28 24 20 16 12 8 4 0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv

LBA: 1278H
PCI: 78H

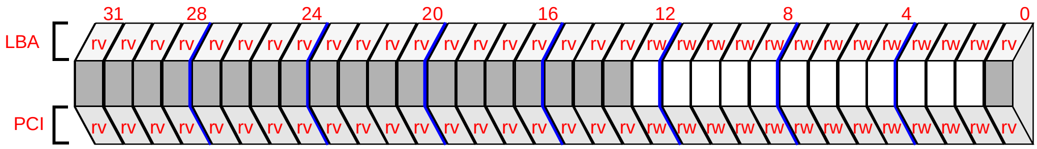
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000H	Expansion ROM 80960 Translation Value - Used to convert PCI addresses to 80960 local addresses for Expansion ROM accesses. The Expansion ROM address translation value must be word aligned on the 80960 local bus.
01:00	00_2	Reserved.

16.7.33 ATU Configuration Register - ATUCR

The ATU Configuration Register contains the control bits to enable and disable the interrupts generated by the ATU. This register also controls the outbound address translation from both the primary and secondary outbound translation units and contains a bit for Expansion ROM width.

Table 16-43. ATU Configuration Register - ATUCR (Sheet 1 of 3)

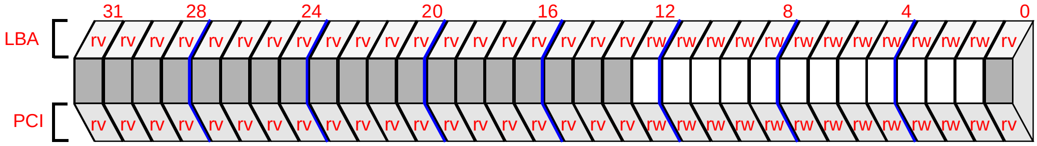

31 28 24 20 16 12 8 4 0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rv
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rv

LBA: 1288H
PCI: 88H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:13	0000H	Reserved.

Table 16-43. ATU Configuration Register - ATUCR (Sheet 2 of 3)

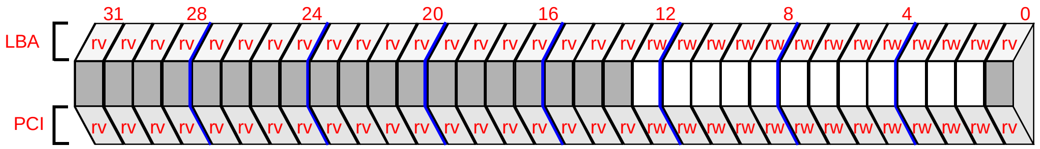

31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rv
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rv

LBA: 1288H
PCI: 88H

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 Local Bus Address
PCI = PCI Configuration Address Offset

Bit	Default	Description
12	0_2	Secondary Bus, Messaging Unit Access Enable - When set, the PCI-to-PCI Bridge unit can forward a transaction from the secondary PCI interface, through the bridge, and to the Messaging Unit (first 4 Kbytes of the primary ATU inbound address space) on the primary PCI interface. For correct operation, the transaction must be a valid bridge address (claimed by secondary interface of the bridge and forwarded to the primary interface of the bridge) as well as a valid Messaging Unit address. When clear, the Messaging Unit cannot claim a transaction mastered by the primary interface of the bridge.
11	0_2	Secondary PCI Boot Mode - When set, the secondary ATU claims all local bus accesses with addresses in the range FE00 0000H to FFFF FFFFH. This allows the i960 core processor to boot from the secondary PCI bus. The translation algorithm uses the Secondary Outbound I/O Window Value Register in this mode.
10	0_2	Secondary SERR Interrupt Enable - When set, the i960 core processor receives an NMI# when the Primary ATU detects that S_SERR# was asserted. When clear, no interrupt is sent.
09	0_2	Primary SERR Interrupt Enable - When set, the i960 core processor receives an NMI# when the Primary ATU detects that S_SERR# was asserted. When clear, no interrupt is sent.
08	0_2	Direct Addressing Enable - When set, enables direct addressing through the ATUs. Local bus cycles with an address between 0000 1000H and 7FFF FFFFH are automatically forwarded to the PCI bus with no address translation. The ATU which claims the direct addressing transaction depends on the Secondary Direct Addressing Select bit state.
07	0_2	Secondary Direct Addressing Select - When set, results in direct addressing outbound transactions to be forwarded through the secondary ATU to the secondary PCI bus. When clear, direct addressing uses the primary ATU and the primary PCI bus. The Direct Addressing Enable bit must be set to enable direct addressing.
06	0_2	Expansion ROM Width - When clear, this bit signifies that an 8-bit Expansion ROM is being used. When set, this bit signifies that 32-bit Expansion ROM is in use. Used in conjunction with the ERBAR address decode enable (bit 0).
05	0_2	Secondary ATU PCI Error Interrupt Enable - This bit acts as a mask for Secondary ATU Interrupt Status Register bits 4:0. When set, enables an interrupt to the i960 core processor when any of these bits are set in the SATUISR. When cleared, disables the interrupt.

Table 16-43. ATU Configuration Register - ATUCR (Sheet 3 of 3)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rv
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rv

LBA: 1288H
PCI: 88H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

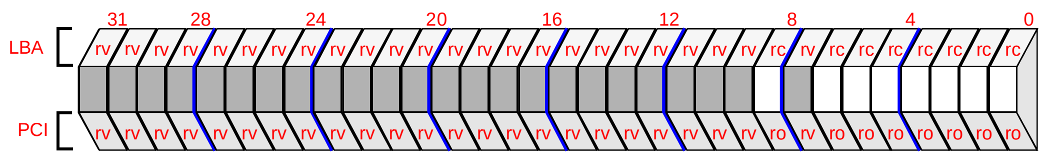
Bit	Default	Description
04	0_2	Primary ATU PCI Error Interrupt Enable - This bit acts as a mask for Primary ATU Interrupt Status Register bits 4:0. When set, enables an interrupt to the i960 core processor when any of these bits are set in the PATUISR. When cleared, disables the interrupt.
03	0_2	ATU BIST Interrupt Enable - When set, enables an interrupt to the i960 core processor when the start BIST bit is set in the ATUBISTR register. This bit is also reflected as the BIST Capable bit 7 in the ATUBISTR register.
02	0_2	Secondary Outbound ATU Enable - When set, enables the secondary outbound address translation unit. When cleared, disables the secondary outbound ATU.
01	0_2	Primary Outbound ATU Enable - When set, enables the primary outbound address translation unit. When cleared, disables the primary outbound ATU.
00	0_2	Reserved.

16.7.34 Primary ATU Interrupt Status Register - PATUISR

The Primary ATU Interrupt Status Register notifies the i960 core processor of the Primary ATU interrupt source. Writes to this register clear the source of the interrupt. All register bits are Read Only from PCI and Read/Clear from the local bus.

Bits 4:0 are a direct reflection of Primary ATU Status Register bit 8 and bits 14:11 (respectively). These bits are set at the same time by hardware but need to be cleared independently. Bits 6:5 are set by an error associated with the Memory Controller. Bit 8 is for software BIST. The conditions that result in a Primary ATU interrupt are cleared when the appropriate bits in this register are set (=1).

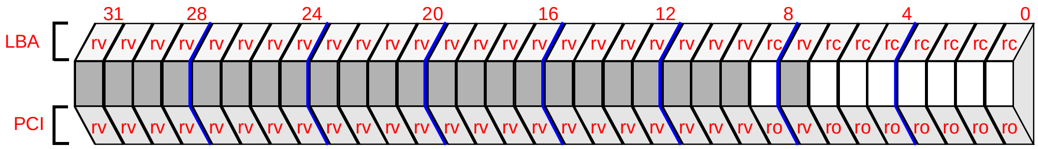
Table 16-44. Primary ATU Interrupt Status Register - PATUISR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rv rc rc rc rc rc rc rc
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro rv ro ro ro ro ro ro ro

LBA: 1290H
PCI: 90H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:09	0000 00H	Reserved.
08	0_2	ATU BIST Interrupt - When set, the host processor has set the start BIST, ATUBISTR register bit 6, and the ATU BIST interrupt enable, ATUCR register bit 12, is enabled. The i960 core processor can initiate the software BIST and store the result in ATUBISTR register bits 3:0.
07	0_2	Reserved.
06	0_2	80960 local bus memory Fault - set when the Memory Controller detects a Memory Fault and the Primary ATU was the master for the transaction.
05	0_2	80960 local bus address Fault - set when the Memory Controller detects a Bus Fault and the Primary ATU was the master for the transaction.
04	0_2	P_SERR# Asserted - set when P_SERR# is asserted on the PCI bus.
03	0_2	PCI Master Abort - set when a transaction initiated by the ATU master interface ends in a Master-abort.
02	0_2	PCI Target Abort (master) - set when a transaction initiated by the ATU master interface ends in a Target Abort.
01	0_2	PCI Target Abort (target) - set when the ATU interface, acting as a target, terminates the transaction on the PCI bus with a target abort.

Table 16-44. Primary ATU Interrupt Status Register - PATUISR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rv rc rc rc rc rc rc rc rc
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro rv ro ro ro ro ro ro ro ro

LBA: 1290H
PCI: 90H
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
			RO = Read Only
			RW = Read/Write

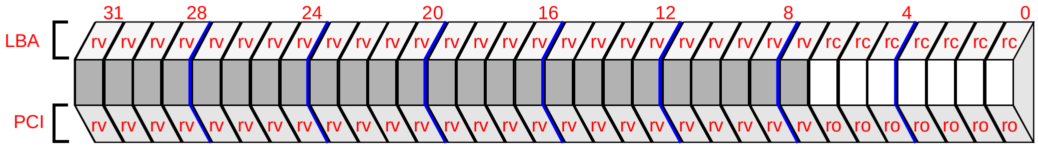
Bit	Default	Description
00	0_2	PCI Master Parity Error - The ATU interface sets this bit when three conditions are met: • bus agent asserted S_PERR# or observed S_PERR# asserted • agent setting the bit acted as the bus master for the operation in which the error occurred • Parity Checking Enable bit is set (in the Primary ATU Command Register)

16.7.35 Secondary ATU Interrupt Status Register - SATUISR

The Secondary ATU Interrupt Status Register notifies the i960 core processor of the Secondary ATU interrupt source. This register is written to clear the interrupt source to the i960 core processor's interrupt unit. All register bits are Read Only from PCI and Read/Clear from the local bus.

Conditions that result in a Secondary ATU interrupt are cleared when the appropriate bit in this register are set (=1).

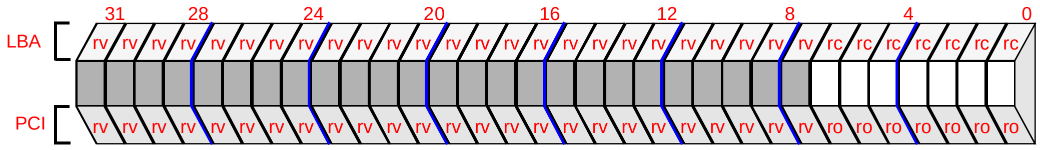
Table 16-45. Secondary ATU Interrupt Status Register - SATUISR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc rc rc
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro ro ro ro

LBA: 1294H
PCI: 94H
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
			RO = Read Only
			RW = Read/Write

Bit	Default	Description
31:07	0000 000H	Reserved.
06	0_2	80960 local bus memory Fault - set when the Memory Controller detects a Memory Fault and the Secondary ATU was the master for the transaction.

Table 16-45. Secondary ATU Interrupt Status Register - SATUISR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc rc
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro ro ro

LBA: 1294H
PCI: 94H

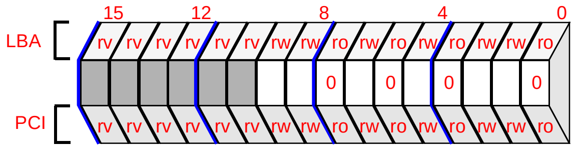
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
05	0_2	80960 local bus address Fault - set when the Memory Controller detects a Bus Fault and the Secondary ATU was the master for the transaction.
04	0_2	S_SERR# Asserted - set when P_SERR# is asserted on the secondary PCI bus.
03	0_2	PCI Master Abort - set when a transaction initiated by the ATU master interface ends in a Master-abort.
02	0_2	PCI Target Abort (master) - set when a transaction initiated by the ATU master interface ends in a Target Abort.
01	0_2	PCI Target Abort (target) - set when the ATU interface, acting as a target, terminates the transaction on the PCI bus with a Target Abort.
00	0_2	PCI Master Parity Error - The secondary ATU interface sets this bit when three conditions are met: • bus agent asserted S_PERR# or observed S_PERR# asserted • agent setting the bit acted as the bus master for the operation in which the error occurred • Parity Checking Enable bit is set (in the Secondary ATU Command Register)

16.7.36 Secondary ATU Command Register - SATUCMD

The Secondary ATU Command Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1 and, in most cases, affect the secondary PCI bus device behavior.

Table 16-46. Secondary ATU Command Register - SATUCMD

<table>
<tr><td colspan="3">

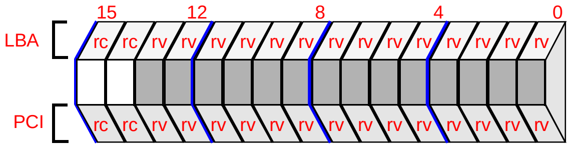
15
12
8
4
0
LBA
rv rv rv rv rv rv rw rw ro rw ro rw ro rw rw ro
0 0 0 0
PCI
rv rv rv rv rv rv rw rw ro rw ro rw ro rw rw ro

</td></tr>
<tr><td colspan="2"><b>LBA:</b> 1298H
<b>PCI:</b> 98H</td><td><b>Legend:</b> NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</td></tr>
<tr><th>Bit</th><th>Default</th><th>Description</th></tr>
<tr><td>15:10</td><td>00H</td><td>Reserved.</td></tr>
<tr><td>09</td><td>0_2</td><td>Fast Back to Back Enable - When cleared, the secondary ATU interface is not allowed to generate fast back-to-back cycles on its bus.</td></tr>
<tr><td>08</td><td>0_2</td><td>S_SERR# Enable - When cleared, the secondary ATU interface is not allowed to assert S_SERR# on the PCI interface.</td></tr>
<tr><td>07</td><td>0_2</td><td>Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field.</td></tr>
<tr><td>06</td><td>0_2</td><td>Parity Checking Enable - When set, the secondary ATU and DMA channel 2 must take normal action when a parity error is detected. When cleared, parity checking is disabled.</td></tr>
<tr><td>05</td><td>0_2</td><td>VGA Palette Snoop Enable - The secondary ATU interface does not support I/O writes and therefore, does not perform VGA palette snooping.</td></tr>
<tr><td>04</td><td>0_2</td><td>Memory Write and Invalidate Enable - When set, DMA channel 2 may generate MWI commands. When clear, DMA channel 2 must use Memory Write commands instead of MWI.</td></tr>
<tr><td>03</td><td>0_2</td><td>Special Cycle Enable - The ATU interface does not respond to special cycle commands in any way. Not applicable. Not implemented and a reserved bit field.</td></tr>
<tr><td>02</td><td>0_2</td><td>Bus Master Enable - The secondary ATU interface can act as a master on the PCI bus. When cleared, disables the secondary ATU from generating PCI accesses. When set, allows the secondary ATU to behave as a bus master.
This enable bit also controls the DMA channel 2 master interface. The bit must be set before initiating an DMA transfer on the PCI bus.</td></tr>
<tr><td>01</td><td>0_2</td><td>Memory Enable - Controls the secondary ATU interface's response to PCI memory addresses. When cleared, the ATU interface does not respond to any memory access on the PCI bus.</td></tr>
<tr><td>00</td><td>0_2</td><td>I/O Space Enable - Controls the ATU interface response to I/O transactions on the primary side. Not implemented. The secondary ATU does not support I/O space.</td></tr>
</table>

16.7.37 Secondary ATU Status Register - SATUSR

Secondary ATU Status Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1 for configuration space device status. The *read/clear* bits can only be set by the internal hardware and are cleared by either a reset condition or by writing a 1_2 to be cleared.

Table 16-47. Secondary ATU Status Register - SATUSR


15
12
8
4
0
LBA
PCI
rc rc rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rc rc rv rv rv rv rv rv rv rv rv rv rv rv rv rv

LBA: 129AH
PCI: 9AH
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

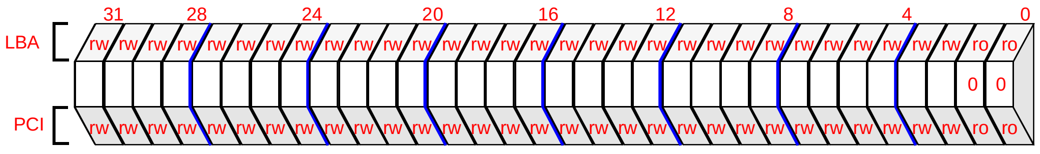
Bit	Default	Description
15	0_2	Parity Error - set when a parity error is detected on the secondary PCI bus even when the SATUCMD Register's Parity Checking Enable bit is clear. 0 - no parity error detected 1 - parity error detected
14	0_2	S_SERR# Asserted - set when the Secondary ATU asserted S_SERR# on the PCI bus. 0 - S_SERR# no asserted 1 - S_SERR# asserted
13:00	0000H	Reserved.

16.7.38 Secondary Outbound DAC Window Value Register - SODWVR

The Secondary Outbound DAC Window Value Register (SODWVR) contains the secondary PCI DAC address used to convert an 80960 local address. This address is driven on the secondary PCI bus as a result of secondary outbound ATU address translation. See Section 16.3.6, Outbound Address Translation (pg. 16-11) for details on outbound address translation. This register is used in conjunction with the Secondary Outbound Upper 64-Bit DAC Register.

The secondary DAC window is from 80960 local bus address 8C00 0000H to 8FFF FFFFH with the fixed length of 64 Mbytes.

Table 16-48. Secondary Outbound DAC Window Value Register - SODWVR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro
0 0
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw ro ro

LBA: 129CH
PCI: 9CH

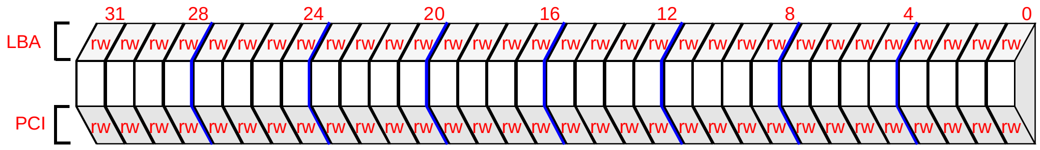
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000H	Secondary Outbound DAC Window Value - The secondary ATU converts 80960 local addresses to PCI addresses.
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. 00=linear incrementing mode.

16.7.39 Secondary Outbound Upper 64-bit DAC Register - SOUDR

The Secondary Outbound Upper 64-bit DAC Register (SOUDR) defines the upper 32 address bits used during a dual address cycle. This enables the secondary outbound ATU to directly address anywhere within the 64-bit host address space.

Table 16-49. Secondary Outbound Upper 64-bit DAC Register - SOUDR



LBA: 12A0H	**Legend:**	NA = Not Accessible	RO = Read Only	
PCI: A0H	RV = Reserved	PR = Preserved	RW = Read/Write	
	RS = Read/Set	RC = Read Clear		
	LBA = 80960 Local Bus Address		PCI = PCI Configuration Address Offset	

Bit	Default	Description
31:00	0000 0000H	Secondary Outbound Upper 64-bit DAC Address - These bits define the upper 32 address bits driven during the dual address cycle (DAC).

16.7.40 Primary Outbound Configuration Cycle Address Register - POCCAR

The Primary Outbound Configuration Cycle Address Register holds the 32-bit PCI configuration cycle address. The i960 core processor writes the PCI configuration cycles address that enables the primary outbound configuration read or write. The i960 core processor performs a read or write to the Primary Outbound Configuration Cycle Data Port to initiate the configuration cycle on the primary PCI bus.

The value programmed into these registers is not a byte address. See the *PCI Local Bus Specification*, revision 2.1 for information regarding configuration address cycle formats.

Table 16-50. Primary Outbound Configuration Cycle Address Register - POCCAR

	31			28				24				20				16				12				8				4				0
LBA	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
PCI	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro

LBA: 12A4H
PCI: A4H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

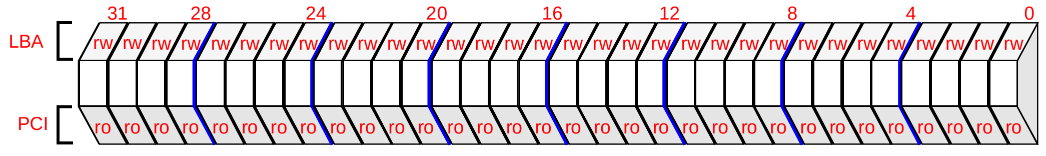
Bit	Default	Description
31:00	0000 0000H	Primary Configuration Cycle Address - These bits define the 32-bit PCI address used during an outbound configuration read or write cycle.

16.7.41 Secondary Outbound Configuration Cycle Address Register - SOCCAR

The Secondary Outbound Configuration Cycle Address Register holds the 32-bit PCI configuration cycle address. The i960 core processor writes the PCI configuration cycles address that enables the secondary outbound configuration read or write. The i960 core processor performs a read or write to the Secondary Outbound Configuration Cycle Data Port to initiate the configuration cycle on the secondary PCI bus.

The value programmed into these registers is not a byte address. See the *PCI Local Bus Specification*, revision 2.1 for information regarding configuration address cycle formats.

Table 16-51. Secondary Outbound Configuration Cycle Address Register - SOCCAR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro

LBA: 12A8H
PCI: A8H

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	Secondary Configuration Cycle Address - These bits define the 32-bit PCI address used during an outbound configuration read or write cycle.

16.7.42 Primary Outbound Configuration Cycle Data Port - POCCDP

The Primary Outbound Configuration Cycle Data Port initiates a configuration read or write on the primary PCI bus. The register is logical rather than physical meaning that it is an address not a register. The i960 core processor reads or writes the data registers memory-mapped address to initiate the configuration cycle on the PCI bus with the address found in the POCCAR.

The configuration cycle generated on the PCI bus enables the same bytes which are accessed in the corresponding data register. For example, a read of all 32 bits of this data register generates a 4-byte configuration read cycle on the primary PCI bus of the addressed configuration register. Also, a write of byte 2 (bits 23:16) of this data register generates a single byte configuration write cycle of byte 2 of the addressed configuration register. Similar actions take place for short accesses.

- For a configuration write, the data is latched from the 80960 local bus and forwarded directly to the ATU ODQ.
- For a read, the data is returned directly from the ATU IDQ to the i960 core processor and is never actually entered into the data register (which does not physically exist).

The POCCDP is only useful from 80960 local bus address space and appears as a reserved value within the ATU configuration space. The 80960 local bus address is 12ACH.

NOTE: This port should never be accessed by PCI Function 1 cycles or ATU inbound transactions.

16.7.43 Secondary Outbound Configuration Cycle Data Port - SOCCDP

The Secondary Outbound Configuration Cycle Data Port initiates a configuration read or write on the secondary PCI bus. The register is logical rather than physical meaning that it is an address not a register. The i960 core processor reads or writes the data registers memory-mapped address to initiate the configuration cycle on the PCI bus with the address found in the SOCCAR.

The configuration cycle generated on the PCI bus enables the same bytes which are accessed in the corresponding data register. For example, a read of all 32 bits of this data register generates a 4-byte configuration read cycle on the secondary PCI bus of the addressed configuration register. Also, a write of byte 2 (bits 23:16) of this data register generates a single byte configuration write cycle of byte 2 of the addressed configuration register. Similar actions take place for short accesses.

- For a configuration write, the data is latched from the 80960 local bus and forwarded directly to the ATU ODQ.
- For a read, the data is returned directly from the ATU IDQ to the i960 core processor and is never actually entered into the data register (which does not physically exist).

NOTE: This port should never be accessed by PCI Function 1 cycles or ATU inbound transactions.

The SOCCDP is only useful from 80960 local bus address space and appears as a reserved value within the ATU configuration space. The 80960 local bus address is 12B0H.

16.8 POWERUP/DEFAULT STATUS

The default/powerup values for all registers are shown within each register description.

16.9 RESET MODES

See sections Section 15.13.23, Bridge Control Register - BCR and Section 15.13.24, Extended Bridge Control Register - EBCR.

17

MESSAGING UNIT

CHAPTER 17
MESSAGING UNIT

This chapter describes the operation of the Messaging Unit (MU). The MU is the communications path between the host operating system and the I/O subsystem.

17.1 OVERVIEW

The MU sends and receives messages; it transfers data between the PCI system and the i960 core processor and notifies the respective system when new data arrives due to an interrupt. The MU has five messaging mechanisms. Each allows a host processor or external PCI agent and the i960®Rx I/O processor to communicate through message passing and interrupt generation. Each mechanism and corresponding sections are summarized as:

- Section 17.2, MESSAGE REGISTERS (pg. 17-2). Each of four registers hold a 32-bit value and generate an interrupt when any value is written.
- Section 17.3, DOORBELL REGISTERS (pg. 17-3). These two registers support software interrupts. Interrupts are generated when a Doorbell Register bit is set.
- Section 17.4, CIRCULAR QUEUES (pg. 17-4). The circular queues comply with the industry-standard Intelligent I/O (I_2O) interface for PCI applications. This message passing scheme uses four circular queues: two for inbound messages, two for outbound messages. Circular queues are implemented in i960 Rx I/O processor local memory.
- Section 17.5, INDEX REGISTERS (pg. 17-10). These use a portion of the i960 Rx I/O processor local memory to implement a large set of message registers. Interrupts are generated when an Index Registers is written. The address of the register written is captured.
- Section 17.6, APIC REGISTERS (pg. 17-11). These two registers provide an external PCI interface for accessing I/O APIC Registers. Interrupts are generated when the APIC Register Select Register is written.

Interrupt status for all interrupts is recorded in the Inbound Interrupt Status Register and Outbound Interrupt Status Register. Any MU-generated interrupt can be masked.

The MU uses the first 4 Kbytes of the primary inbound translation window in the Primary Address Translation Unit (PATU). This PCI address window is used for PCI transactions that access the i960 Rx I/O processor's local memory. The primary inbound translation window's PCI address is contained in the Primary Inbound ATU Base Address Register. See Section 16.3, ATU ADDRESS TRANSLATION (pg. 16-4) for more details on inbound ATU addressing.

From the PCI perspective, the MU is part of the Primary Address Translation Unit. The MU uses the PCI configuration registers of the Primary ATU for control and status information. The MU observes all PCI control bits in the Primary ATU Command Register and ATU Configuration Register. The MU reports all PCI errors in the Primary ATU Status Register.

Table 17-1 summarizes the five MU mechanisms.

Table 17-1. Messaging Unit (MU) Summary

Mechanism	Quantity	Assert PCI Interrupt Signals?	Generate i960 Core Processor Interrupt?
Message Registers	Two Inbound	No	Optional
	Two Outbound	Yes	No
Doorbell Registers	One Inbound	No	Optional
	One Outbound	Yes	No
Circular Queues	Four Circular Queues	Under certain conditions	Under certain conditions
Index Registers	1004 32-bit Memory Locations	No	Optional
APIC Registers	One Register Select	No	Yes
	One Window	No	Yes

17.2 MESSAGE REGISTERS

The i960 Rx I/O processor uses the message registers to send and receive messages. When written, these registers may cause an interrupt to be generated to either the i960 core processor or the PCI interrupt signals.

- Inbound messages are sent by the host processor and received by the i960 Rx I/O processor.
- Outbound messages are sent by the i960 Rx I/O processor and received by the host processor.

The interrupt status for outbound messages is recorded in the Outbound Interrupt Status Register. Interrupt status for inbound messages is recorded in the Inbound Interrupt Status Register.

17.2.1 Outbound Messages

The MU contains two outbound message registers. When an outbound message register is written by the i960 core processor, an interrupt may be generated on the P_INTA#, P_INTB#, P_INTC#, or P_INTD# interrupt pins. The interrupt pin used is determined by the value programmed in the ATU Interrupt Pin Register (See CHAPTER 16, ADDRESS TRANSLATION UNIT).

The PCI interrupt is recorded in the Outbound Interrupt Status Register. The interrupt causes the Outbound Message Interrupt bit to be set in the Outbound Interrupt Status Register. This is a Read/Clear bit that is set by MU hardware and cleared by software.

The interrupt is cleared when an external PCI agent writes a value of 1 to the Outbound Message Interrupt bit in the Outbound Interrupt Status Register to clear the bit (via a PCI configuration cycle).

The interrupt may be masked by the Mask bits in the Outbound Interrupt Mask Register.

17.2.2 Inbound Messages

The MU contains two inbound message registers. When an inbound message register is written by an external PCI agent, an interrupt may be generated to the i960 core processor. The interrupt may be masked by the Mask bits in the Inbound Interrupt Mask Register.

The i960 core processor interrupt is recorded in the Inbound Interrupt Status Register. The interrupt causes the Inbound Message Interrupt bit to be set in the Inbound Interrupt Status Register. This is a Read/Clear bit set by the MU.

The interrupt is cleared when the i960 core processor sets (=1) the Inbound Message Interrupt bit in the Inbound Interrupt Status Register.

17.3 DOORBELL REGISTERS

The two doorbell registers generate interrupts when their bits are set. The registers, described in the following subsections, are:

- Outbound Doorbell Register — allows the i960 core processor to generate a PCI interrupt.
- Inbound Doorbell Register — allows external PCI agents to generate interrupts to the i960 core processor.

17.3.1 Outbound Doorbells

The i960 core processor generates an interrupt by setting bits in the Outbound Doorbell Register and external PCI agents clear the interrupt by also setting bits in the same register.

When the Outbound Doorbell Register is written by the i960 core processor, an interrupt may be generated on the P_INTA#, P_INTB#, P_INTC#, or P_INTD# interrupt pins. An interrupt is generated when any doorbell register bits are set. The i960 core processor clearing (writing a 0 to) any bit, does not change the value of that bit and does not cause an interrupt to be generated. Once a bit is set in the Outbound Doorbell Register, it cannot be cleared by the i960 core processor.

The PCI interrupt pin used is determined by the value programmed in the ATU Interrupt Pin Register (See CHAPTER 16, ADDRESS TRANSLATION UNIT). The interrupt is recorded in the Outbound Interrupt Status Register.

The interrupt may be masked by the Outbound Interrupt Mask Register's Mask bits. When a Mask bit is set, no interrupt is generated for that bit. The Outbound Interrupt Mask Register affects only the generation of the interrupt and not the values written to the Outbound Doorbell Register.

The interrupt is cleared when an external PCI agent writes a 1 to the bits in the Outbound Doorbell Register that are set. Clearing a bit does not change the value of that bit and does not clear the interrupt.

17.3.2 Inbound Doorbells

When the Inbound Doorbell Register is written by an external PCI agent, an interrupt may be generated to the i960 core processor. An interrupt is generated when any doorbell register bits are set. An external PCI agent clearing (write a 0 to) any bit, does not change the value of that bit and does not cause an interrupt to be generated. Once a bit is set in the Inbound Doorbell Register, it cannot be cleared by any external PCI agent.

The interrupt is recorded in the Inbound Interrupt Status Register.

The interrupt may be masked by the Inbound Doorbell Interrupt Mask bit in the Inbound Interrupt Mask Register. When a mask bit is set, no interrupt is generated for that bit. The Inbound Interrupt Mask Register affects only the generation of the interrupt and not the values written to the Inbound Doorbell Register.

One bit in the Inbound Doorbell Register is reserved for an NMI interrupt.

The interrupt is cleared when the i960 core processor writes a value of 1 to the bits in the Inbound Doorbell Register that are set. Clearing a bit does not change the value of that bit and does not clear the interrupt.

17.4 CIRCULAR QUEUES

The MU has four circular queues: two inbound and two outbound. Messages are either *posted* or *free*. As related to these circular queues, *inbound* and *outbound* refer to the direction of message flow.

Inbound messages are either:

- *posted*: messages from other processors that the i960 core processor must process
- *free*: (empty) messages that other processors can reuse

Outbound messages are either:

- *posted*: messages from the i960 core processor which other processors must process
- *free*: (empty) messages that the i960 core processor can reuse

The two outbound queues allow the i960 core processor to post outbound messages in one queue and receive free messages returning from the host processor. The i960 core processor posts outbound messages; the host processor receives the posted message and, when finished with the message, places it back on the outbound free queue for reuse by the i960 core processor.

The two inbound queues allow the host processor to post inbound messages for the i960 Rx I/O processor in one queue and receive free messages returning from the i960 Rx I/O processor. The host processor posts inbound messages; the i960 core processor receives the posted message and, when finished with the message, places it back on the inbound free queue for reuse by the host processor.

The circular queues are accessed by external PCI agents through the inbound and outbound queue port locations in the PCI address space:

- The Inbound Queue Port, when read by an external PCI agent, returns the Inbound Free Queue data. When written, the data is placed on the Inbound Post Queue.
- The Outbound Queue Port, when read by an external PCI agent, returns the Outbound Post Queue data. When written, the data is placed on the Outbound Free Queue.

Data storage for circular queues must be allocated in i960 Rx I/O processor local memory. The circular queues base address is contained in the Queue Base Address Register (QBAR). Each queue entry is a 32-bit data value. Each read or write of the queue may access only one queue entry. Multi-word and sub-word accesses are not allowed.

Circular queue size ranges from 4 K entries (16 Kbytes) to 64 K entries (256 Kbytes). All four queues must be the same size and must be contiguous. Therefore, the total amount of local memory needed by the circular queues ranges from 64 Kbytes to 1 Mbytes. Queue size is determined by the MUCR Queue Size field; see Section 17.7.11 (pg. 17-26).

Starting addresses of each queue is based on the Queue Base Address and the Queue Size field. See Table 17-2.

Table 17-2. Queue Starting Addresses

Queue	Starting Address
Inbound Free Queue	QBAR
Inbound Post Queue	QBAR + Queue Size
Outbound Post Queue	QBAR + 2 * Queue Size
Outbound Free Queue	QBAR + 3 * Queue Size

Each circular queue has a head pointer and a tail pointer. The pointers are offsets from the Queue Base Address. Writes to a queue occur at the head of the queue and reads occur from the tail. The head and tail pointers are incremented by either the i960 core processor or MU hardware. Which unit maintains the pointer is determined by the writer of the queue. The pointers are incremented after the queue access. Both pointers wrap around to the first address of the circular queue when they reach the circular queue size.

The MU generates an interrupt to the i960 core processor and, under certain conditions, generates a PCI interrupt. In general, when a Post queue is written, an interrupt is generated to notify the receiver that a message was posted.

The MU only prevents queue overflows for the Outbound Free Queue. For the Outbound Free Queue, an NMI interrupt to the i960 core processor is generated when the head pointer equals the tail pointer and the queue is full to notify software of the error condition. Software must manage the circular queues to prevent overflow conditions.

Figure 17-1 diagrams the Circular Queue usage.

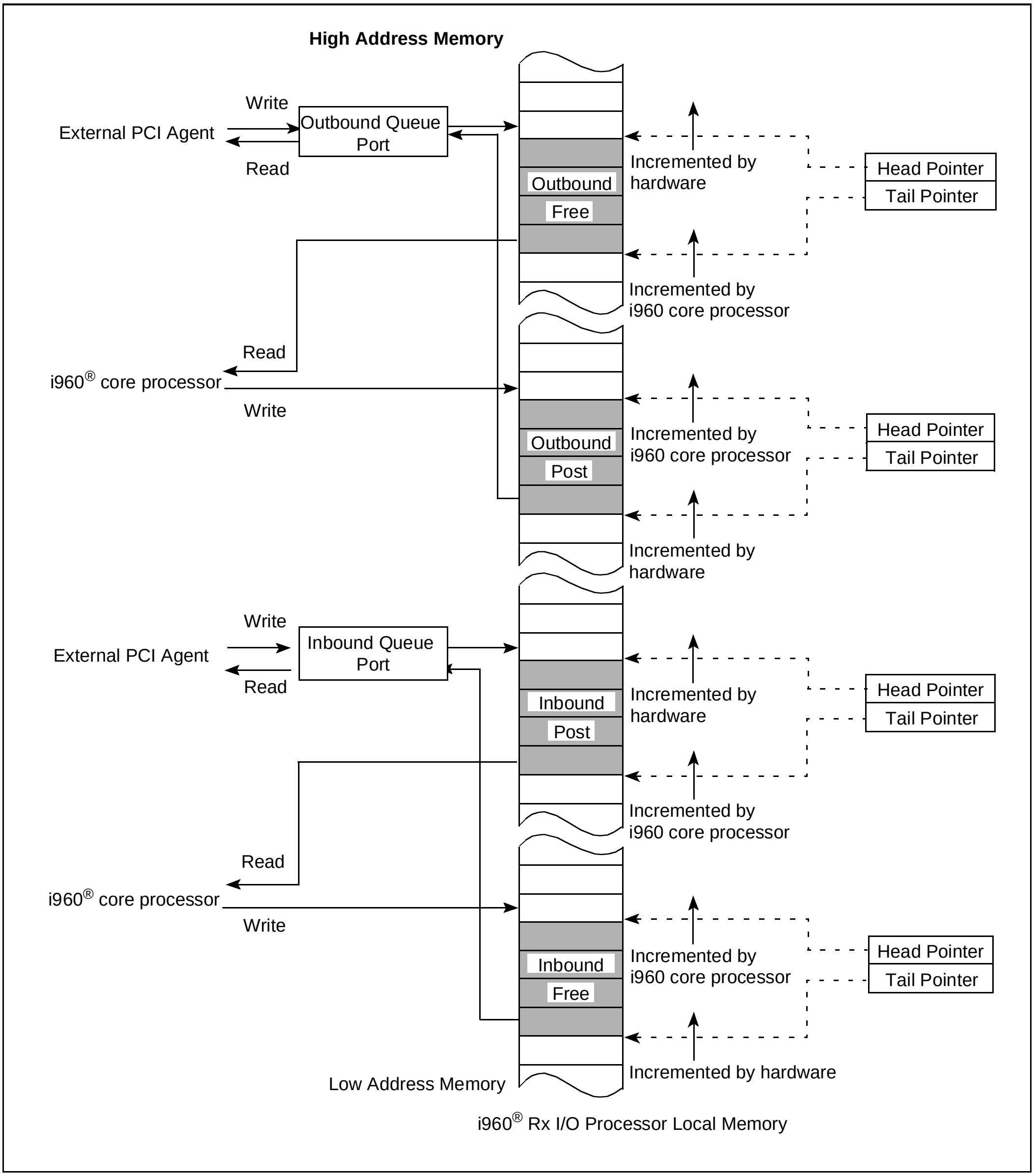


Figure 17-1. Circular Queue Operation

17.4.1 Inbound Post Queue

The Inbound Post Queue holds posted messages from external PCI masters for the i960 core processor to process. This queue is read from the queue tail by the i960 core processor. It is written to the queue head by external PCI agents. The tail pointer is maintained by the i960 core processor. The head pointer is maintained by MU hardware.

For a PCI write transaction that accesses the Inbound Queue Port, the MU writes the data to the local memory location pointed by the Inbound Post Head Pointer Register. The local memory address is Queue Base Register + Inbound Post Head Pointer Register.

When the data written to the Inbound Queue Port is written to local memory, the MU increments the Inbound Post Head Pointer Register.

An i960 core processor interrupt is generated when the Inbound Post Head Pointer Register is written. The Inbound Interrupt Status Register's Inbound Post Queue Interrupt bit indicates interrupt status. The interrupt is cleared when the Inbound Post Queue Interrupt bit is clear. The interrupt can be masked by the Inbound Interrupt Mask Register.

From the time that the PCI write transaction is received until the data is written in local memory and the Inbound Post Head Pointer Register is incremented, any PCI transaction that accesses the Inbound Queue Port is signalled a Retry.

The i960 core processor may read messages from the Inbound Post Queue by reading the data from the local memory location pointed to by the inbound Post Tail Pointer Register. The i960 core processor must then increment the Inbound Post Tail Pointer Register.

17.4.2 Inbound Free Queue

The Inbound Free Queue holds free inbound messages from the i960 core processor for external PCI masters to use. This queue is read from the queue tail by external PCI agents. It is written to the queue head by the i960 core processor. The tail pointer is maintained by MU hardware. The head pointer is maintained by the i960 core processor.

For a PCI read transaction that accesses the Inbound Queue Port, the MU reads data at the local memory location pointed by the Inbound Free Tail Pointer. The local memory address is Queue Base Address Register + Inbound Free Tail Pointer Register.

- When the queue is not empty (head and tail pointers are not equal), the data is returned and the MU increments the value in the Inbound Free Tail Pointer Register.
- When the queue is empty (head and tail pointers are equal), the value of -1 (FFFF FFFFH) is returned.

The i960 core processor may place messages in the Inbound Free Queue by writing the data to the local memory location pointed to by the head pointer. The local memory address is Queue Base Address Register + Inbound Free Head Pointer Register. The processor must then increment the Inbound Free Head Pointer Register.

17.4.3 Outbound Post Queue

The Outbound Post Queue holds outbound posted messages from the i960 core processor for other processors to process. This queue is read from the queue tail by external PCI agents. It is written to the queue head by the i960 core processor. The tail pointer is maintained by MU hardware. The head pointer is maintained by the i960 core processor.

For a PCI read transaction that accesses the Outbound Queue Port, the MU reads data at the local memory location pointed by the Outbound Post Tail Pointer. The local memory address is Queue Base Address Register + Outbound Post Tail Pointer Register.

- When the queue is not empty (head and tail pointers are not equal), the data is returned and the MU increments the value in the Outbound Post Tail Pointer Register.
- When the queue is empty (head and tail pointers are equal), the value of -1 (FFFF FFFFH) is returned.

A PCI interrupt is generated while the Outbound Post Head Pointer Register is not equal to the Outbound Post Tail Pointer Register. When they are equal, no interrupt is generated. The Outbound Post Queue Interrupt bit in the Outbound Interrupt Status Register indicates register comparison status and, therefore, interrupt status. The interrupt is cleared when the head and tail pointers become equal. This occurs when the external PCI agent reads enough queue entries to empty the queue. The interrupt can be masked by the Outbound Interrupt Mask Register.

The i960 core processor may place messages in the Outbound Post Queue by writing the data to the local memory location pointed to by the head pointer. The i960 core processor must then increment the Outbound Post Head Pointer Register.

17.4.4 Outbound Free Queue

The Outbound Free Queue holds free messages from other processors for the i960 core processor to use. This queue is read from the queue tail by the i960 core processor. It is written to the queue head by external PCI agents. The tail pointer is maintained by the i960 core processor. The head pointer is maintained by MU hardware.

For a PCI write transaction that accesses the Outbound Queue Port, the MU writes data to the local memory location pointed by the Outbound Free Head Pointer Register.

When data written to the Outbound Queue Port is written to local memory, MU hardware increments the Outbound Free Head Pointer Register.

When the head pointer and the tail pointer become equal and the queue is full, the MU signals an NMI interrupt to the i960 core processor. This interrupt is recorded in the Inbound Interrupt Status Register.

From the time that the PCI write transaction is received until the data is written in local memory and the Outbound Free Head Pointer Register is incremented, any PCI transaction that accesses the Inbound Queue Port is signalled a Retry.

The i960 core processor may read messages from the Outbound Free Queue by reading the data from the local memory location pointed to by the tail pointer. The processor must then increment the Outbound Free Tail Pointer Register.

Table 17-3. Circular Queue Summary

Queue Name	PCI Port	Generate PCI Interrupt?	Generate i960® Core Processor Interrupt?	Head Pointer maintained by	Tail Pointer maintained by
Inbound Post Queue	Inbound Queue Port	No	Yes, when queue is written	MU hardware	i960 core processor
Inbound Free Queue		No	No	i960 core processor	MU hardware
Outbound Post Queue	Outbound Queue Port	Yes, when queue is not empty	No	i960 core processor	MU hardware
Outbound Free Queue		No	Yes, when the queue overflows	MU hardware	i960 core processor

17.5 INDEX REGISTERS

The Index Registers are a set of 1004 registers that, when written by an external PCI agent, generate an interrupt to the i960 core processor. These registers are for inbound messages only. The interrupt is recorded in the Inbound Interrupt Status Register.

Index Register storage is allocated from i960 Rx I/O processor local memory. PCI write accesses to Index Registers write the data to local memory. PCI read accesses to Index Registers read the data from local memory. Local memory for Index Registers ranges from Primary Inbound ATU Translate Value Register + 050H to Primary Inbound ATU Translate Value Register + FFFH. CHAPTER 16, ADDRESS TRANSLATION UNIT describes how PCI addresses are translated to local memory addresses.

The first write access address is stored in the Index Address Register. This register is written during the earliest write access and provides a means to determine which Index Register was written. Once updated by the MU, the Index Address Register is not updated until the Inbound Interrupt Status Register's Index Register Interrupt bit is cleared. When the interrupt is cleared, the Index Address Register is re-enabled and stores the next Index Register write access address.

Writes by the i960 core processor to local memory used by the Index Registers do not cause an interrupt and do not update the Index Address Register.

17.6 APIC REGISTERS

The two APIC Registers are APIC Register Select Register and APIC Window Register. The APIC Register Select Register selects which APIC I/O Unit Register appears in the APIC Window Register. A write to the APIC Register Select Register generates an interrupt to the i960 core processor. APIC emulation software updates the APIC Window Register contents.

To prevent multiple accesses to APIC registers before APIC emulation software has a chance to update the register contents, the MU implements a hardware interlock for PCI accesses to APIC Registers.

When the APIC Register Select Register is written during a PCI transaction, the interlock is set and the i960 core processor is signaled an interrupt. All subsequent PCI accesses to either APIC Register are signalled a Retry until the interlock is cleared. The interlock must be cleared by software, by clearing the Inbound Interrupt Status Register's APIC Register Select Interrupt bit.

The same interlock mechanism applies to PCI writes to the APIC Window Register. When the APIC Window Register is written during a PCI transaction, the interlock is set and the i960 core processor is signaled an interrupt. The interlock is cleared by clearing the Inbound Interrupt Status Register's APIC Window Interrupt bit.

The interlock mechanism is enabled regardless of whether the APIC is enabled. Do not access either of the two APIC registers if the APIC is disabled, otherwise, the interrupt is never generated and the PCI bus deadlocks.

The interlock mechanism is enabled only when the APIC unit is enabled by setting the APIC bus interface enable bit in the APIC Control Status Register.

The i960 core processor must clear the interrupt bit to allow a retried PCI master to complete its transaction.

17.7 REGISTER DEFINITIONS

Figure 17-2 shows the PCI memory map and identifies the first 4 Kbytes of ATU Primary Inbound PCI address space. Registers in Table 17-4 are located in primary PCI address space and Peripheral Memory-Mapped Register (PMMR) address space. They are accessible through primary PCI bus transactions and i960 core processor bus accesses. In primary PCI address space, they are mapped into the first 80 bytes of the Primary ATU's primary inbound address window.

First 4 Kbytes of the ATU Primary Inbound PCI Address Space

0000H	APIC Register Select Register	2 APIC Registers
0004H	reserved	
0008H	APIC Window Register	
000CH	reserved	
0010H	Inbound Message Register 0	4 Message Registers
0014H	Inbound Message Register 1	
0018H	Outbound Message Register 0	
001CH	Outbound Message Register 1	
0020H	Inbound Doorbell Register	2 Doorbell Registers and 4 Interrupt Registers
0024H	Inbound Interrupt Status Register	
0028H	Inbound Interrupt Mask Register	
002CH	Outbound Doorbell Register	
0030H	Outbound Interrupt Status Register	
0034H	Outbound Interrupt Mask Register	
0038H	reserved	
003CH	reserved	
0040H	Inbound Queue Port	2 Queue Ports
0044H	Outbound Queue Port	
0048H	reserved	
004CH	reserved	
0050H 0FFCH	i960® Rx Processor Local Memory	1004 Index Registers

Figure 17-2. PCI Memory Map

Registers in Table 17-4 are located in Peripheral Memory-Mapped Register (PMMR) address space as described in APPENDIX C, MEMORY-MAPPED REGISTERS. Reading or writing a register that is reserved is undefined.

Table 17-4. Peripheral Memory-Mapped Register Summary

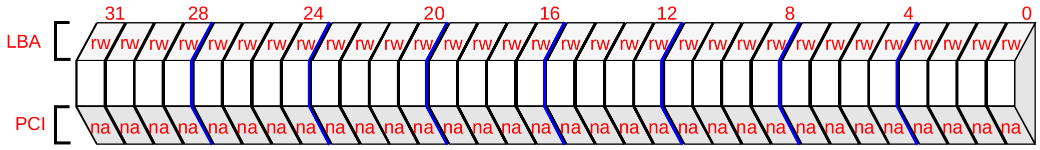
Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
17.7.1	APIC Register Select Register - ARSR	17-15	32	0000 1300H	NA
17.7.2	APIC Window Register - AWR	17-15	32	0000 1308H	NA
17.7.3	Inbound Message Registers - IMRx	17-16	32	0 - 0000 1310H 1 - 0000 1314H	NA
17.7.4	Outbound Message Registers - OMRx	17-17	32	0 - 0000 1318H 1- 0000 131CH	NA
17.7.5	Inbound Doorbell Register - IDR	17-18	32	0000 1320H	NA
17.7.6	Inbound Interrupt Status Register - IISR	17-19	32	0000 1324H	NA
17.7.7	Inbound Interrupt Mask Register - IIMR	17-20	32	0000 1328H	NA
17.7.8	Outbound Doorbell Register - ODR	17-22	32	0000 132CH	NA
17.7.9	Outbound Interrupt Status Register - OISR	17-23	32	0000 1330H	NA
17.7.10	Outbound Interrupt Mask Register - OIMR	17-24	32	0000 1334H	NA
17.7.11	Messaging Unit Configuration Register - MUCR	17-26	32	0000 1350H	NA
17.7.12	Queue Base Address Register - QBAR	17-27	32	0000 1354H	NA
17.7.13	Inbound Free Head Pointer Register - IFHPR	17-28	32	0000 1360H	NA
17.7.14	Inbound Free Tail Pointer Register - IFTPR	17-29	32	0000 1364H	NA
17.7.15	Inbound Post Head Pointer Register - IPHPR	17-30	32	0000 1368H	NA
17.7.16	Inbound Post Tail Pointer Register - IPTPR	17-31	32	0000 136CH	NA
17.7.17	Outbound Free Head Pointer Register - OFHPR	17-32	32	0000 1370H	NA
17.7.18	Outbound Free Tail Pointer Register - OFTPR	17-33	32	0000 1374H	NA
17.7.19	Outbound Post Head Pointer Register - OPHPR	17-34	32	0000 1378H	NA
17.7.20	Outbound Post Tail Pointer Register - OPTPR	17-35	32	0000 137CH	NA
17.7.21	Index Address Register - IAR	17-36	32	0000 1380H	NA

17.7.1 APIC Register Select Register - ARSR

The APIC Register Select Register (ARSR) selects the I/O APIC Register which appears in the APIC Window Register. A write to the APIC Register Select Register generates an interrupt to the i960 core processor.

When the APIC Register Select Register is written, an interlock is set and the i960 core processor is signaled an interrupt. All subsequent PCI accesses to this register are signalled a Retry until the interlock is cleared. The interlock is cleared by clearing the APIC Register Select Interrupt bit in the Inbound Interrupt Status Register. The interlock is disabled/enabled by the APIC Bus Interface Enable bit in the APIC Control / Status Register.

Table 17-5. APIC Register Select Register - ARSR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1300H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write
			
			PCI = PCI Configuration Address Offset

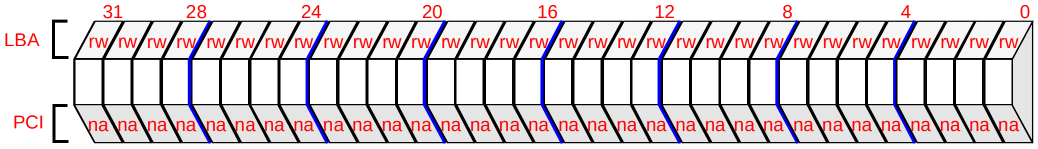
Bit	Default	Description
31:00	0000 0000H	Register Select - contains the I/O APIC Register address.

17.7.2 APIC Window Register - AWR

The APIC Window Register (AWR) contains the APIC I/O Unit Register value selected by the APIC Register Select Register. I/O APIC emulation software is responsible for reading or writing this register after the APIC Register Select Register is written. A write to the APIC Window Register causes an interrupt to be generated to the i960 core processor.

When the APIC Window Register is written, an interlock is set and the i960 core processor is signaled an interrupt. All subsequent PCI accesses to this register are signalled a Retry until the interlock is cleared. The interlock is cleared by clearing the APIC Window Interrupt bit in the Inbound Interrupt Status Register. The interlock is disabled/enabled by the APIC Bus Interface Enable bit in the APIC Control / Status Register.

Table 17-6. APIC Window Register - AWR


31 28 24 20 16 12 8 4 0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

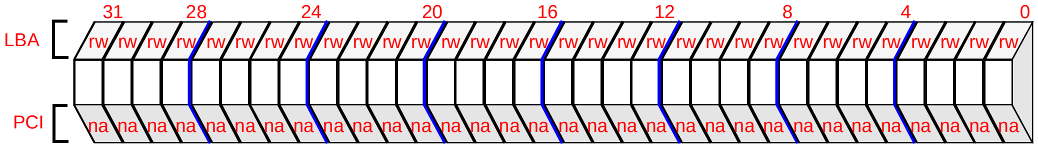
LBA: 1308H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	Window - contains I/O APIC Register contents.

17.7.3 Inbound Message Registers - IMRx

The two Inbound Message Registers are IMR0 and IMR1. When IMR registers are written, an interrupt to the i960 core processor is generated. The interrupt is recorded in the Inbound Interrupt Status Register and may be masked by the Inbound Interrupt Mask Register’s Inbound Message Interrupt Mask bit.

Table 17-7. Inbound Message Register - IMRx


31 28 24 20 16 12 8 4 0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

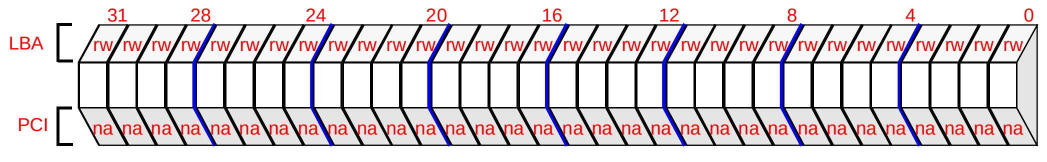
LBA: CH. 0 = 1310H
CH. 1 = 1314H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	Inbound Message - This 32-bit message is written by an external PCI agent. When written, an interrupt to the i960 core processor is generated.

17.7.4 Outbound Message Registers - OMRx

The two Outbound Message Registers are OMR0 and OMR1. When an OMR register is written, a PCI interrupt is generated. The interrupt is recorded in the Outbound Interrupt Status Register and may be masked by the Outbound Message Interrupt Mask bit in the Outbound Interrupt Mask Register.

Table 17-8. Outbound Message Register - OMRx


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

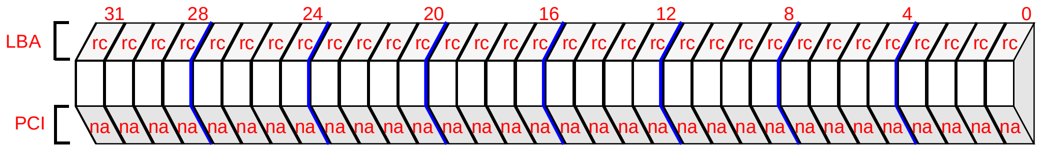
LBA:
 	
PCI:
	CH. 0 = 1318H
	CH. 1 = 131CH
	NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:00	0000 0000H	Outbound Message - This is 32-bit message written by the i960 core processor. When written, an interrupt is generated on the PCI Interrupt pin determined by the ATU Interrupt Pin Register.

17.7.5 Inbound Doorbell Register - IDR

The Inbound Doorbell Register (IDR) is used to generate interrupts to the i960 core processor. Bit 31 is reserved for generating an NMI interrupt. When bit 31 is set, an NMI interrupt is generated to the NMI interrupt latch. All other bits, when set, cause the i960 core processor's XINT7 interrupt line to assert from the XINT7 interrupt latch, when the interrupt is not masked by the Inbound Interrupt Mask Register's Inbound Doorbell Interrupt Mask bit. IDR register bits can only be set by an external PCI agent and can only be cleared by the i960 core processor. Refer to Section 8.3.3, Internal Peripheral Interrupt Routing (pg. 8-26).

Table 17-9. Inbound Doorbell Register - IDR


31
28
24
20
16
12
8
4
0
LBA
rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc rc
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

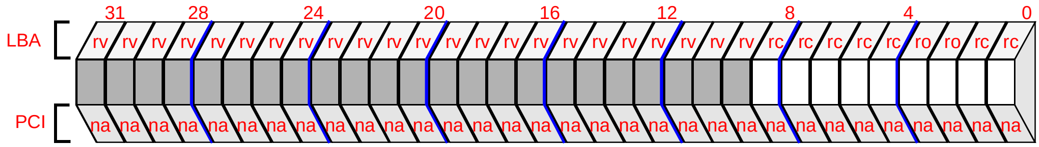
LBA: 1320H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31	0_2	NMI Interrupt - Generate an NMI Interrupt to the i960 core processor.
30:00	0000 000H	XINT7 Interrupt - When any bit is set, generate an XINT7 interrupt to the i960 core processor. When all bits are clear, do not generate an XINT7 interrupt.

17.7.6 Inbound Interrupt Status Register - IISR

The Inbound Interrupt Status Register (IISR) contains hardware interrupt status. It records the status of i960 core processor interrupts generated by the Message Registers, Doorbell Registers, and the Circular Queues. All interrupts are routed to the i960 core processor's XINT7 interrupt input, except for the NMI Doorbell Interrupt and the Outbound Free Queue Overflow interrupt; these two are routed to the NMI interrupt input. The generation of interrupts recorded in the Inbound Interrupt Status Register may be masked by setting the corresponding bit in the Inbound Interrupt Mask Register. Some bits in this register are Read Only. For those bits, the interrupt must be cleared through another register.

Table 17-10. Inbound Interrupt Status Register - IISR (Sheet 1 of 2)

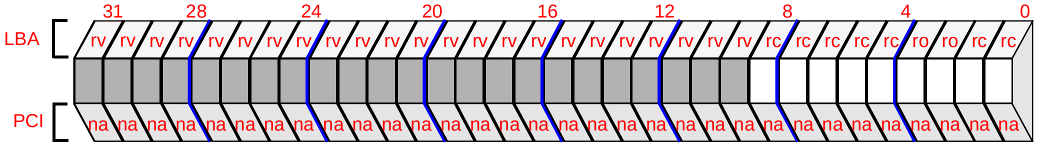

31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc ro ro rc rc
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1324H
PCI: NA

Legend:
NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:09	0000 00H	Reserved.
08	0_2	APIC Window Interrupt - set by MU hardware when the APIC Window Register is written by a PCI transaction.
07	0_2	APIC Register Select Interrupt - set by MU hardware when the APIC Register Select Register is written by a PCI transaction.
06	0_2	Index Register Interrupt - set by MU hardware when an Index Register is written by a PCI transaction.
05	0_2	Outbound Free Queue Overflow Interrupt - set when the Outbound Free Head Pointer becomes equal to the Tail Pointer and the queue is full. An NMI interrupt is generated for this condition.
04	0_2	Inbound Post Queue Interrupt - set by MU hardware when the Inbound Post Queue has been written.
03	0_2	NMI Doorbell Interrupt - set when the Inbound Doorbell Register NMI Interrupt is set. To clear this bit (and the interrupt), the Inbound Doorbell Register NMI Interrupt bit in the Inbound Doorbell Register must be clear.
02	0_2	Inbound Doorbell Interrupt - set when at least one XINT7 Interrupt bit in the Inbound Doorbell Register is set. To clear this bit (and the interrupt), the XINT7 Interrupt bits in the Inbound Doorbell Register must all be clear.
01	0_2	Inbound Message 1 Interrupt - set when the Inbound Message 1 Register has been written.

Table 17-10. Inbound Interrupt Status Register - IISR (Sheet 2 of 2)


31 28 24 20 16 12 8 4 0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc ro ro rc rc
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1324H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

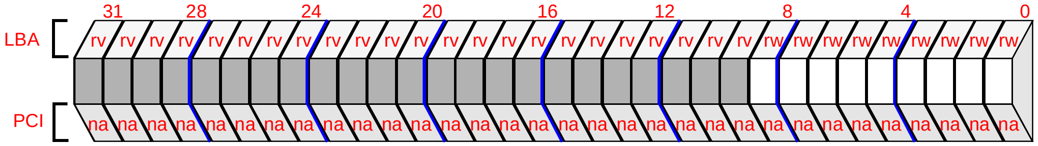
Bit	**Default**	**Description**
00	0_2	Inbound Message 0 Interrupt - set when the Inbound Message 0 Register has been written.

17.7.7 Inbound Interrupt Mask Register - IIMR

The Inbound Interrupt Mask Register (IIMR) provides the ability to mask i960 core processor interrupts that the MU generates. Each Mask register bit corresponds to an interrupt bit in the Inbound Interrupt Status Register.

Setting or clearing bits in this register does not affect the Inbound Interrupt Status Register. They only affect i960 core processor interrupt generation.

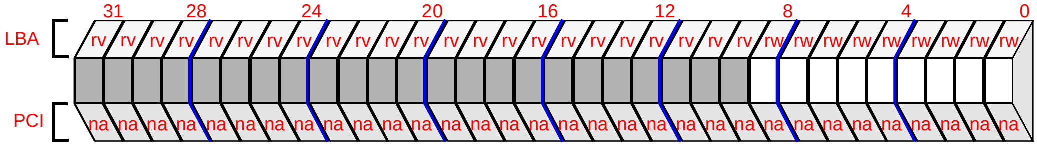
Table 17-11. Inbound Interrupt Mask Register - IIMR (Sheet 1 of 2)


31 28 24 20 16 12 8 4 0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1328H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:09	0000 00H	Reserved.
08	0_2	APIC Window Interrupt Mask - When set, this bit masks the interrupt generated by MU hardware when the APIC Window Register is written to by a PCI transaction.
07	0_2	APIC Register Select Interrupt Mask - When set this bit masks the interrupt generated by MU hardware when the APIC Register Select Register is written to by a PCI transaction.

Table 17-11. Inbound Interrupt Mask Register - IIMR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1328H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

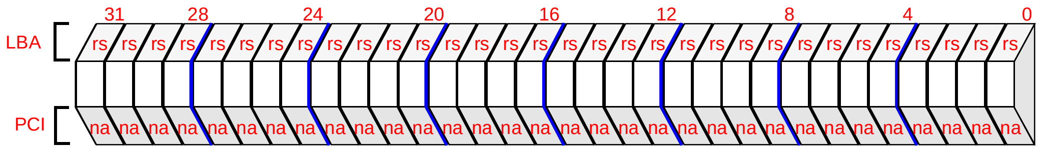
Bit	Default	Description
06	0_2	Index Register Interrupt Mask - When set, this bit masks the interrupt generated by MU hardware when an Index Register has been written after a PCI transaction.
05	0_2	Outbound Free Queue Overflow Interrupt Mask - When set, this bit masks the NMI interrupt generated when the Outbound Free Head Pointer becomes equal to the Tail Pointer and the queue is full.
04	0_2	Inbound Post Queue Interrupt Mask - When set, this bit masks the interrupt generated by MU hardware when the Inbound Post Queue has been written.
03	0_2	NMI Doorbell Interrupt Mask - When set, this bit masks the NMI Interrupt when the Inbound Doorbell Register NMI Interrupt bit is set.
02	0_2	Inbound Doorbell Interrupt Mask - When set, this bit masks the interrupt generated when at least one XINT7 Interrupt bit in the Inbound Doorbell Register is set.
01	0_2	Inbound Message 1 Interrupt Mask - When set, this bit masks the Inbound Message 0 Interrupt generated by a write to the Inbound Message 0 Register.
00	0_2	Inbound Message 0 Interrupt Mask - When set, this bit masks the Inbound Message 0 Interrupt generated by a write to the Inbound Message 0 Register.

17.7.8 Outbound Doorbell Register - ODR

The Outbound Doorbell Register (ODR) allows software interrupt generation. It allows the i960 core processor to generate PCI interrupts to the host processor by writing to the Software Interrupt bits or to a specific PCI interrupt bit. PCI interrupt generation through the Outbound Doorbell Register may be masked by setting the Outbound Doorbell Interrupt Mask bit in the Outbound Interrupt Mask Register.

Software Interrupt bits in this register can only be set by the i960 core processor and can only be cleared by an external PCI agent.

Table 17-12. Outbound Doorbell Register - ODR


LBA
PCI
31
28
24
20
16
12
8
4
0
rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs rs
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 132CH
PCI: NA

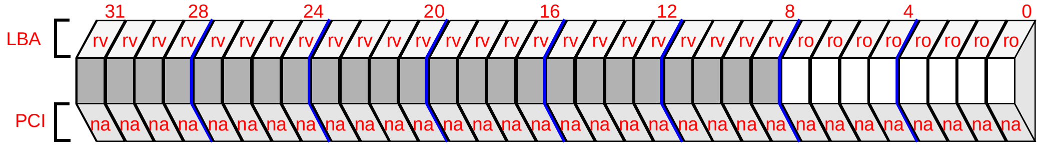
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31	0_2	PCI Interrupt D - When set, this bit causes P_INTD# to assert. When cleared, P_INTD# deasserts.
30	0_2	PCI Interrupt C - When set, this bit causes P_INTC# to assert. When cleared, P_INTC# deasserts.
29	0_2	PCI Interrupt B- When set, this bit causes P_INTB# to assert. When cleared, P_INTB# deasserts.
28	0_2	PCI Interrupt A- When set, this bit causes P_INTA# to assert. When cleared, P_INTA# deasserts.
27:00	0000 000H	Software Interrupt - When any bit is set, generate a PCI interrupt. The PCI interrupt pin used is determined by the ATU Interrupt Pin Register. When all bits are clear, do not generate a PCI interrupt.

17.7.9 Outbound Interrupt Status Register - OISR

The Outbound Interrupt Status Register (OISR) contains hardware interrupt status. It records the status of PCI interrupts generated by the Message Registers, Doorbell Registers, and the Circular Queues. All interrupts are routed to the PCI interrupt pin selected by the ATU Interrupt Pin Register (ATUIPR), except the PCI Interrupt “X” interrupts which are individually routed. The PCI interrupt generation recorded in the Outbound Interrupt Status Register may be masked by setting the corresponding bit in the Outbound Interrupt Mask Register. Some bits in this register are Read Only; for these bits, the interrupt must be cleared through another register.

Table 17-13. Outbound Interrupt Status Register - OISR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro ro ro ro ro
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1330H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

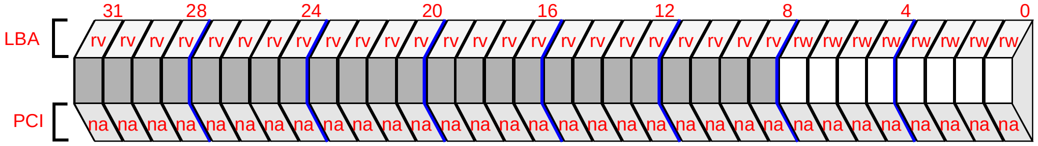
Bit	**Default**	**Description**
31:08	0000 00H	Reserved.
07	0_2	PCI Interrupt D - set when the PCI Interrupt D bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt D bit in the Outbound Doorbell Register must be cleared.
06	0_2	PCI Interrupt C - set when the PCI Interrupt C bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt C bit in the Outbound Doorbell Register must be cleared.
05	0_2	PCI Interrupt B - set when the PCI Interrupt B bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt B bit in the Outbound Doorbell Register must be cleared.
04	0_2	PCI Interrupt A - set when the PCI Interrupt A bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt A bit in the Outbound Doorbell Register must be cleared.
03	0_2	Outbound Post Queue Interrupt - set when the Outbound Post Head Pointer Register does *not* equal the Outbound Post Tail Pointer Register. This bit is cleared when the Outbound Post Head Pointer Register equals the Outbound Post Tail Pointer Register.
02	0_2	Outbound Doorbell Interrupt - set when at least one Software Interrupt bit in the Outbound Doorbell Register is set. To clear this bit (and the interrupt), Software Interrupt bits in the Outbound Doorbell Register must all be clear.
01	0_2	Outbound Message 1 Interrupt - set by the MU when the Outbound Message 1 Register is written. Clearing this bit clears the interrupt.
00	0_2	Outbound Message 0 Interrupt - set by the MU when the Outbound Message 0 Register is written. Clearing this bit clears the interrupt.

17.7.10 Outbound Interrupt Mask Register - OIMR

The Outbound Interrupt Mask Register (OIMR) provides the ability to mask outbound PCI interrupts that the MU generates. Each mask register bit corresponds to a hardware interrupt bit in the Outbound Interrupt Status Register. When the bit is set, the PCI interrupt is not generated. When the bit is clear, the interrupt is allowed to be generated.

Setting or clearing bits in this register does not affect the Outbound Interrupt Status Register; they only affect PCI interrupt generation.

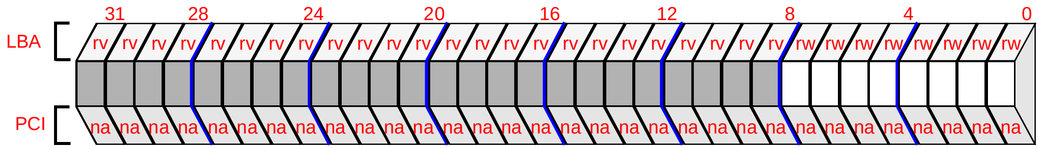
Table 17-14. Outbound Interrupt Mask Register - OIMR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1334H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:08	0000 00H	Reserved.
07	0_2	PCI Interrupt D Mask - When set, this bit masks the PCI Interrupt D signal when the PCI Interrupt D bit in the in the Outbound Doorbell Register is set. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated
06	0_2	PCI Interrupt C Mask - When set, this bit masks the PCI Interrupt C signal when the PCI Interrupt C bit in the in the Outbound Doorbell Register is set. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated
05	0_2	PCI Interrupt B Mask - When set, this bit masks the PCI Interrupt B signal when the PCI Interrupt B bit in the in the Outbound Doorbell Register is set. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated
04	0_2	PCI Interrupt A Mask - When set, this bit masks the PCI Interrupt A signal when the PCI Interrupt A bit in the in the Outbound Doorbell Register is set. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated
03	0_2	Outbound Post Queue Interrupt Mask - When set, this bit masks the PCI interrupt generated when the Outbound Post Head Pointer Register does *not* equal the Outbound Post Tail Pointer Register. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated

Table 17-14. Outbound Interrupt Mask Register - OIMR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

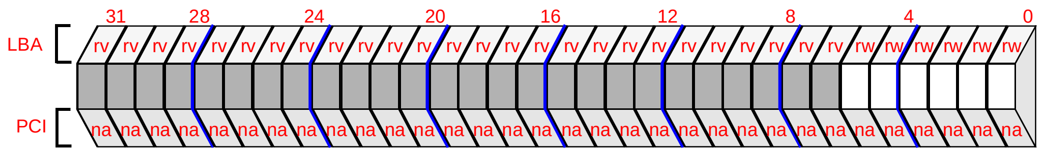
LBA: 1334H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
02	0_2	Outbound Doorbell Interrupt Mask - When set, this bit masks the Software Interrupt generated by the Outbound Doorbell Register. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated
01	0_2	Outbound Message 1 Interrupt Mask - When set, this bit masks the Outbound Message 1 Interrupt generated by a write to the Outbound Message 1 Register. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated
00	0_2	Outbound Message 0 Interrupt Mask- When set, this bit masks the Outbound Message 0 Interrupt generated by a write to the Outbound Message 0 Register. 0 - allow interrupt to be generated 1 - do not allow interrupt to be generated

17.7.11 Messaging Unit Configuration Register - MUCR

The Messaging Unit Configuration Register (MUCR) contains Circular Queue Enable bit and the size of one of the four Circular Queues. The Circular Queues are disabled at reset to allow the software to initialize the head and tail pointer registers before any PCI accesses to the Queue Ports occur. Each Circular Queue may range from 4 Kbyte entries (16 Kbytes) to 64 Kbyte entries (256 Kbytes).

Table 17-15. Messaging Unit Configuration Register - MUCR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1350H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

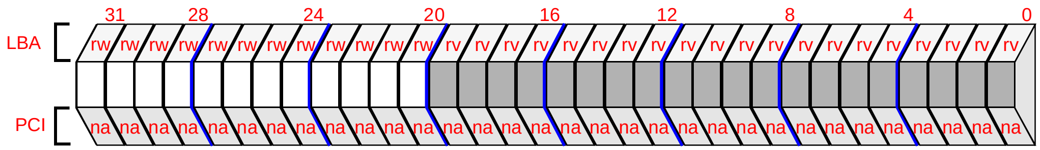
Bit	Default	Description
31:05	0000 00H	Reserved.
00	0_2	Circular Queue Enable - This bit enables or disables the Circular Queues. When clear, Circular Queues are disabled. The MU accepts PCI accesses to the Circular Queue Ports; the MU ignores data for Writes and returns FFFF FFFFH for Reads. When set, Circular Queues are enabled.
05:01	1H	Circular Queue Size - This field determines the size of each Circular Queue. All four queues are the same size. Circular Queue Size: • (00001_2) 4K entries (16 Kbytes) • (00010_2) 8K entries (32 Kbytes) • (00100_2) 16K entries (64 Kbytes) • (01000_2) 32K entries (128 Kbytes) • (10000_2) 64K entries (256 Kbytes)

17.7.12 Queue Base Address Register - QBAR

The Queue Base Address Register (QBAR) contains the Circular Queue local memory address. The base address must be located on a 1 Mbyte address boundary.

All Circular Queue head and tail pointers are based on the QBAR. Writing to the upper 12 bits of the head and tail pointer register does not affect the Queue Base Address in the QBAR.

Table 17-16. Queue Base Address Register - QBAR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1354H
PCI: NA

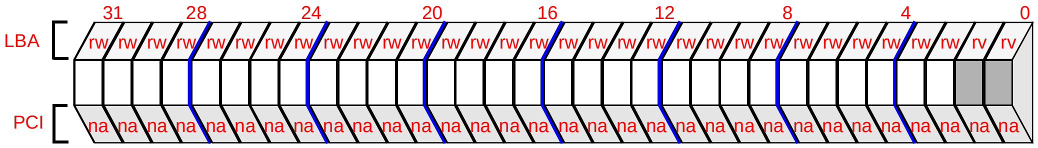
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - circular queue local memory address.
19:00	0 0000H	Reserved.

17.7.13 Inbound Free Head Pointer Register - IFHPR

The Inbound Free Head Pointer Register (IFHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Inbound Free Queue. The Head Pointer must be aligned on a word address boundary. This register is maintained by software. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-17. Inbound Free Head Pointer Register - IFHPR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1360H

PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - local memory address of the Circular Queues.
19:02	0000 0H	Inbound Free Head Pointer - Local memory offset of the head pointer for the Inbound Free Queue.
01:00	00_2	Reserved.

17.7.14 Inbound Free Tail Pointer Register - IFTPR

The Inbound Free Tail Pointer Register (IFTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Inbound Free Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-18. Inbound Free Tail Pointer Register - IFTPR

	31			28				24				20				16				12				8				4				0
LBA	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rv	rv
PCI	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

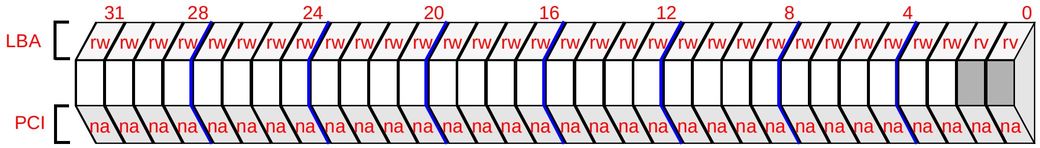
LBA: 1364H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - local memory address of the Circular Queues.
19:02	0000 0H	Inbound Free Tail Pointer - Local memory offset of the tail pointer for the Inbound Free Queue.
01:00	00_2	Reserved.

17.7.15 Inbound Post Head Pointer Register - IPHPR

The Inbound Post Head Pointer Register (IPHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Inbound Post Queue. The Head Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-19. Inbound Post Head Pointer Register - IPHPR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

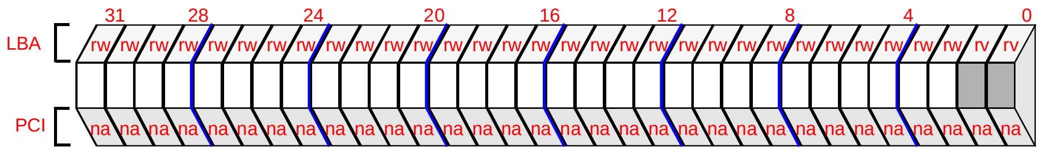
LBA: 1368H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:20	000H	Reserved
19:02	0000 0H	Inbound Post Head Pointer - Local memory offset of the head pointer for the Inbound Post Queue.
01:00	00_2	Reserved.

17.7.16 Inbound Post Tail Pointer Register - IPTPR

The Inbound Post Tail Pointer Register (IPTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Inbound Post Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-20. Inbound Post Tail Pointer Register - IPTPR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

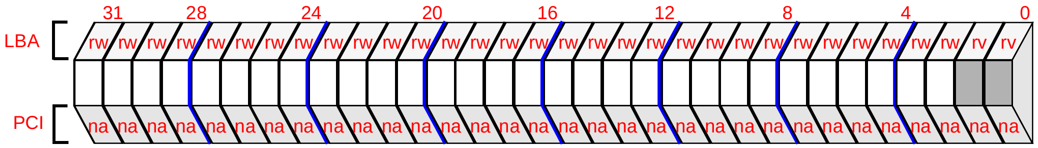
LBA: 136CH
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Reserved
19:02	0000 0H	Inbound Post Tail Pointer - Local memory offset of the tail pointer for the Inbound Post Queue.
01:00	00_2	Reserved.

17.7.17 Outbound Free Head Pointer Register - OFHPR

The Outbound Free Head Pointer Register (OFHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Outbound Free Queue. The Head Pointer must be aligned on a word address boundary. This register is maintained by software. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-21. Outbound Free Head Pointer Register - OFHPR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1370H
PCI: NA

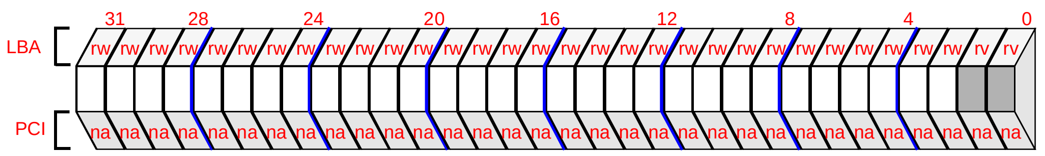
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - local memory address of the Circular Queues.
19:02	0000 0H	Outbound Free Head Pointer - Local memory offset of the head pointer for the Outbound Free Queue.
01:00	00_2	Reserved.

17.7.18 Outbound Free Tail Pointer Register - OFTPR

The Outbound Free Tail Pointer Register (OFTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Outbound Free Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-22. Outbound Free Tail Pointer Register - OFTPR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

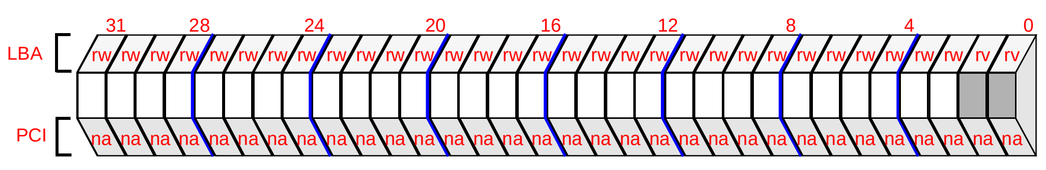
LBA: 1374H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write
			
			PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - local memory address of the Circular Queues.
19:02	0000 0H	Outbound Free Tail Pointer - Local memory offset of the tail pointer for the Outbound Free Queue.
01:00	00_2	Reserved.

17.7.19 Outbound Post Head Pointer Register - OPHPR

The Outbound Post Head Pointer Register (OPHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Outbound Post Queue. The Head Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of this register.

Table 17-23. Outbound Post Head Pointer Register - OPHPR


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1378H
PCI: NA

Legend:	NA = Not Accessible	RO = Read Only
RV = Reserved	PR = Preserved	RW = Read/Write
RS = Read/Set	RC = Read Clear	

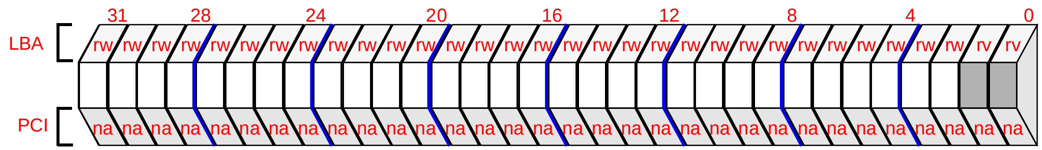
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - local memory address of the Circular Queues.
19:02	000 0H	Outbound Post Head Pointer - Local memory offset of the head pointer for the Outbound Post Queue.
01:00	00_2	Reserved.

17.7.20 Outbound Post Tail Pointer Register - OPTPR

The Outbound Post Tail Pointer Register (OPTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Outbound Post Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of this register.

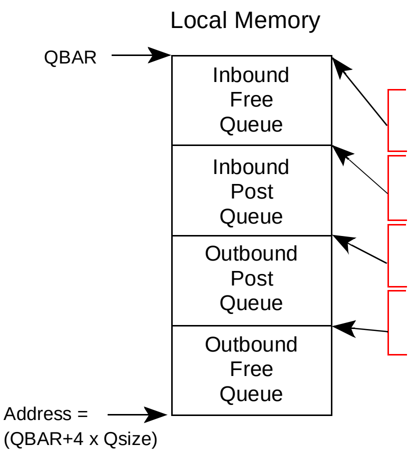
Table 17-24. Outbound Post Tail Pointer Register - OPTPR


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 137CH
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:20	000H	Queue Base Address - local memory address of the Circular Queues.
19:02	0000 0H	Outbound Post Tail Pointer - Local memory offset of the tail pointer for the Outbound Post Queue.
01:00	00_2	Reserved.



Queue Offset Registers	4K Entries Qsize	8K Entries Qsize	16K Entries Qsize	32K Entries Qsize	64K Entries Qsize
IFHPR IFTPR	0	0	0	0	0
IPHPR IPTPR	4000H	8000H	10000H	20000H	40000H
OPHPR OPTPR	8000H	10000H	20000H	40000H	80000H
OFHPR OFTPR	C000H	18000H	30000H	60000H	C0000H

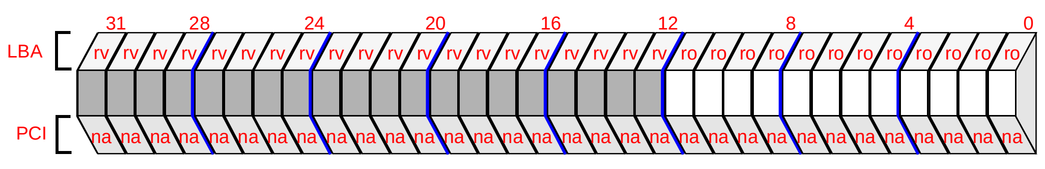
Figure 17-3. Initialization Values Programmed by Software

17.7.21 Index Address Register - IAR

The Index Address Register (IAR) contains the offset of the least recently accessed Index Register. The MU writes to this register when the Index Registers are written by a PCI agent. The register is not updated until the Inbound Interrupt Status Register’s Index Interrupt bit is cleared.

The local memory address of the Index Register least recently accessed is computed by adding the Index Address Register to the Primary Inbound ATU Translate Value Register.

Table 17-25. Index Address Register - IAR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro ro ro ro ro ro ro ro ro
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1380H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:12	0000 0H	Reserved.
11:00	000H	Index Address - contains the local memory offset of the Index Register written (050H to FFCH)

18

BUS ARBITRATION

CHAPTER 18
BUS ARBITRATION

This chapter describes the bus arbitration units of the i960® Rx I/O processor. The four arbitration units include: the internal local bus arbiter, the secondary PCI bus arbiter for external secondary PCI bus masters, the primary PCI bus arbiter for internal PCI bus masters, and the secondary PCI bus arbiter for internal secondary PCI bus masters.

Some of the other topics discussed in this chapter include: the priority mechanism used in all of the arbitration units, the memory-mapped registers used in programming the arbitration units, local bus backoff, secondary bus arbitration parking.

CHAPTER 1, INTRODUCTION contains a block diagram of the i960 Rx I/O processor in Figure 1-2, which shows the four arbitration units.

18.1 OVERVIEW

The i960 Rx I/O processor requires an arbitration mechanism to control 80960 local bus ownership. Bus masters connected to the local bus consist of:

- Three DMA channels
- Primary PCI address translation unit
- Secondary PCI address translation unit
- i960 core processor
- External bus masters

The local bus arbitration unit is responsible for granting the local bus to a bus master. There is a programmable 12-bit counter to limit the amount of time a bus master has control of the local bus and to dictate when a bus master must relinquish ownership when other bus masters are requesting the local bus.

The secondary PCI bus arbiter supports:

- Six external PCI bus masters
- Current owner of the i960 Rx I/O processor secondary bus interface, granted by the internal secondary PCI bus arbiter

The PCI local bus specification defines the handshaking protocol for each bus master when granted ownership of the secondary PCI bus by the secondary PCI bus arbiter.

In addition to the local bus arbiter and the secondary PCI bus arbiter, the i960 Rx I/O processor contains two internal PCI arbitration units. The primary internal arbitration unit controls access to the internal primary PCI bus. Arbitration occurs for the primary PCI bus between the primary ATU, DMA channels 0 and 1, and the primary interface of the PCI-to-PCI bridge unit. The secondary internal arbitration unit controls access to the internal secondary PCI bus. Arbitration occurs for the secondary PCI bus between the secondary ATU, DMA channel 2, and the secondary interface of the PCI-to-PCI bridge unit. Both internal PCI arbitration units function in a similar manner. The internal arbiters are not programmable.

18.2 LOCAL BUS ARBITRATION UNIT

The 80960 local bus arbitration unit supports up to seven local bus masters. Table 18-1 shows the seven bus masters. Each master can be disabled or programmed to one of three priority levels. The Local Bus Arbitration Control Register (LBACR), programmed by application software, sets the priorities for each of the bus masters. Each priority level uses a round-robin algorithm to guarantee that each device has a chance at bus ownership. When one device has finished, the next device, assuming one is currently requesting the bus, is granted ownership.

When a bus master requests the local bus, the arbiter first obtains control of the local bus from the i960 core processor or the current bus owner, based on the programmed priority and the current local bus arbitration latency counter value. The arbiter then grants the local bus to the requesting bus master by returning the respective internal GNT# signal.

When there are no masters requesting the local bus, the local bus arbiter parks the local bus with the i960 core processor. The local bus arbitration latency counter is reset each time a master is granted the local bus, with the exception of when the bus is being parked. When the arbiter parks the bus by granting ownership to the processor, the local bus arbitration latency counter is not reset.

Table 18-1. Local Bus Masters

Bus Master
i960 Core Processor (Parked Master)
DMA Channel 0
DMA Channel 1
DMA Channel 2
Primary ATU (for inbound transactions)
Secondary ATU (for inbound transactions)
External Local Bus Device

The initial priority for each bus master is programmable by software. While running, the arbiter promotes and demotes the bus masters using the round robin scheme shown in Figure 18-1. After a device relinquishes control of the bus, it returns to its initial programmed priority. Table 18-2 shows the 2-bit values that correspond to each priority level.

Table 18-2. Programmed Priority Control

2-Bit Programmed Value	Priority Level
00_2	High Priority
01_2	Medium Priority
10_2	Low Priority
11_2	Disabled

The arbitration scheme supports three levels of round-robin arbitration. The three levels define a low, medium and high priority. Using the round-robin mechanism ensures there is a winner for each priority level. To enforce the concept of fairness, a slot is reserved for the winner of each priority level (except the highest) in the next highest priority. When the winner of a priority level is not granted the bus during that particular arbitration sequence, it is promoted to the next highest level of priority. Once its bus ownership is removed, the device is reset to its initially programmed priority and may start arbitration once again. Figure 18-1 and Table 18-3 show the three priority levels and the reserved slots for the promoted requestor.

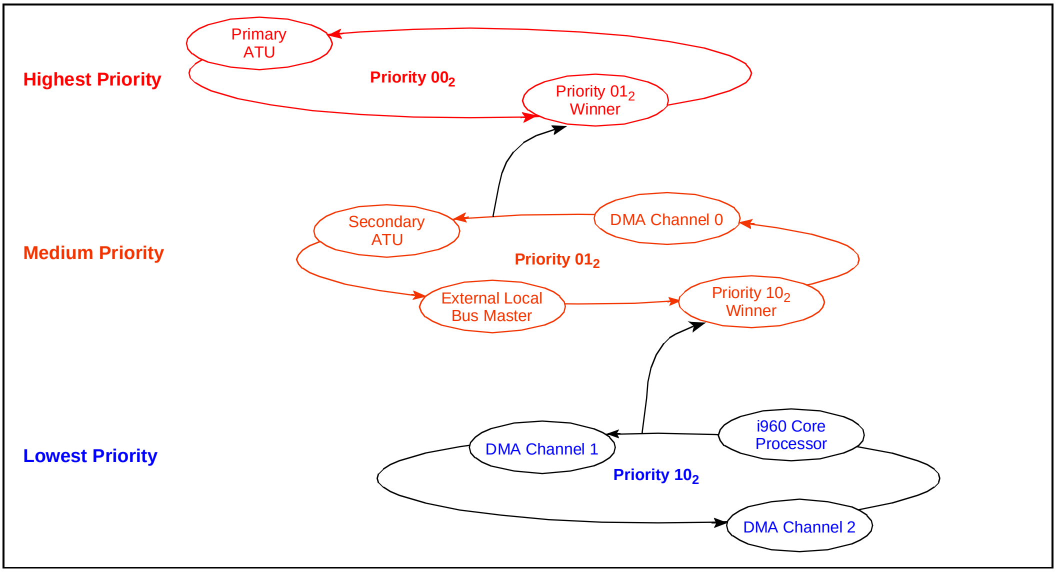


Figure 18-1. Local Bus Arbitration Example

Table 18-3. Priority Programming for Local Bus Arbitration Example

Bus Master	Programmed Priority
Primary ATU	High - 00_2
Secondary ATU	Medium - 01_2
DMA Channel 0	Medium - 01_2
External Bus Master	Medium - 01_2
DMA Channel 1	Low - 10_2
DMA Channel 2	Low - 10_2
i960 Core Processor	Low - 10_2

Table 18-4 is an example of bus arbitration, with three bus masters. Each of the bus masters is constantly requesting the bus, and each is at a different priority level. The top row of the table lists the current bus master/winner of the highest priority group. The three rows labeled as high, medium and low represent the actual priority levels that devices are currently at based on either their initial programmed priority or promotion through the levels. For example, device C starts out at low priority. Because it is the only device at this priority, it is the winner at low priority and is promoted to medium priority. Later it wins at medium priority (against device B) and is promoted to high priority where it wins the level (against device A) and the bus. Device C is then put back at its programmed priority of low and starts the whole cycle over.

Table 18-4. Bus Arbitration Example – Three Bus Masters

Priority Level	Initial State	Winning Bus Master							
		A	B	A	C	A	B	A	C
High	A	B	A	C	A	B	A	C	A
Medium	B	C	C	B	B	–	C	B	B
Low	C	–	–	–	–	C	–	–	–

NOTE: In this example, all bus masters are continually requesting the bus.

The winning bus master pattern for the bus arbitration example in Table 18-4 would continue on as follows: **ABACABACABACABAC**.

Table 18-5 is an example of bus arbitration, with six bus masters. Each of the bus masters is constantly requesting the bus, and there are two masters programmed at each different priority level. The top row of the table lists the current bus master/winner of the highest priority group. The three rows labeled as high, medium and low represent the actual priority levels that devices are currently at based on either their initial programmed priority or promotion through the levels.

Table 18-5. Bus Arbitration Example – Six Bus Masters

Priority Level	Initial State	Winning Bus Master								
		A	B	C	A	B	D	A	B	E
High	AB	BC	AC	AB	BD	AD	AB	BE	AE	AB
Medium	CD	DE	DE	DE	CE	CE	CE	CDF	CDF	CDF
Low	EF	F	F	F	F	F	F	–	–	–

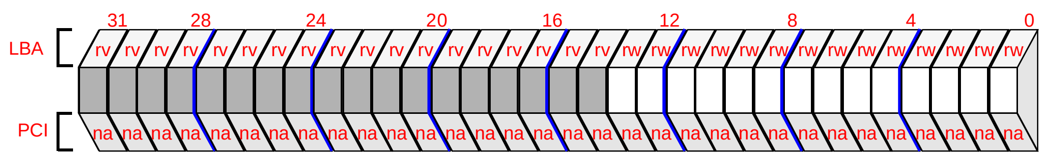
NOTE: In this example, all bus masters are continually requesting the bus.

The winning bus master pattern for the bus arbitration example in Table 18-5 would continue on as follows: **ABCABDABFABCABDABEABCABDABF**.

18.2.1 Local Bus Arbitration Control Register - LBACR

The Local Bus Arbitration Control Register (LBACR - Table 18-6) sets the arbitration priority of each device that uses the local bus. This register is accessible only from the 80960 processor bus. Each device is given a 2-bit priority. At reset, all devices default to 00_2, high priority, which results in a simple round-robin for all local bus masters. As devices are promoted up through the priority levels in the internal arbitration scheme, the LBACR does not change to reflect the current priority of a device. It always contains the device's programmed priority.

Table 18-6. Local Bus Arbitration Control Register – LBACR (Sheet 1 of 2)

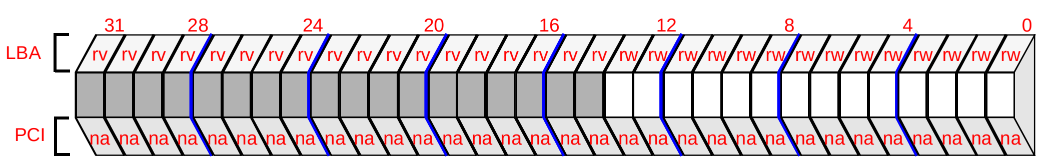

LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1600H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bits	Default	Description
31:14	0000 0H	Reserved.
13:12	00_2	External Bus Master Priority Note: Programming 11_2 (disabled) is not allowed for this bus master.
11:10	00_2	Primary ATU Priority
09:08	00_2	Secondary ATU Priority
07:06	00_2	DMA Channel 2 Priority

Table 18-6. Local Bus Arbitration Control Register – LBACR (Sheet 2 of 2)


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1600H
PCI: NA

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 Local Bus Address
PCI = PCI Configuration Address Offset

Bits	Default	Description
05:04	00_2	DMA Channel 1 Priority
03:02	00_2	DMA Channel 0 Priority
01:00	00_2	i960 Core Processor Priority

18.2.2 Removing Local Bus Ownership

With the exception of an external bus master currently owning the bus, the arbiter only removes GNT# to the current bus owner if the local bus arbitration latency counter has expired or the current bus owner removes its REQ#. When GNT# is removed, the bus master must get off the bus by removing its REQ#. See section 18.2.4, "External Bus Arbitration Support" (pg. 18-7) for more information on external bus masters.

When the current local bus owner relinquishes ownership, it removes its REQ# output for a minimum of one local bus clock. Once the arbiter detects the current owner's inactive REQ#, it grants the local bus to the next local bus winner by activating the appropriate GNT# signal. After the one clock deassertion, the previous local bus master is free to reassert its REQ# signal.

When a local bus master has completed its transaction, it removes the REQ# signal to the arbiter, regardless of the remaining count in the LBALCR. The arbiter is free to assign a new bus owner at this time. The LBALCR is reloaded with a new count value whenever new bus ownership is assigned, except when the bus is parked with the i960 core processor.

Due to the buffering capability within the DMA Controller and ATUs, data transfers to the PCI bus may continue. This means that ownership of a PCI bus may continue after ownership of the local bus has been lost, since ownership of the local bus and the PCI bus are independent.

When a DMA channel is performing a PCI memory write and invalidate transaction, the DMA channel does not relinquish the local bus until it has transferred a full cache line into its internal buffer. This means the DMA channel only relinquishes the bus on host system cache line aligned boundaries, regardless of the state of the internal GNT# signal and the LBALCR.

18.2.3 i960® Core Processor Bus Usage

The i960 core processor releases control of the local bus, when the latency timer times out, under the following conditions:

- After completing a data access
- After completing an instruction fetch access
- After completing an atomic access (read-modify-write)

Since software has no control over when the processor needs the bus, it should make use of the on-chip instruction cache, data cache, and internal data RAM to help reduce the number of processor bus requests.

18.2.4 External Bus Arbitration Support

External bus masters may be used on the local bus by adding external logic to control the HOLD/HOLDA mechanism. The i960 Rx I/O processor allows for one external bus master to participate in the fairness algorithm. Multiple bus masters require external logic to treat all external devices as a single bus master.

The 80960 arbitration logic supports external bus masters to control local bus. The arbiter maintains the standard HOLD/HOLDA protocol used on previous 80960 processors except that the i960 Rx I/O processor does not respond to the HOLD signal (i.e., assert HOLDA) while the core processor is in reset. Refer to section 13.6.1, "HOLD/HOLDA Protocol" (pg. 13-25) for a complete description of the HOLD/HOLDA interface for external bus masters.

18.2.5 Local Bus Arbitration Latency Counter

The Local Bus Arbitration Latency Counter Register (LBALCR) value sets the minimum period that the active bus master has control of the local bus. This register's value is loaded into the 12-bit counter each time the arbiter grants the local bus. The counter decrements on each processor clock until it reaches zero. When the counter reaches zero, two possible scenarios may occur:

- When a high-priority request is pending, the arbiter notifies the existing bus master and waits for the pending request to be removed (signifying the completion of the current data transfer). The programmed count value is reloaded into the LBALCR and the pending request is granted control of the local bus.

- When no pending requestsare pending and the current bus master still needs the bus, the arbiter continues to grant the current bus master control of the local bus. Every clock thereafter, the arbiter continues checking for pending bus requests. Upon recognizing a bus request the arbiter notifies the bus master and waits for the current bus request to de-assert. The arbiter then grants the pending bus master control and reloads the LBALCR. When the current bus master completes its transaction and there are no outstanding bus requests, the arbiter parks the local bus on the i960 core processor. The LBALCR is not reloaded when the bus is parked. It is not reloaded until a bus master, including the i960 core processor, is granted the bus.

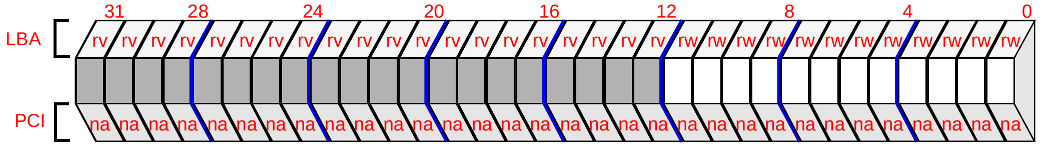
Table 18-7 shows the bit definitions for the local bus arbitration latency counter register.

18.2.6 Local Bus Arbitration Latency Counter Register – LBALCR

The Local Bus Arbitration Latency Counter Register (LBALCR) value sets the minimum period that the active bus master has control of the local bus.

LBALCR is a read/write register accessible through a memory-mapped interface from the local bus. The maximum value programmable is 0000 0FFFH. The minimum value programmable (0000 0000H) could result in the local bus being reassigned on every clock. When reading the LBALCR, the value returned is the programmed value, not the current count value.

Table 18-7. Local Bus Arbitration Latency Count Register – LBALCR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1604H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Reserved.
11:00	FFFH	Local Bus Arbitration Counter

18.2.7 Local Bus Backoff

The i960 Rx I/O processor backoff unit prevents deadlocks that occur when an i960 core processor outbound transaction through an ATU occurs simultaneously with an inbound ATU transaction and both transactions require the same resources. When backoff is required, the backoff unit three-states the address/data bus and the necessary bus control signals (as in HOLD/HOLDA assertion) to electrically remove the processor from the bus. The backoff unit simulates a cycle completion by asserting BLAST# on the cycle following the address phase.

The i960 core processor always backs off under the following situations:

- Outbound configuration read to the ATU
- Outbound memory read to the ATU
- Outbound I/O Read to the ATU
- Any transaction (read or write) to a busy ATU

A busy ATU is one that is currently processing an inbound transaction (inbound address queue is valid). When backoff occurs, the 12-bit arbitration counter is reset and is reloaded with a new count when the local bus is granted to another bus master. When the i960 Rx I/O processor is removed from backoff, the 12-bit arbitration counter is reset to a full count regardless of its value when backoff occurred. In addition, the processor moves to the highest priority in the local bus arbitration sequence for the purposes of reacquiring the bus after backoff. After bus re-acquisition, the processor returns to its preprogrammed priority.

18.3 SECONDARY PCI ARBITRATION UNIT

The secondary bus arbiter supports up to six secondary bus masters plus the secondary bus interface itself. Each request can be programmed to one of three priority levels or be disabled. A memory mapped control register, programmed by the application software, sets the priorities for each of the bus masters. Each priority level is handled in a round-robin fashion.

The initial priority for each bus master is programmable by software. While running, the arbiter promotes and demotes the bus masters using the round-robin scheme shown in Figure 18-2. After a device relinquishes control of the bus, it returns to its initial programmed priority. Table 18-2 shows the 2-bit values that correspond to each priority level.

The round-robin arbitration scheme supports three levels of round-robin arbitration. The three levels define a low, medium and high priority. Using the round-robin mechanism ensures there is a winner for each priority level. To enforce the concept of fairness, a slot is reserved for the winner of each priority level (except the highest) in the next highest priority. When the winner of a priority level is not granted the bus during that particular arbitration sequence, it is promoted to the next highest level of priority. Once its bus ownership is removed, the device is reset to its initially programmed priority and may start arbitration once again. Figure 18-2 and Table 18-8 show the three priority levels and the reserved slots for the promoted requestor. Refer to Table 18-4., Bus Arbitration Example – Three Bus Masters (pg. 18-4) and Table 18-5., Bus Arbitration Example – Six Bus Masters (pg. 18-5) for examples of the arbitration algorithm.

Each master on the secondary PCI bus is required to implement a latency timer. The timer determines the maximum period that the master is allowed to retain ownership of the bus.

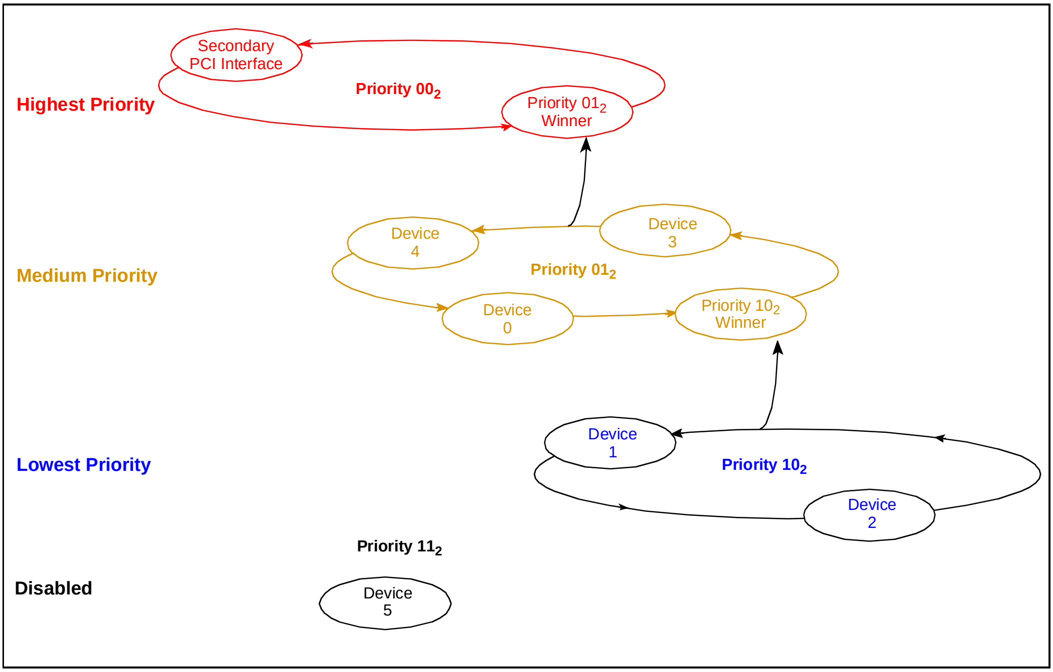


Figure 18-2. Secondary PCI Bus Arbitration Example

Table 18-8. Priority Programming for Secondary PCI Bus Arbitration Example

Bus Master	Programmed Priority
Secondary Interface	High - 00_2
Device 0	Medium - 01_2
Device 3	Medium - 01_2
Device 1	Medium - 01_2
Device 4	Low - 10_2
Device 2	Low - 10_2
Device 5	Disabled - 11_2

18.3.1 Arbitration Signaling Protocol

An agent requests the bus by asserting its REQ# output. Agents must only use REQ# to signal a true need for the bus, not to reserve the bus. When the secondary arbiter determines an agent may use the bus, it asserts the agent’s GNT# input.

The secondary arbiter may deassert an agent’s GNT# on any PCI clock. An agent must ensure its GNT# is asserted on the clock edge where it wants to start a transaction. When GNT# is deasserted, the transaction must not proceed. Once GNT# is asserted, it may be deasserted according to the following rules:

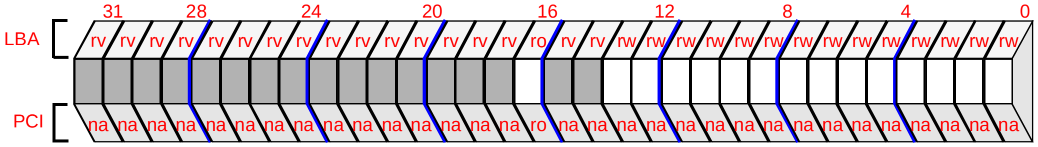
- When GNT# is deasserted and FRAME# is asserted, the bus transaction is valid and continues.
- One GNT# can be deasserted coincident with another GNT# being asserted if the bus is not in the IDLE state. Otherwise, a one clock delay is added between the deassertion of a GNT# and the assertion of the next GNT#. This prevents contention on the AD bus.
- While FRAME# is deasserted, GNT# may be deasserted any time in order to service another master, or in response to the associated REQ# being deasserted.

18.3.2 Secondary Arbitration Control Register - SACR

The Secondary Arbitration Control Register (SACR) sets the arbitration priority of each device that uses the secondary PCI bus. Each device is given a 2-bit priority as shown in Table 18-2. The SACR register is located in the PCI-to-PCI bridge configuration space.

The SACR register also contains the Secondary Arbiter Status bit for the secondary bus arbitration unit. This bit is set at reset by sampling the S_REQ5#/S_ARB_EN signal. When this bit is clear, the secondary bus arbiter is disabled and the bridge drives S_REQ# on S_GNT0# and samples S_GNT# on S_REQ0#. When S_REQ5#/S_ARB_EN is high on the rising edge of P_RST#, the internal secondary arbitration unit is enabled. When S_REQ5#/S_ARB_EN is low on the rising edge of P_RST#, the internal secondary arbitration unit is disabled.

Table 18-9. Secondary Arbitration Control Register - SACR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro rv rv rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na ro na na na na na na na na na na na na na na na na

LBA: 104CH
PCI: 4CH

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bits	Default	Description
31:17	0000H	Reserved.
16[1]	Based on S_REQ5#/S_ARB_EN signal at reset	Secondary Arbiter Enable 0 = Disabled 1 = Enabled
15:14	00_2	Reserved.
13:12	00_2	Device 5 Priority
11:10	00_2	Device 4 Priority
09:08	00_2	Device 3 Priority
07:06	00_2	Device 2 Priority
05:04	00_2	Device 1 Priority
03:02	00_2	Device 0 Priority
01:00	00_2	Secondary PCI Interface Priority (Bridge, DMA Channel 2, or Secondary ATU)

NOTES:

1. This bit is Read Only from BOTH the PCI interface and the 80960 local bus. This is unlike other Read Only bits which are read only from the PCI interface, and Read/Write from the 80960 local bus.

18.3.3 Secondary Bus Arbitration Parking

Arbitration parking occurs when the internal arbitration unit asserts GNT# to a selected PCI bus agent, when no agent is currently using or requesting the bus. The i960 Rx I/O processor’s internal arbitration unit parks the secondary bus with the secondary PCI interface when no other agent is requesting the bus. This maximizes the performance of primary to secondary bus transactions by allowing downstream transactions without the secondary bus asserting REQ# internally and waiting for the return of GNT#.

In the situation where the secondary bus is parked at the i960 Rx I/O processor secondary PCI interface, the bridge asserts S_AD31:00, S_C/BE3:0# and one clock later S_PAR to prevent the PCI bus from floating. The secondary interface asserts these signals within two to three (for PAR) PCI clocks. Refer to the *PCI Local Bus Specification Revision 2.1*.

When the bus is parked at the secondary interface and in an IDLE state, it loses the bus when the arbiter asserts another agent's GNT#. The parked agent (the secondary interface) relinquishes the bus and deassert its address and command signals in 1 PCI clock and parity 1 PCI clock after that (1-2 clocks total). When the arbiter removes the secondary bus internal GNT# at the same time that the secondary interface drives FRAME# on the bus, the secondary interface completes the initiated bus transaction.

18.4 INTERNAL ARBITRATION UNITS

The i960 Rx I/O processor contains two internal arbitration units that control access to the internal PCI buses within the device. The Primary Internal PCI Arbitration Unit arbitrates for the following internal units:

- Primary Bridge Interface
- Primary ATU
- DMA Channel 0, DMA Channel 1

The Secondary Internal PCI Arbitration Unit arbitrates for the following internal units:

- Secondary Bridge Interface
- Secondary ATU
- DMA Channel 2

Each internal PCI arbitration unit uses a fixed round-robin arbitration scheme with each device on a bus having equal priority.

18.4.1 Internal Master Latency Timer

Each PCI interface of the i960 Rx I/O processor (primary and secondary) contains a Master Latency Timer (MLT) for use by the internal resources when they are acting as PCI bus masters. Both ATUs, the DMA channels, and the bridge interfaces use an MLT. MLT usage is explained in the *PCI Local Bus Specification Revision 2.1*. As defined by the PCI specification, a PCI bus master must release bus ownership when it has lost grant and its MLT has expired. The internal PCI arbitration unit extends this concept by adding all of the internal bus master resources to the arbitration equation and is therefore capable of removing the current bus master when its MLT has expired.

Each internal bus master may lose its grant based on whether an external bus master wants the bus (external grant inactive) or whether an internal bus master wants the bus (internal grant inactive while external grant still active). Each bus master must relinquish the bus when an external device or one of the internal resources requests the bus.

19

TIMERS

CHAPTER 19
TIMERS

This chapter describes the i960® Rx I/O processor dual, independent 32-bit timers. Topics include timer registers (TMRx, TCRx and TRRx), timer operation, timer interrupts, and timer register values at initialization.

Each timer is programmed by the timer registers. These registers are memory-mapped within the processor, addressable on 32-bit boundaries. When enabled, a timer decrements the user-defined count value with each Timer Clock (TCLOCK) cycle. The countdown rate is also user-configurable to be equal to the bus clock frequency, or the bus clock rate divided by 2, 4 or 8. The timers can be programmed to either stop when the count value reaches zero (single-shot mode) or run continuously (auto-reload mode). When a timer's count reaches zero, the timer's interrupt unit signals the processor's interrupt controller. Figure 19-1 shows a diagram of the timer functions. See also Figure 19-2 for the Timer Unit state diagram.

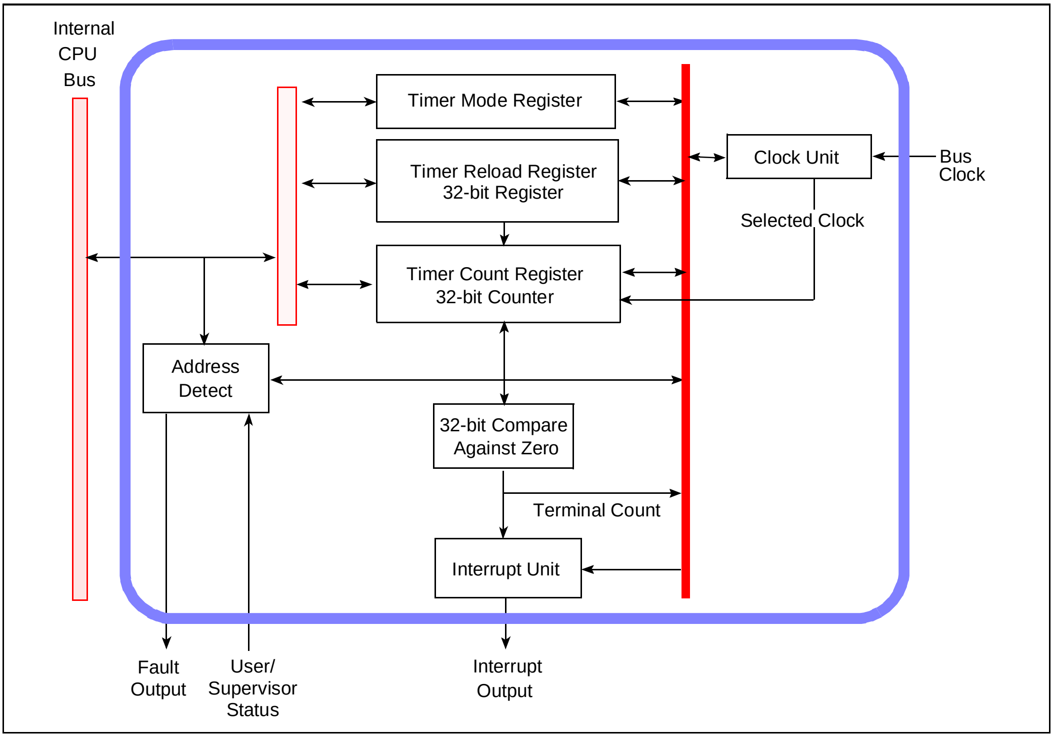


Figure 19-1. Timer Functional Diagram

Table 19-1. Timer Performance Ranges

Bus Frequency (MHz)	Max Resolution (ns)	Max Range (mins)
40	25	14.3
33	30.3	17.4
25	40	22.9
20	50	28.6
16	62.5	35.8

19.1 TIMER REGISTERS

As shown in Table 19-2, each timer has three memory-mapped registers:

- Timer Mode Register - programs the specific mode of operation or indicates the current programmed status of the timer. This register is described in section 19.1.1, Timer Mode Registers – TMR0:1 (pg. 19-2).
- Timer Count Register - contains the timer's current count. See section 19.1.2, Timer Count Register – TCR0:1 (pg. 19-5).
- Timer Reload Register - contains the timer's reload count. See section 19.1.3, Timer Reload Register – TRR0:1 (pg. 19-6).

Table 19-2. Timer Registers

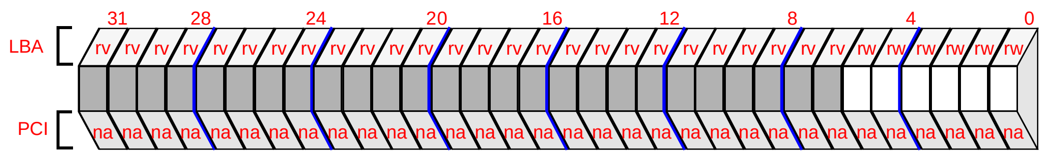
Timer Unit	Register Acronym	Register Name
Timer 0	TMR0	Timer Mode Register 0
	TCR0	Timer Count Register 0
	TRR0	Timer Reload Register 0
Timer 1	TMR1	Timer Mode Register 1
	TCR1	Timer Count Register 1
	TRR1	Timer Reload Register 1

For register memory locations, see Table C-3., Timer Registers (pg. C-4).

19.1.1 Timer Mode Registers – TMR0:1

The Timer Mode Register (TMRx) lets the user program the mode of operation and determine the current status of the timer. TMRx bits are described in the subsections following Table 19-3 and are summarized in Table 19-7.

Table 19-3. Timer Mode Register – TMRx


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH 0-0308H
	CH 1-0318H
		Legend:
		RV = Reserved
		RS = Read/Set
			NA = Not Accessible
			PR = Preserved
			RC = Read Clear
				RO = Read Only
				RW = Read/Write

PCI:	NA	LBA = 80960 Local Bus Address	PCI = PCI Configuration Address Offset	

Bit	Default	Description
31:06	0000 000H	Reserved. Initialize to 0.
05:04	00_2	Timer Input Clock Selects - TMRx.csel1:0 (00) 1:1 Timer Clock = Bus Clock (01) 2:1 Timer Clock = Bus Clock / 2 (10) 4:1 Timer Clock = Bus Clock / 4 (11) 8:1 Timer Clock = Bus Clock / 8
03	0_2	Timer Register Supervisor Write Control - TMRx.sup (0) Supervisor and User Mode Write Enabled (1) Supervisor Mode Only Write Enabled
02	0_2	Timer Auto Reload Enable - TMRx.reload (0) Auto Reload Disabled (1) Auto Reload Enabled
01	0_2	Timer Enable - TMRx.enable (0) Disabled (1) Enabled
00	0_2	Terminal Count Status - TMRx.tc (0) No Terminal Count (1) Terminal Count

19.1.1.1 Bit 0 - Terminal Count Status Bit (TMRx.tc)

The TMRx.tc bit is set when the Timer Count Register (TCRx) decrements to 0 and bit 2 (TMRx.reload) is not set for a timer. The TMRx.tc bit allows applications to monitor timer status through software instead of interrupts. TMRx.tc remains set until software accesses (reads or writes) the TMRx. The access clears TMRx.tc. The timer ignores any value specified for TMRx.tc in a write request.

When auto-reload is selected for a timer and the timer is enabled, the TMRx.tc bit status is unpredictable. Software should not rely on the value of the TMRx.tc bit when auto-reload is enabled.

The processor also clears the TMRx.tc bit upon hardware or software reset. Refer to section 11.2, 80960Rx INITIALIZATION (pg. 11-2).

19.1.1.2 Bit 1 - Timer Enable (TMRx.enable)

The TMRx.enable bit allows user software to control the timer's RUN/STOP status. When:

TMRx.enable = 1 The Timer Count Register (TCRx) value decrements every Timer Clock (TCLOCK) cycle. TCLOCK is determined by the Timer Input Clock Select (TMRx.csel bits 0-1). See section 19.1.1.5. When TMRx.reload=0, the timer automatically clears TMRx.enable when the count reaches zero. When TMRx.reload=1, the bit remains set. See section 19.1.1.3.

TMRx.enable = 0 The timer is disabled and ignores all input transitions.

User software sets this bit. Once started, the timer continues to run, regardless of other processor activity. Three events can stop the timer:

- User software explicitly clearing this bit (i.e., TMRx.enable = 0).
- TCRx value decrements to 0, and the Timer Auto Reload Enable (TMRx.reload) bit = 0.
- Hardware or software reset. Refer to section 11.2, 80960Rx INITIALIZATION (pg. 11-2).

19.1.1.3 Bit 2 - Timer Auto Reload Enable (TMRx.reload)

The TMRx.reload bit determines whether the timer runs continuously or in single-shot mode. When TCRx = 0 and TMRx.enable = 1 and:

TMRx.reload = 1 The timer runs continuously. The processor:

1. Automatically loads TCRx with the value in the Timer Reload Register (TRRx), when TCRx value decrements to 0.

2. Decrements TCRx until it equals 0 again.

Steps 1 and 2 repeat until software clears TMRx bits 1 or 2.

TMRx.reload = 0 The timer runs until the Timer Count Register = 0. TRRx has no effect on the timer.

User software sets this bit. When TMRx.enable and TMRx.reload are set and TRRx does not equal 0, the timer continues to run in auto-reload mode, regardless of other processor activity. Two events can stop the timer:

- User software explicitly clearing either TMRx.enable or TMRx.reload.
- Hardware or software reset.

The processor clears this bit upon hardware or software reset.

19.1.1.4 Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup)

The TMRx.sup bit enables or disables user mode writes to the timer registers (TMRx, TCRx, TRRx). Supervisor mode writes are allowed regardless of this bit's condition. Software can read these registers from either mode.

When:

TMRx.sup = 1 The timer generates a TYPE.MISMATCH fault when a user mode task attempts a write to any of the timer registers; however, supervisor mode writes are allowed.

TMRx.sup = 0 The timer registers can be written from either user or supervisor mode.

The processor clears TMRx.sup upon hardware or software reset. Refer to section 11.2, 80960Rx INITIALIZATION (pg. 11-2).

19.1.1.5 Bits 4, 5 - Timer Input Clock Select (TMRx.csel1:0)

User software programs the TMRx.csel bits to select the Timer Clock (TCLOCK) frequency. See Table 19-4. As shown in Figure 19-1, the bus clock is an input to the timer clock unit. These bits allow the application to specify whether TCLOCK runs at or slower than the bus clock frequency.

Table 19-4. Timer Input Clock (TCLOCK) Frequency Selection

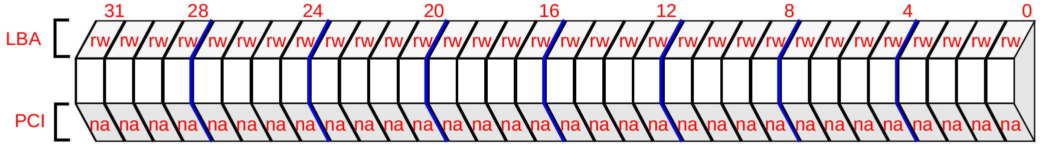
Bit 5 TMRx.csel1	Bit 4 TMRx.csel0	Timer Clock (TCLOCK)
0	0	Timer Clock = Bus Clock
0	1	Timer Clock = Bus Clock / 2
1	0	Timer Clock = Bus Clock / 4
1	1	Timer Clock = Bus Clock / 8

The processor clears these bits upon hardware or software reset (TCLOCK = Bus Clock).

19.1.2 Timer Count Register – TCR0:1

The Timer Count Register (TCRx) is a 32-bit register that contains the timer's current count. The register value decrements with each timer clock tick. When this register value decrements to zero (terminal count), a timer interrupt is generated. When TMRx.reload is not set for the timer, the status bit in the timer mode register (TMRx.tc) is set and remains set until the TMRx register is accessed. Table 19-5 shows the timer count register.

Table 19-5. Timer Count Register – TCRx



LBA:	CH 0-0304H
	CH 1-0314H
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

PCI:	na	

Bit	Default	Description
31:00	0000 0000H	Timer Count Value - TCRx.d31:0

The valid programmable range is from 1H to FFFF FFFFH. Avoid programming TCRx to 0 as it will have varying results as described in section 19.5, UNCOMMON TCRx AND TRRx CONDITIONS (pg. 19-10).

User software can read or write TCRx whether the timer is running or stopped. Bit 3 of TMRx determines user read/write control (see section 19.1.1.4). The TCRx value is undefined after hardware or software reset.

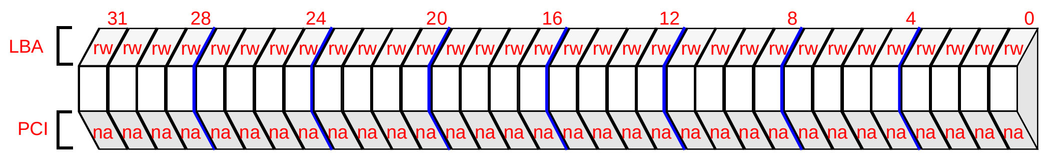
19.1.3 Timer Reload Register – TRR0:1

The Timer Reload Register (TRRx; Table 19-6) is a 32-bit register that contains the timer's reload count. The timer loads the reload count value into TCRx when TMRx.reload is set (1), TMRx.enable is set (1) and TCRx equals zero.

As with TCRx, the valid programmable range is from 1H to FFFF FFFFH. Avoid programming a value of 0, as it may prevent TINTx from asserting continuously. (See section 19.5, UNCOMMON TCRx AND TRRx CONDITIONS (pg. 19-10) for more information.)

User software can access TRRx whether the timer is running or stopped. Bit 3 of TMRx determines read/write control (see section 19.1.1.4, Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup) (pg. 19-5)). TRRx value is undefined after hardware or software reset.

Table 19-6. Timer Reload Register – TRRx


31
28
24
20
16
12
8
4
0
LBA
PCI
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: CH 0-0300H
CH 1-0310H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:00	0000 0000H	Timer Auto-Reload Value - TRRx.d31:0

19.2 TIMER OPERATION

This section summarizes timer operation and describes load/store access latency for the timer registers.

19.2.1 Basic Timer Operation

Each timer has a programmable enable bit in its control register (TMRx.enable) to start and stop counting. The supervisor (TMRx.sup) bit controls write access to the enable bit. This allows the programmer to prevent user mode tasks from enabling or disabling the timer. Once the timer is enabled, the value stored in the Timer Count Register (TCRx) decrements every Timer Clock (TCLOCK) cycle. TCLOCK is determined by the Timer Input Clock Select (TMRx.csel) bit setting. The countdown rate can be set to equal the bus clock frequency, or the bus clock rate divided by 2, 4 or 8. Setting TCLOCK to a slower rate lets the user specify a longer count period with the same 32-bit TCRx value.

Software can read or write the TCRx value whether the timer is running or stopped. This lets the user monitor the count without using hardware interrupts. The TMRx.sup bit lets the programmer allow or prevent user mode writes to TCRx, TMRx and TRRx.

When the TCRx value decrements to zero, the unit’s interrupt request signals the processor’s interrupt controller. See section 19.3, TIMER INTERRUPTS (pg. 19-10) for more information. The timer checks the value of the timer reload bit (TMRx.reload) setting. When TMRx.reload. = 1, the processor:

- Automatically reloads TCRx with the value in the Timer Reload Register (TRRx).
- Decrements TCRx until it equals 0 again.

This process repeats until software clears TMRx.reload or TMR.enable.

When TMRx.reload = 0, the timer stops running and sets the terminal count bit (TMRx.tc). This bit remains set until user software reads or writes the TMRx register. Either access type clears the bit. The timer ignores any value specified for TMRx.tc in a write request.

Table 19-7. Timer Mode Register Control Bit Summary

Bit 3 (TMRx.sup)	TRRx	TCRx	Bit 2 (TMRx.reload)	Bit 1 (TMRx.enable)	Action
X	X	X	X	0	Timer disabled.
X	X	N	0	1	Timer enabled, TMRx.enable is cleared when TCRx decrements to zero.
X	N	N	1	1	Timer and auto reload enabled,TMRx.enable remains set when TCRx=0. When TCRx=0, TCRx equals the TRRx value.
0	X	X	X	X	No faults for user mode writes are generated.
1	X	X	X	X	TYPE.MISMATCH fault generated on user mode write.

Notes: X = don’t care

N = a number between 1H and FFFF FFFFH

19.2.2 Load/Store Access Latency for Timer Registers

As with all other load accesses from internal memory-mapped registers, a load instruction that accesses a timer register has a latency of one internal processor cycle. With one exception, a store access to a timer register completes and all state changes take effect before the next instruction begins execution. The exception to this is when disabling a timer. Latency associated with the disabling action is such that a timer interrupt may be posted immediately after the disabling instruction completes. This can occur when the timer is near zero as the store to TMRx occurs. In this case, the timer interrupt is posted immediately after the store to TMRx completes and before the next instruction can execute. Table 19-8 summarizes the timer access and response timings. Refer also to the individual register descriptions for details.

Note that the processor may delay the actual issuing of the load or store operation due to previous instruction activity and resource availability of processor functional units.

The processor ensures that the TMRx.tc bit is cleared within one bus clock after a load or store instruction accesses TMRx.

Table 19-8. Timer Responses to Register Bit Settings

Name	Status	Action
(TMRx.tc) Terminal Count Bit 0	READ	Timer clears this bit when user software accesses TMRx. This bit can be set 1 bus clock later. The timer sets this bit within 1 bus clock of TCRx reaching zero when TMRx.reload=0.
	WRITE	Timer clears this bit within 1 bus clock after the software accesses TMRx. The timer ignores any value specified for TMRx.tc in a write request.
(TMRx.enable) Timer Enable Bit 1	READ	Bit is available 1 bus clock after executing a read instruction from TMRx.
	WRITE	Writing a ‘1’ enables the bus clock to decrement TCRx within 1 bus clock after executing a store instruction to TMRx.
(TMRx.reload) Timer Auto Reload Enable Bit 2	READ	Bit is available 1 bus clock after executing a read instruction from TMRx.
	WRITE	Writing a ‘1’ enables the reload capability within 1 bus clock after the store instruction to TMRx has executed. The timer loads TRRx data into TCRx and decrements this value during the next bus clock cycle.
(TMRx.sup) Timer Register Supervisor Write Control Bit 3	READ	Bit is available 1 bus clock after executing a read instruction from TMRx.
	WRITE	Writing a ‘1’ locks out user mode writes within 1 bus clock after the store instruction executes to TMRx. Upon detecting a user mode write the timer generates a TYPE.MISMATCH fault.
(TMRx.csel1:0) Timer Input Clock Select Bits 4-5	READ	Bits are available 1 bus clock after executing a read instruction from TMRx.csel1:0 bit(s).
	WRITE	The timer re-synchronizes the clock cycle used to decrement TCRx within one bus clock cycle after executing a store instruction to TMRx.csel1:0 bit(s).
(TCRx.d31:0) Timer Count Register	READ	The current TCRx count value is available within 1 bus clock cycle after executing a read instruction from TCRx. When the timer is running, the pre-decremented value is returned as the current value.
	WRITE	The value written to TCRx becomes the active value within 1 bus clock cycle. When the timer is running, the value written is decremented in the current clock cycle.
(TRRx.d31:0) Timer Reload Register	READ	The current TRRx count value is available within 1 bus clock after executing a read instruction from TRRx. When the timer is transferring the TRRx count into TCRx in the current count cycle, the timer returns the new TCRx count value to the executing read instruction.
	WRITE	The value written to TRRx becomes the active value stored in TRRx within 1 bus clock cycle. When the timer is transferring the TRRx value into the TCRx, data written to TRRx is also transferred into TCRx.

19.3 TIMER INTERRUPTS

Each timer is the source for one interrupt. When a timer detects a zero count in its TCRx, the timer generates an internal edge-detected Timer Interrupt signal (TINTx) to the interrupt controller, and the interrupt-pending (IPND.tipx) bit is set in the interrupt controller. Each timer interrupt can be selectively masked in the Interrupt Mask (IMSK) register or handled as a dedicated hardware-requested interrupt. Refer to CHAPTER 8, INTERRUPTS for a description of hardware-requested interrupts.

When the interrupt is disabled after a request is generated, but before a pending interrupt is serviced, the interrupt request is still active (the Interrupt Controller latches the request). When a timer generates a second interrupt request before the CPU services the first interrupt request, the second request may be lost.

When auto-reload is enabled for a timer, the timer continues to decrement the value in TCRx even after entry into the timer interrupt handler.

19.4 POWERUP/RESET INITIALIZATION

Upon power up, external hardware reset or software reset (**sysctl**), the timer registers are initialized to the values shown in Table 19-9.

Table 19-9. Timer Powerup Mode Settings

Mode/Control Bit	Notes
TMRx.tc = 0	No terminal count
TMRx.enable = 0	Prevents counting and assertion of TINTx
TMRx.reload = 0	Single terminal count mode
TMRx.sup = 0	Supervisor or user mode access
TMRx.csel1:0 = 0	Timer Clock = Bus Clock
TCRx.d31:0 = 0	Undefined
TRRx.d31:0 = 0	Undefined
TINTx output	Deasserted

19.5 UNCOMMON TCRx AND TRRx CONDITIONS

Table 19-7 summarizes the most common settings for programming the timer registers. Under certain conditions, however, it may be useful to set the Timer Count Register or the Timer Reload Register to zero before enabling the timer. Table 19-10 details the conditions and results when these conditions are set.

Table 19-10. Uncommon TMRx Control Bit Settings

TRRx	TCRx	Bit 2 (TMRx.reload)	Bit 1 (TMRx.enable)	Action
X	0	0	1	TMRx.tc and TINTx set, TMR.enable cleared
0	0	1	1	Timer and auto reload enabled, TINTx not generated and timer enable remains set.
0	N	1	1	Timer and auto reload enabled. TINT.x set when TCRx=0. The timer remains enabled but further TINTx's are not generated.
N	0	1	1	Timer and auto reload enabled, TINTx not set initially, TCRx = TRRx, TINTx set when TCRx has completely decremented the value it loaded from TRRx. TMRx.enable remains set.

NOTE:
X = don't care

N = a number between 1H and FFFF FFFFH

19.6 TIMER STATE DIAGRAM

Figure 19-2 shows the common states of the Timer Unit. For uncommon conditions see section 19.5, UNCOMMON TCRx AND TRRx CONDITIONS.

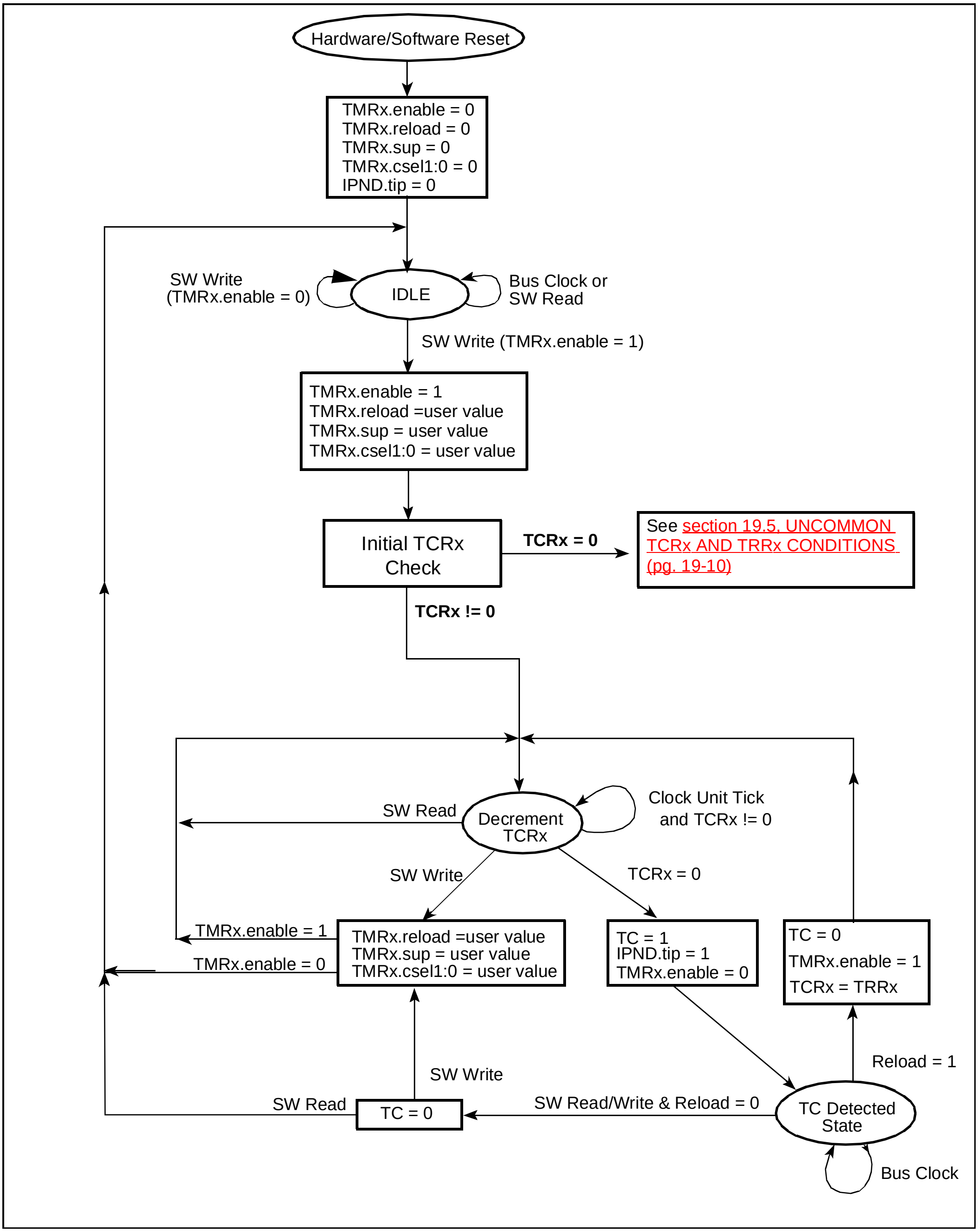


Figure 19-2. Timer Unit State Diagram

20

DMA CONTROLLER

CHAPTER 20
DMA CONTROLLER

This chapter describes the integrated Direct Memory Access (DMA) Controller, including the operation modes, setup, external interface, registers and interrupts.

20.1 OVERVIEW

The DMA Controller provides low-latency, high-throughput data transfer capability. The DMA Controller optimizes block transfers of data between the PCI bus and 80960 local bus memory. The DMA is an initiator on the PCI bus with PCI burst capabilities to provide a maximum throughput of 132 Mbytes/sec at 33 MHz.

Each channel contains a 64-byte data queue. This queue temporarily holds data to increase data transfer performance in both directions.

Figure 20-1 shows the DMA channel to PCI bus connections.

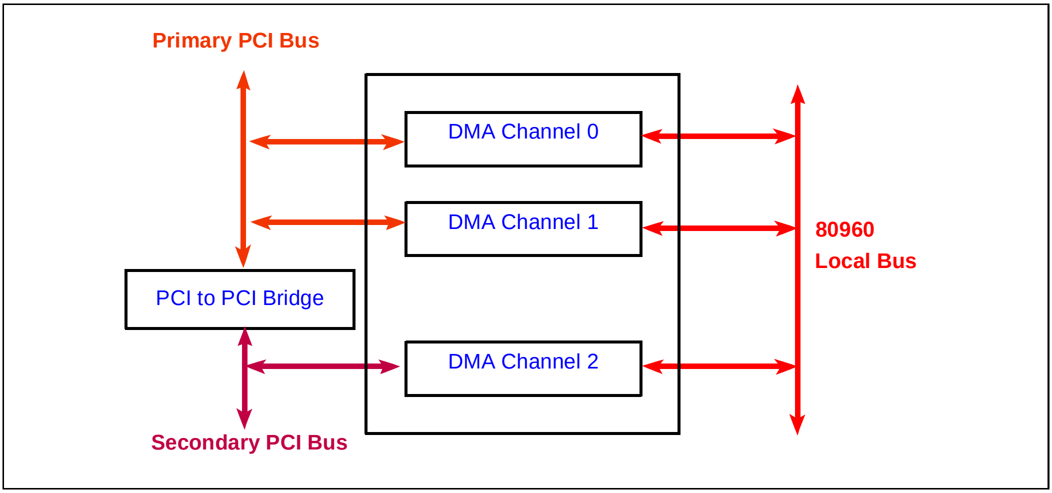


Figure 20-1. DMA Controller Block Diagram

The DMA Controller hardware executes data transfers and provides the programming interface. Features include:

- Three Independent Channels
- Memory Controller Interface
- 32-bit addressing range on the 80960 local bus
- 64-bit addressing range on the primary and secondary PCI interfaces by using PCI Dual Address Cycle (DAC)
- Independent PCI interfaces to the primary and secondary PCI buses
- Hardware support for unaligned data transfers for both the PCI bus and 80960 local bus
- Full 132 Mbyte/sec burst support for both the PCI bus and 80960 local bus
- Direct addressing to and from the PCI bus
- Fully programmable from the i960 core processor
- Support for automatic data chaining for gathering and scattering of data blocks
- Demand Mode Support for 32 bit external devices on DMA channel 0

20.2 THEORY OF OPERATION

The DMA Controller provides three channels of high throughput PCI-to-memory transfers:

- Channels 0 and 1 transfer data blocks between the primary PCI bus and 80960 local memory. Channel 0 also supports demand-mode transfers.
- Channel 2 transfers blocks of data between the secondary PCI bus and the 80960 local memory.

Channel 0’s additional support for demand mode operation enables an external device to assert a DMA request signal and provide the data for a DMA transfer. During demand mode operation, the DMA controller supports the full 132 Mbytes/sec data throughput. The DMA Controller only supports 32-bit wide external 80960 local bus widths.

Each channel has a PCI bus interface and an 80960 local bus interface. Figure 20-2 shows the block diagram for one DMA Controller channel. Each channel also has an independent bus request/grant signal pair to the 80960 local bus arbitration to decide which local bus master has access to the 80960 local bus.

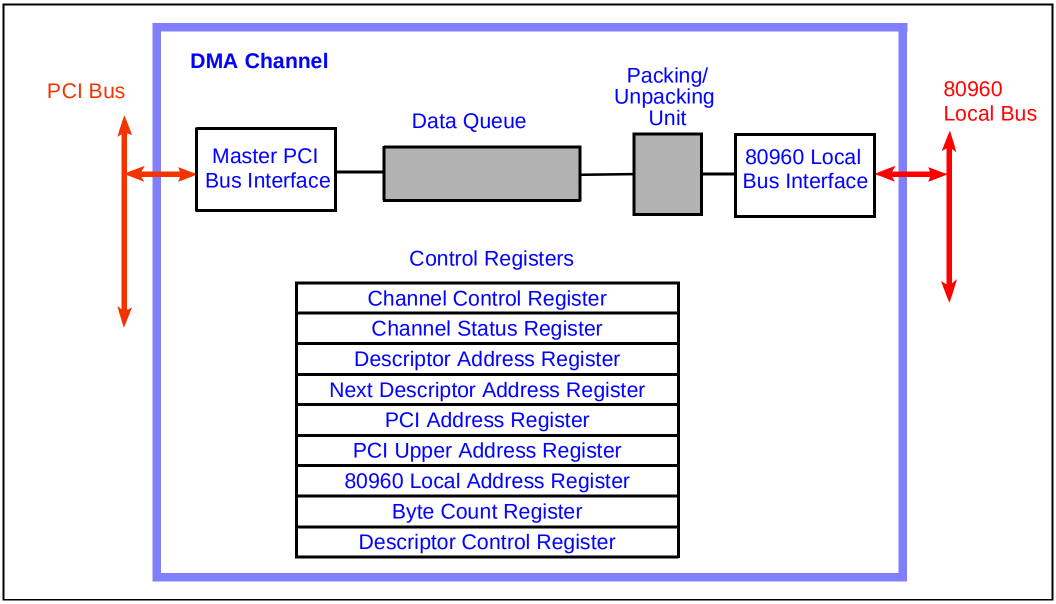


Figure 20-2. DMA Channel Block Diagram

Each DMA channel uses direct addressing for both the PCI bus and the 80960 local bus. It supports data transfers to and from the full 64-bit PCI bus address range. This includes 64-bit addressing using PCI DAC command. Each DMA channel provides a special register which contains the upper 32 address bits for the 64-bit address. The DMA channels do not support data transfers that cross a 32-bit address boundary. The PCI interface and the 80960 local bus interface support 2 Kbyte burst lengths. The DMA Unit rearbitrates for the 80960 local bus at 2 Kbyte boundaries.

The DMA channel programming interface is accessible from the 80960 local bus through a memory-mapped register interface. Each DMA channel is programmed independently and has its own set of registers. A DMA transfer is configured by writing the source address, destination address, number of bytes to transfer, and various control information into a chain descriptor in 80960 local memory. Chain descriptors are described in detail in section 20.3, DMA TRANSFER.

Each DMA channel supports chaining. Chain descriptors describe one DMA transfer and can be linked together in 80960 local memory to form a linked list. Each chain descriptor contains all the necessary information for transferring a block of data in addition to a pointer to the next chain descriptor. End of chain is indicated when the pointer is zero.

Each DMA channel contains a hardware data packing and unpacking unit. This unit enables data transfers from or to unaligned addresses in either the PCI address space or the 80960 local address space. All combinations of unaligned data are supported with the packing and unpacking unit.

20.3 DMA TRANSFER

A DMA transfer is a block move of data from one memory address space to another. DMA transfers are configured and initiated through a set of memory-mapped registers, and one or more chain descriptors located in local memory. Table 20-1 identifies the registers; see also section 20.7, REGISTER DEFINITIONS. A DMA transfer is defined by the source address, destination address, number of bytes to transfer, and control values. These values are loaded into the chain descriptor before a DMA transfer begins.

Table 20-1. DMA Registers

Register	Abbreviation	Description
Channel Control Register	CCR	Channel Control Word
Channel Status Register	CSR	Channel Status Word
Descriptor Address Register	DAR	Address of Current Chain Descriptor
Next Descriptor Address Register	NDAR	Address of Next Chain Descriptor
PCI Address Register	PADR	Lower 32-bit PCI Address of Source/Destination
PCI Upper Address Register	PUADR	Upper 32-bit PCI Address of Source/Destination
80960 Local Address Register	LADR	80960 Local Bus Address of Source/Destination
Byte Count Register	BCR	Number of Bytes to transfer
Descriptor Control Register	DCR	Chain Descriptor Control Word

20.3.1 Chain Descriptors

All DMA transfers are controlled by chain descriptors located in local memory. A chain descriptor contains the necessary information to complete one data transfer. A single DMA transfer has only one chain descriptor in memory. Chain descriptors can be linked together to form more complex DMA operations.

To perform a DMA transfer, one or more chain descriptors must first be written to 80960 local memory. Figure 20-3 shows the format of an individual chain descriptor. Every descriptor requires six contiguous words in 80960 local bus memory and is required to be aligned on an 8-word boundary. All six words are required.

Chain Descriptor in 80960 Memory	Description
Next Descriptor Address (NDA)	Address of Next Chain Descriptor
PCI Address [31:0] (PAD)	Lower 32-bit PCI Source/Destination Address
PCI Upper Address [63:32] (PUAD)	Upper 32-bit PCI Source/Destination Address
80960 Local Address (LAD)	80960 Local Bus Address
Byte Count (BC)	Number of Bytes to Transfer
Descriptor Control (DC)	Descriptor Control

Figure 20-3. DMA Chain Descriptor

Each chain descriptor word is analogous to control register values. Bit definitions for chain descriptor words are the same as for the DMA control registers.

- The first word is the 80960 local bus memory address of the next chain descriptor. A zero value specifies the end of chain. This value is loaded into the Next Descriptor Address Register. Because chain descriptors must be aligned on an 8-word boundary, the channel may ignore bits 04:00 of this address.
- The second word is the lower 32-bit PCI source/destination address. This address is generated on the PCI bus. This value is loaded into the PCI Address Register.
- The third word is the upper 32-bit PCI source/destination address, if needed. This address is used during Dual Address Cycles for driving 64-bit PCI addresses. The address is ignored when DAC is disabled. This value is loaded into the PCI Upper Address Register.
- The fourth word is the 80960 local bus source/destination address. This address is driven on the 80960 local bus. This value is loaded into the 80960 Local Address Register.
- The fifth word is the Byte Count value. This value determines the number of bytes to transfer. This value is loaded into the Byte Count Register.
- The sixth word is the Descriptor Control word. This word configures the DMA channel for one DMA transfer. It contains the PCI command type, which determines data transfer direction. This value is loaded into the Descriptor Control Register.

There are no data alignment requirements for either the PCI address or the 80960 local bus address. However, maximum performance is obtained from aligned transfers, especially small transfers. See section 20.9, PACKING AND UNPACKING (pg. 20-37).

A series of chain descriptors can be built in local memory to transfer data between the PCI buses and 80960 local bus. For example, the application can build multiple chain descriptors to transfer many blocks of data which have different source addresses within local memory. When the multiple chain descriptors are built in 80960 local bus memory, the application can link each chain descriptor using the Next Descriptor Address in the chain descriptor. This address logically links the chain descriptors together. This allows the application to build a list of DMA transfers which may not require the i960 core processor until all DMA transfers are complete. Figure 20-4 shows a list of DMA transfers built in external memory and how they are linked together.

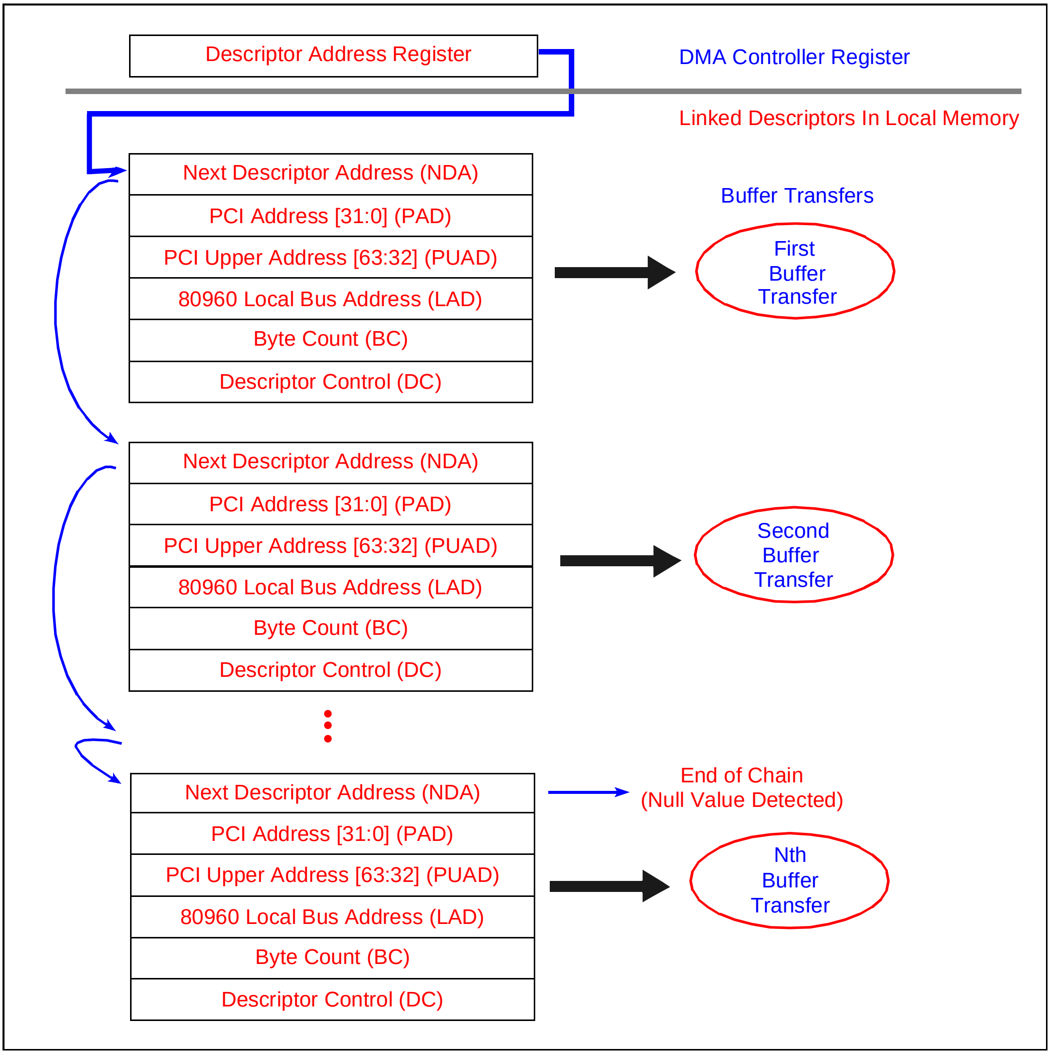

Descriptor Address Register
DMA Controller Register
Linked Descriptors In Local Memory
Next Descriptor Address (NDA)
PCI Address [31:0] (PAD)
PCI Upper Address [63:32] (PUAD)
80960 Local Bus Address (LAD)
Byte Count (BC)
Descriptor Control (DC)
Buffer Transfers
First Buffer Transfer
Next Descriptor Address (NDA)
PCI Address [31:0] (PAD)
PCI Upper Address [63:32] (PUAD)
80960 Local Bus Address (LAD)
Byte Count (BC)
Descriptor Control (DC)
Second Buffer Transfer
Next Descriptor Address (NDA)
PCI Address [31:0] (PAD)
PCI Upper Address [63:32] (PUAD)
80960 Local Bus Address (LAD)
Byte Count (BC)
Descriptor Control (DC)
End of Chain (Null Value Detected)
Nth Buffer Transfer

Figure 20-4. DMA Chaining Operation

20.3.2 Initiating DMA Transfers

A DMA transfer is started by first building one or more chain descriptors in 80960 local memory. Each chain descriptor takes the form shown in Figure 20-3. The chain descriptors are required to be aligned on an 8-word boundary in 80960 local memory. The following steps describe new DMA transfer initiation:

1. The channel must be inactive prior to starting a DMA transfer. This can be checked by software by reading the Channel Status Register's (CSR) Channel Active bit. When this bit is clear, the channel is inactive. When this bit is set, the channel is currently active with a DMA transfer.
2. Software writes the first chain descriptor's address to the Next Descriptor Address Register.
3. Software sets the Channel Control Register's (CCR) Channel Enable bit. Because this is the start of a new DMA transfer and not the resumption of a previous DMA transfer, the CCR Chain Resume bit should be clear.
4. The channel starts the DMA transfer by reading the chain descriptor at the address contained in the Next Descriptor Address Register. The channel loads the chain descriptor values into the CCRs and begins data transfer. The Descriptor Address Register now contains the address of the chain descriptor just read and the Next Descriptor Address Register now contains the Next Descriptor Address from the chain descriptor just read.

The last descriptor in the DMA chain list has zero in the next descriptor address field, which identifies it as the last chain descriptor. The NULL value notifies the DMA channel to stop reading chain descriptors from memory.

Once a DMA transfer is active, it may be temporarily suspended by clearing the CCR Channel Enable bit. Note that this does not abort the DMA transfer; the channel resumes the DMA transfer when the Channel Enable bit is set.

When descriptors are read from external memory, bus latency and memory speed affect chaining latency. Chaining latency is defined as the time required for the channel to access the next chain descriptor plus the time required to set up for the next DMA transfer.

20.3.3 Scatter Gather DMA Transfers

The DMA Controller can be used to perform typical scatter gather data transfers. This consists of programming the chain descriptors to gather the data which may be located in non-contiguous blocks of memory. The chain descriptor specifies the destination location such that once the data has been transferred, the data is contiguous in memory. Figure 20-5 shows how the destination pointers can gather data.

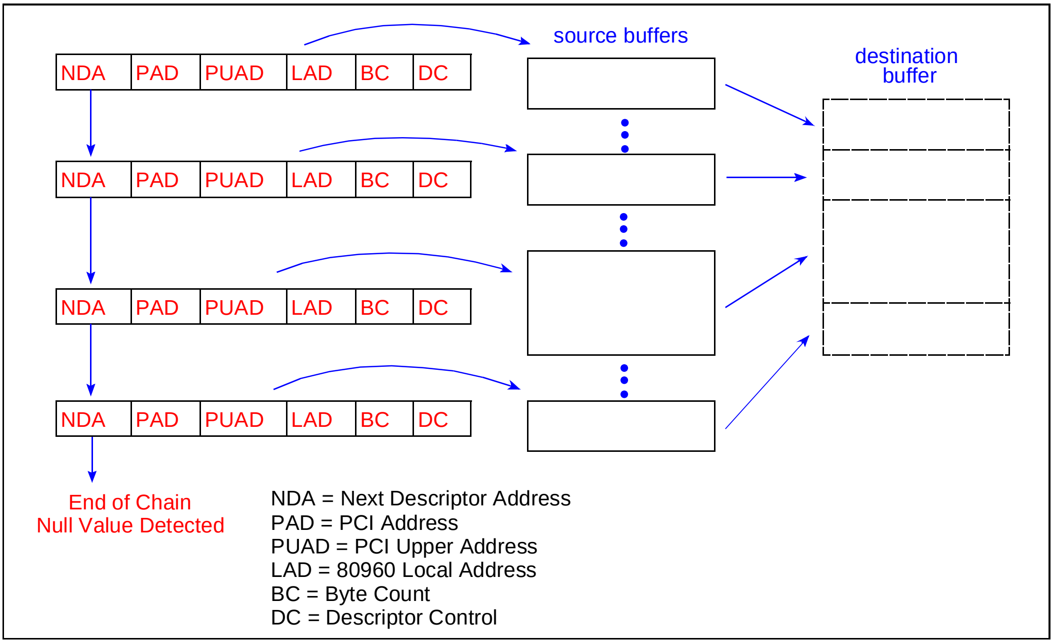


Figure 20-5. Example of Gather Chaining

20.3.4 Synchronizing a Program to Chained Transfers

Chained DMA transfers can be synchronized to a program executing on the i960 core processor through the use of processor interrupts. The channel generates an interrupt to the i960 core processor under certain conditions. They are:

- Interrupt & Continue - The channel completes the data transfer for a chain descriptor and the Next Descriptor Address Register is non-zero. When the Descriptor Control Register's Interrupt Enable bit is set, an interrupt is generated to the i960 core processor. This interrupt is for synchronization purposes only. The channel sets the CSR's End Of Descriptor Interrupt flag. Since it is not the last chain descriptor in the list, the DMA channel starts to process the next chain descriptor without requiring any processor interaction.
- End of Chain - The DMA channel completes the data transfer for a DMA chain descriptor and the Next Descriptor Address Register is zero specifying end of chain. When the Descriptor Control Register's Interrupt Enable bit is set, an interrupt is generated to the i960 core processor. The channel sets the CSR's End Of Chain Interrupt flag.
- Error - An error condition occurs during a DMA transfer. The channel halts operation on the current chain descriptor and does not proceed to the next chain descriptor.

Each chain descriptor can independently set the Descriptor Control Register's Interrupt Enable bit. This bit enables an independent channel interrupt upon completion of the data transfer for the chain descriptor. This bit can be set or clear within each chain descriptor. Control of interrupt generation within each descriptor aids in the synchronization of the executing software with the DMA transfers.

Figure 20-6 shows two examples of program synchronization. The left column shows program synchronization based on individual chain descriptors. Descriptor 1A generated an interrupt to the processor, while descriptor 2A did not because the *Interrupt Enable* bit was clear. The last descriptor *n*A, generated an interrupt to signify end of chain is reached. The right column shows an example where the interrupt was generated on the last descriptor signifying the end of chain.

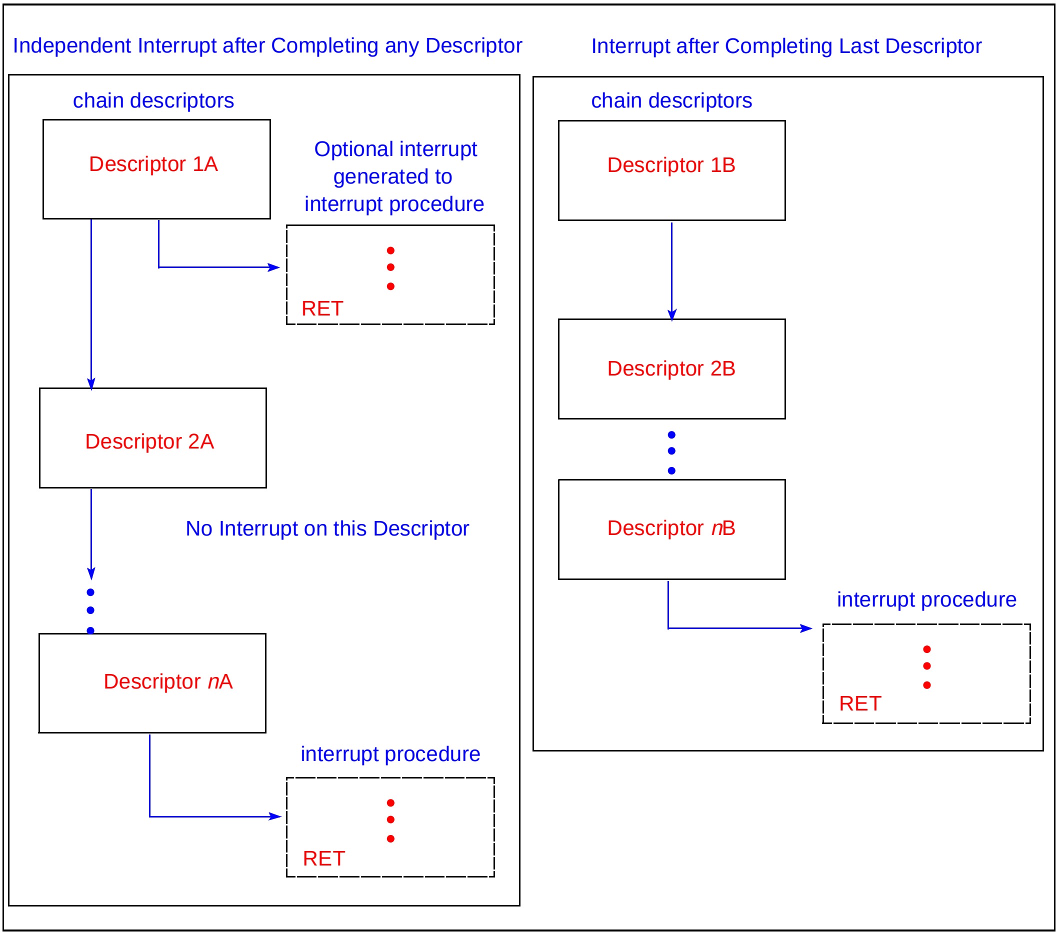

Independent Interrupt after Completing any Descriptor
chain descriptors
Descriptor 1A
Optional interrupt generated to interrupt procedure
RET
Descriptor 2A
No Interrupt on this Descriptor
Descriptor *n*A
interrupt procedure
RET
Interrupt after Completing Last Descriptor
chain descriptors
Descriptor 1B
Descriptor 2B
Descriptor *n*B
interrupt procedure
RET

Figure 20-6. Synchronizing to Chained Transfers

20.3.5 Appending to The End of a Chain

Once the channel starts processing a chain of DMA descriptors, application software may need to append a chain descriptor to the current chain without interrupting the transfer in progress. This action is controlled by the CCR Chain Resume bit.

The channel reads the entire chain descriptor each time the channel completes a chain descriptor and the Next Descriptor Address Register is non-zero.

- The Next Descriptor Address Register always contains the address of the next chain descriptor to be read
- The Descriptor Address Register always contains the current chain descriptor’s address

The procedure for appending chains requires software to find the last chain descriptor in the current chain and change the Next Descriptor Address in that descriptor to the address of the new chain. Software then sets the CCR’s *Chain Resume* bit for the channel — whether the channel is active or not.

The channel examines the CCR’s Chain Resume bit when the channel is idle or upon completion of a chain of DMA transfers. When this bit is set, the channel re-reads the Next Descriptor Address of the current chain descriptor and loads it into the Next Descriptor Address Register. The current chain descriptor’s address is contained in the Descriptor Address Register. The channel clears the *Chain Resume* bit and examines the Next Descriptor Address Register. When the Next Descriptor Address Register is not zero, the channel reads the chain descriptor using this new address and begins a new DMA transfer. When the Next Descriptor Address Register is zero, the channel remains or returns to idle.

Three cases to consider when appending a chain descriptor are:

1. The channel completes a DMA transfer and it is not the last descriptor in the chain. In this case, the channel clears the *Chain Resume* bit and reads the next chain descriptor. The appended descriptor is read when the channel reaches the end of the original chain.
2. The channel completes a DMA transfer and it is the last descriptor in the chain. In this case, the channel examines the state of the Chain Resume bit. When the bit is set, the channel re-reads the current descriptor to get the appended chain descriptor’s address, placed there by software. When the bit is clear, the channel returns to idle.
3. The channel is idle. In this case, the channel examines the Chain Resume bit state when the CCR is written. When the bit is set, the channel re-reads the last descriptor from the most-recent chain to get the appended chain descriptor placed there by the software.

20.4 DEMAND MODE DMA

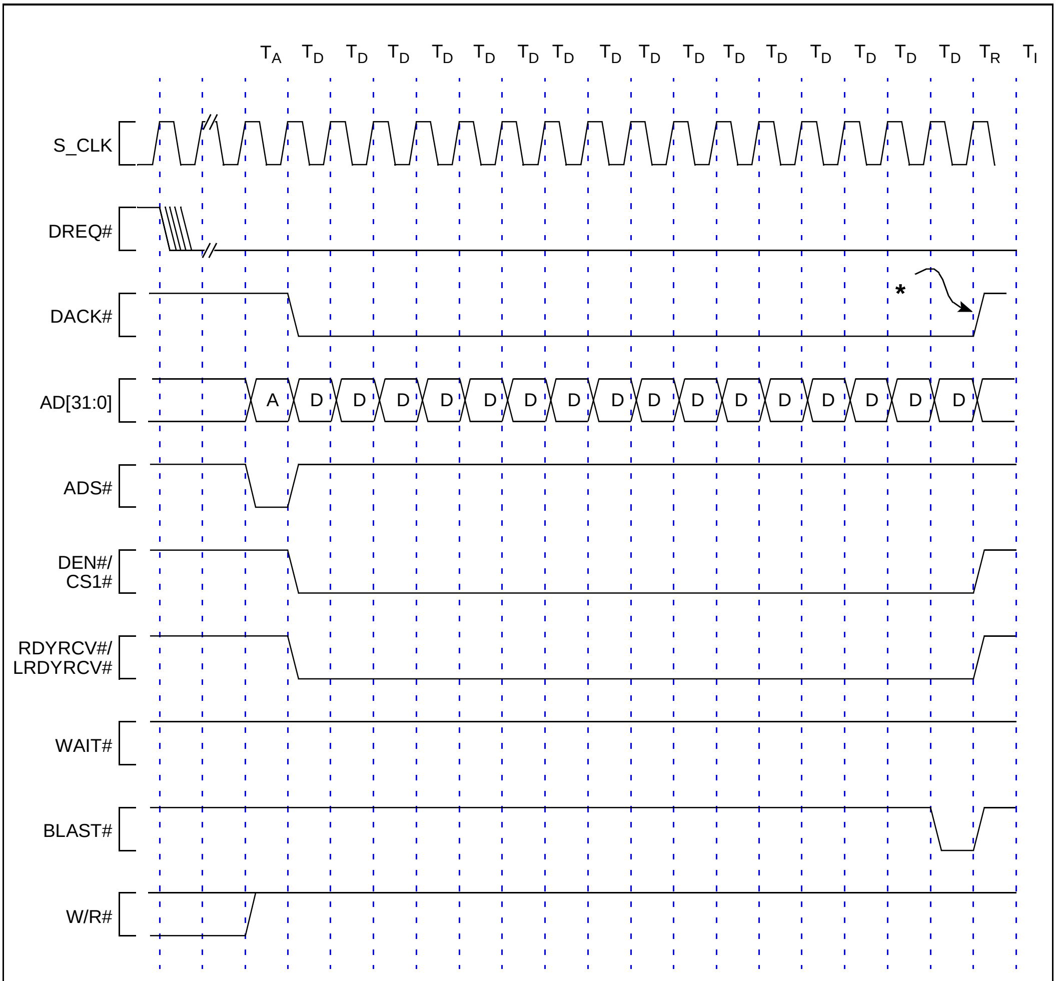
DMA controller Channel 0 provides a two pin interface which supports DMA transfers to and from 32-bit external devices on the 80960 local bus. This interface consists of a DREQ# pin which the external device asserts signifying there is new data to transfer or it has available buffers for DMA transfers into the device. The second pin, DACK#, is driven by the DMA controller to notify the device that it can receive additional data or it has data to send to the device.

The demand mode DMA transfers requires the 32-bit external device to be connected to the 80960 local bus and have the ability to support the 80960 local bus control signals through a direct interface or custom external logic. The waveforms shown in Figure 20-7 through Figure 20-14 describe the control signal interface using the DREQ# and DACK# pins.

The *Demand Mode Enable* bit in the Descriptor Control Register (refer to section 20.7.9) for channel 0 enables demand mode transfers. When demand mode is enabled, the *80960 Address Increment Hold Enable* bit in the Descriptor Control Register allows the application programmer to program the 80960 local bus address in DMA channel 0 to a fixed value. When this bit is set, the channel holds the 80960 local bus address to the same value on every burst transfer. The external device is responsible for internally keeping track of the data transfer address. Typically, holding the 80960 local bus address is used for data transfers to a port, which may contain a deep FIFO to buffer the data. The address increment hold is only available on DMA controller channel 0.

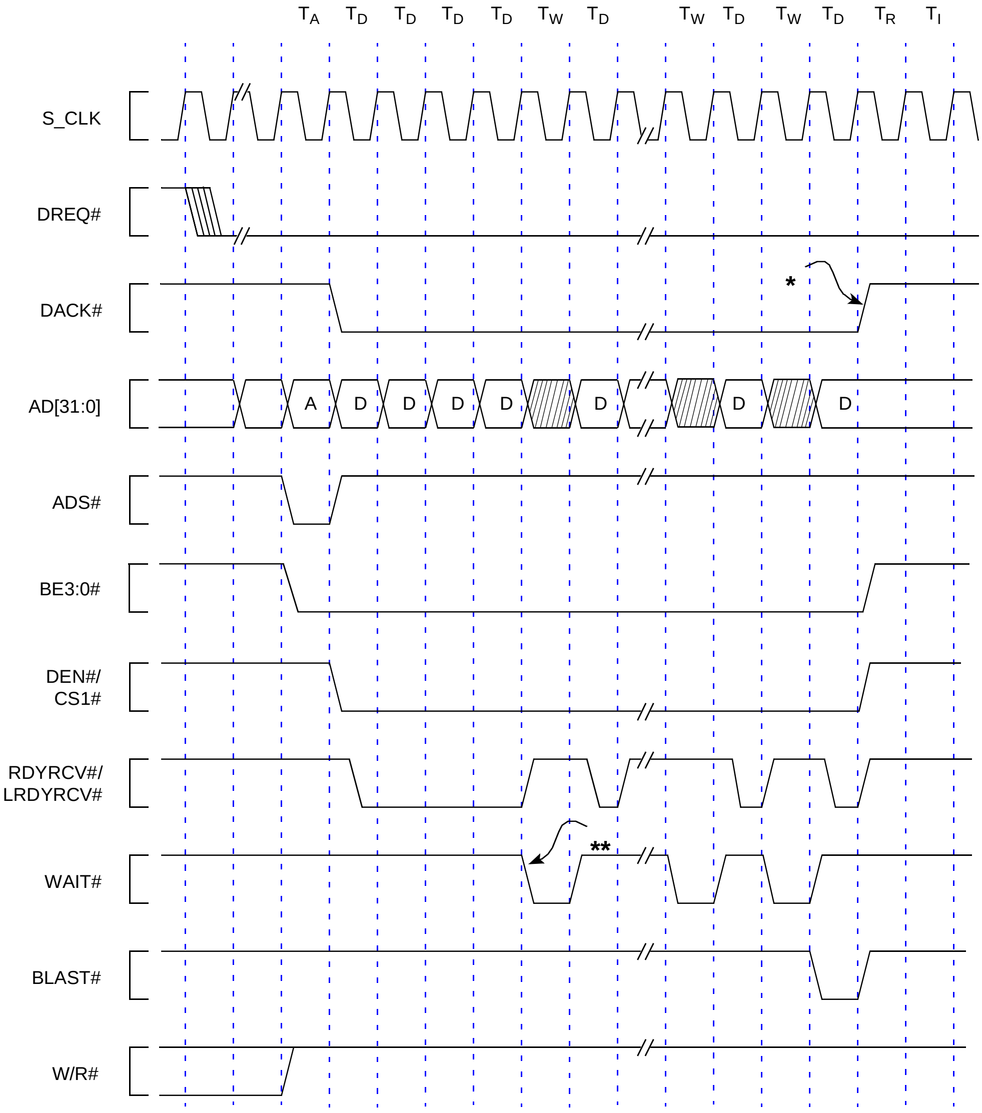
20.5 WAIT STATES INITIATED BY THE DMA CONTROLLER

The PCI bus allows PCI master and PCI slave devices to insert wait states during a burst transfer. This is done through the PCI control signals P_IRDY# and P_TRDY#. These signals can change the PCI bus's data throughput characteristics. This, in turn, requires all DMA Channels to have a similar control signal to notify the external device of the change in data rate. The WAIT# signal is generated by the DMA Controller to insert wait states in the data stream between the external device and the DMA controller. WAIT#, for the 80960 local bus, is similar in function to the PCI bus' IRDY# signal. It may assert at any time when DEN# is asserted and BLAST# is not asserted; and as long as WAIT# is asserted, LRDYRCV#/RDYRCV# is a don't care. WAIT# will not assert when the 80960 local bus is idle. This WAIT# signal is also shown in Figure 20-7 through Figure 20-14.



NOTE:
* DMA transfers one queue of data

Figure 20-7. DMA - Aligned Write to Device, Wait States, Device Always Requesting

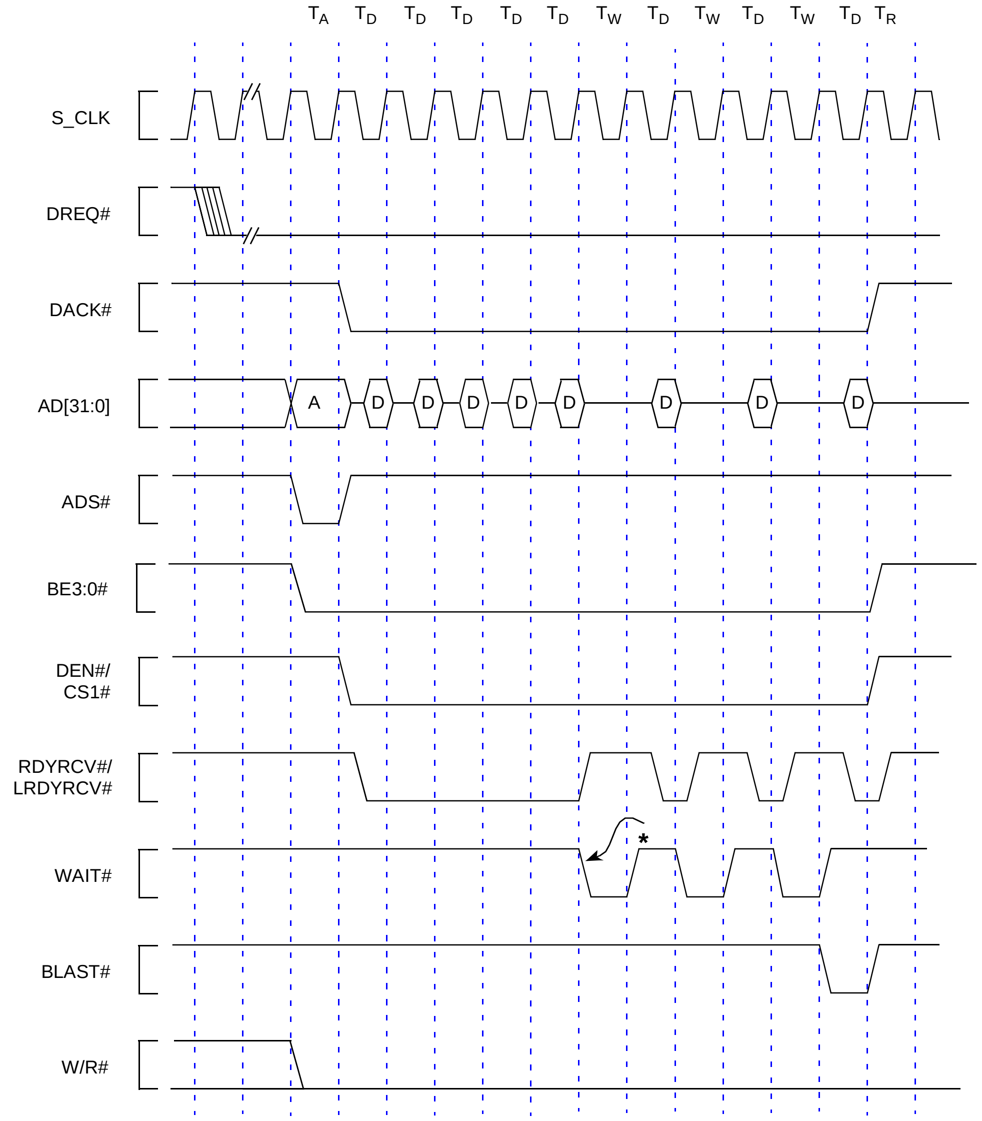


NOTE:

* DMA transfer ended - Lost PCI Bus Grant

** Wait states inserted by DMA controller after the queue was drained (data may be streaming in from the PCI bus with wait states).

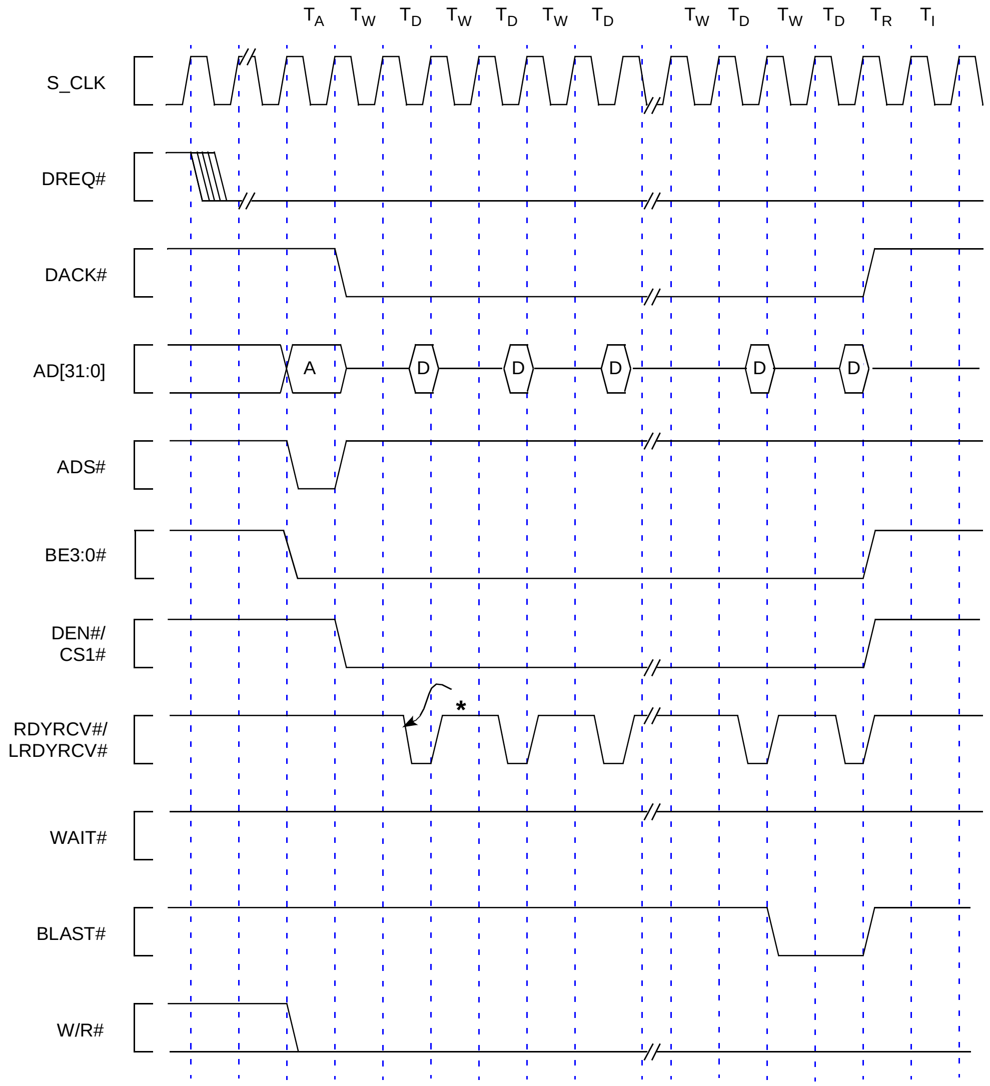
Figure 20-8. DMA - Aligned Write to Device, DMA Inserting Wait States, Device Always Requesting



NOTE:

* Wait states inserted by the DMA controller after the queue filled (data may be streaming on the PCI bus with wait states).

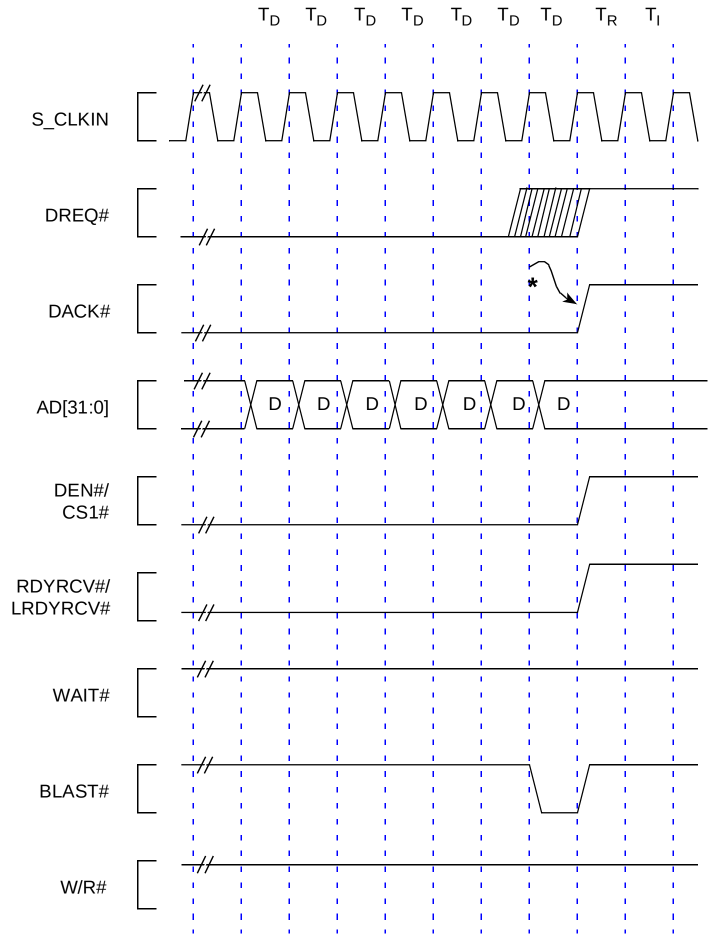
Figure 20-9. DMA - Aligned Read from Device, DMA Inserting Wait States, Device Always Requesting



NOTE:

* Wait states inserted by memory controller (LRDYRCV#) -
using memory bank #1 for connecting the peripheral device
OR
Wait states inserted by an external memory controller and driving RDYRCV#

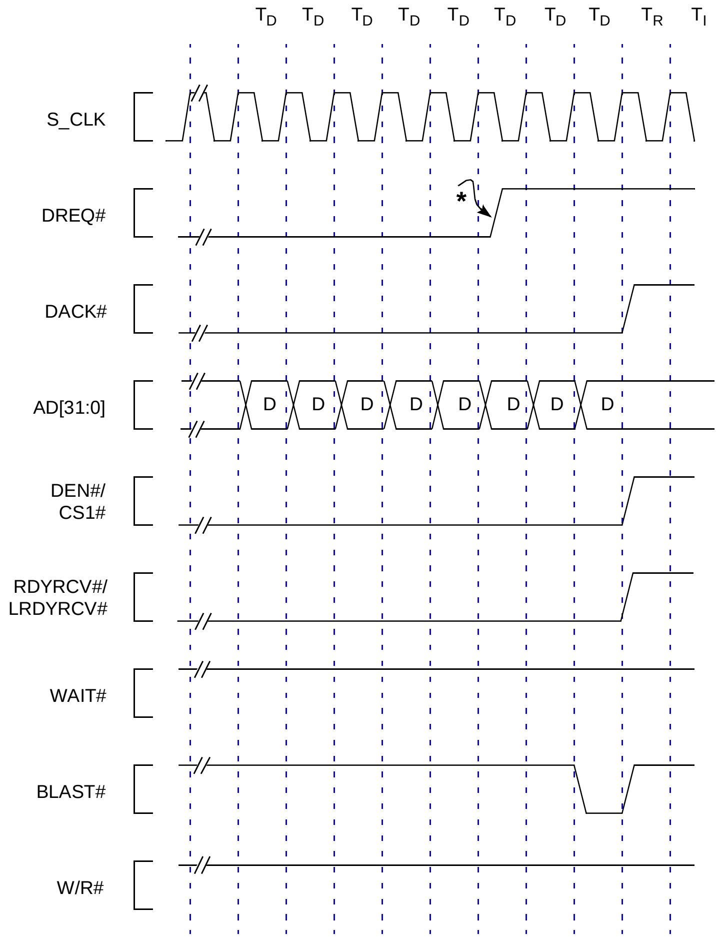
Figure 20-10. DMA - Aligned Read from Device, Device Inserting Wait States, Device Always Requesting



NOTE:

* DMA channel end transfer - DMA queue empty
and lost PCI Bus Grant (only one data transfer after deasserting DREQ#)

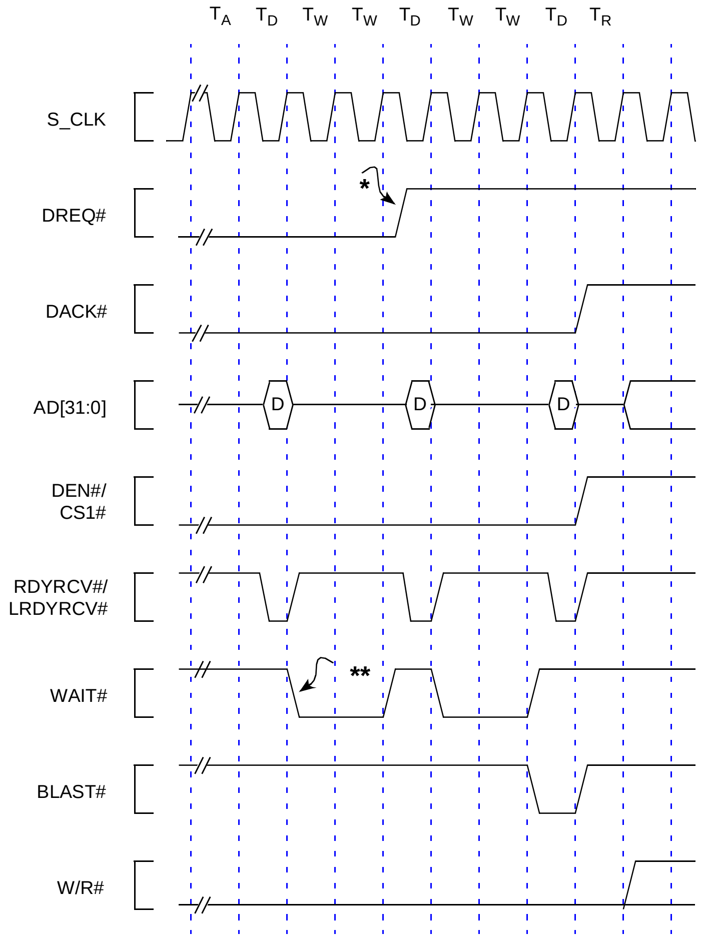
Figure 20-11. DMA - Aligned Write to Device, Zero Wait States, Device ends Transfer



NOTE:

* Device ends transfer by deasserting DREQ#
(maximum 2 data transfers)

Figure 20-12. DMA - Aligned Write to Device, Zero Wait States, Device ends Transfer

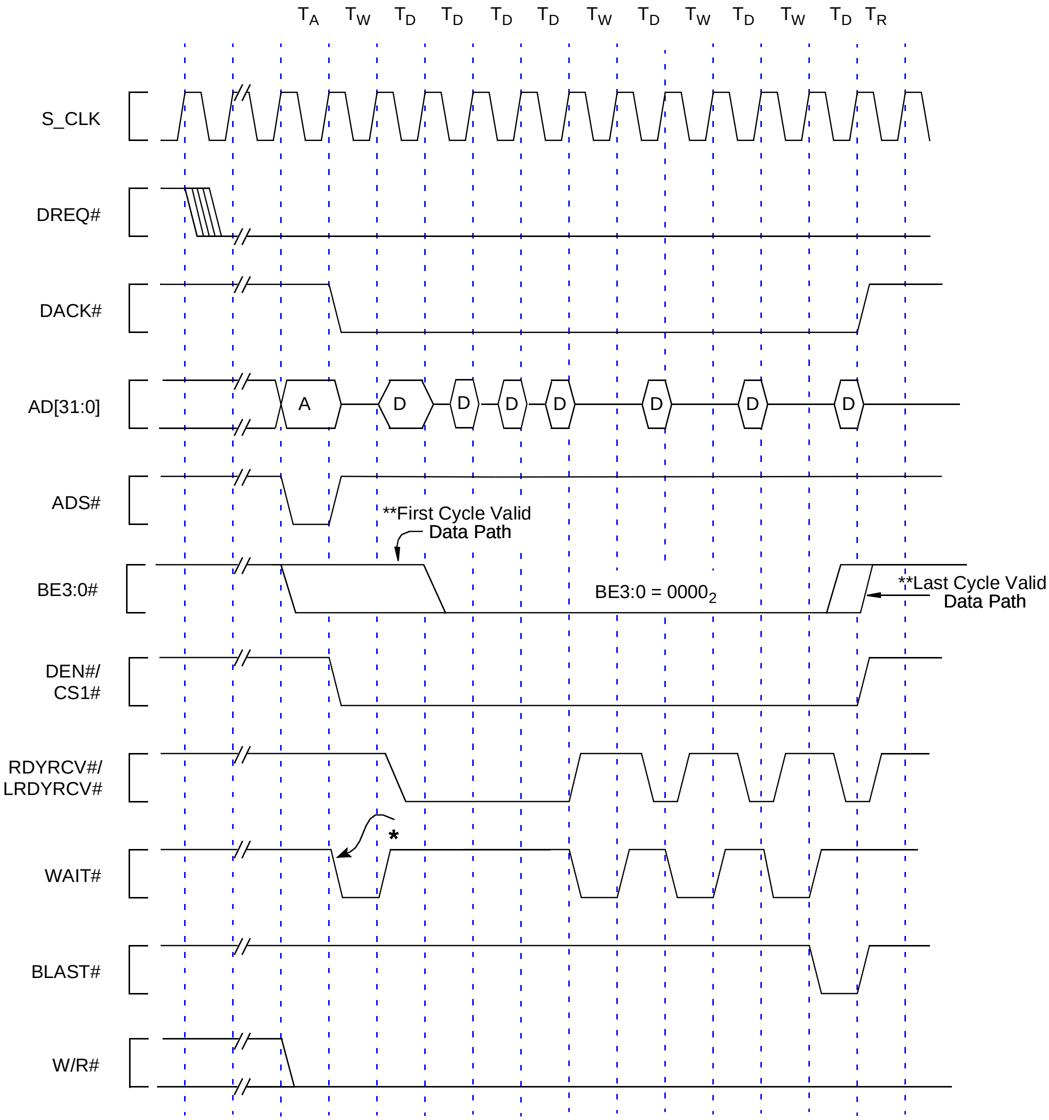


NOTE:

* Device ends transfer by deasserting DREQ# (only 1 data transfer).

** Wait states inserted by DMA channel after queue filled (data may be streaming on the PCI bus with wait states).

Figure 20-13. DMA - READ from Device, Wait States, Device ends Transfer


T_A T_W T_D T_D T_D T_D T_W T_D T_W T_D T_W T_D T_R
S_CLK
DREQ#
DACK#
AD[31:0]
A
D
D
D
D
D
D
D
ADS#
**First Cycle Valid Data Path
BE3:0#
BE3:0 = 0000_2
**Last Cycle Valid Data Path
DEN#/ CS1#
RDYRCV#/ LRDYRCV#
*
WAIT#
BLAST#
W/R#

NOTE:

* Wait states inserted by the DMA controller after the queue filled (data may be streaming on the PCI bus with wait states).

** For First and Last Cycle Valid Data Path bit settings.

Figure 20-14. DMA - Unaligned Read from Device, DMA Inserting Wait States, Device Always Requesting

20.6 DATA TRANSFERS

The DMA controller is optimized to perform data transfers between the PCI bus and local memory. The DMA channels issue both read and write accesses to the PCI bus and the 80960 local bus. The DMA channels have bus mastering capabilities only. The same condition applies to the 80960 local bus interface. These transfers are summarized in the following sections.

20.6.1 PCI to Local Memory Transfers

PCI to local memory transfers perform read cycles on the PCI bus and place the data into the DMA channel queues. Once the first data is placed into the queue, the DMA channel's local bus interface requests the local bus and drains the queue by writing the data to local memory.

Application software can use the various PCI command types to improve system performance for these transfers. The three defined PCI read commands include: Memory Read, Memory Read Line, and Memory Read Multiple. Refer to the *PCI Local Bus Specification,* revision 2.1 for full PCI command descriptions.

For example, a Memory Read Multiple command can be programmed when the block size is larger than a cache line. This notifies the PCI target that the DMA channel intends to transfer a large block of data and the target should try to read ahead and anticipate the DMA controller read requests. Application software can select which command type is best to satisfy system requirements.

The following describes a DMA transfer from the PCI bus to the 80960 local memory:

- The DMA channel requests the PCI bus. Once the DMA channel has at least one WORD in the queue, it asserts the request for the 80960 local bus.
- The DMA channel reads data from the PCI bus and fills the channel queue. When the 80960 local bus has not been granted and the queues become full, it removes the request from the PCI bus and continue to request the 80960 local bus.
- When the DMA channel reaches a byte count of zero while filling the queues or reaches a queue full condition before acquiring the 80960 local bus, the DMA channel ends the data transfer on the PCI bus, and removes the request for the PCI bus.
- When the DMA channel acquires the 80960 local bus while filling the queues, the DMA channel transfers data from the channel queue to local memory. At the same time, the DMA channel continues to request data on the PCI bus. This continues until one of four conditions occur:
    - loss of PCI bus ownership
    - loss of 80960 local bus ownership
    - PCI bus error condition
    - 80960 local bus error condition

  - When the DMA channel reaches a 2 Kbyte address boundary, the DMA controller stops the current 80960 local bus transaction

NOTE: The loss of ownership on the PCI bus is determined by ATU latency timer expiration and the removal of the PCI grant signal. Loss of ownership on the 80960 local bus is determined solely by the removal of the grant signal.

- Upon losing the PCI bus, the DMA channel completes the current data transfer in progress, terminates gracefully and removes the request on the 80960 local bus.
- Upon losing the 80960 local bus, the DMA channel completes the current data transfer in progress, terminates gracefully and removes the request on the PCI bus.
- Error conditions on either bus terminates data transfers on both interfaces, sets the corresponding bit in the status register, and generates an interrupt to the i960 core processor.

20.6.2 Local Memory to PCI Transfers

Local memory to PCI transfers perform read cycles on the local bus and place the data into the DMA channel queues. Once the first data is placed into the queue, the DMA channel's PCI bus interface requests the PCI bus and drains the queue by writing data to the PCI bus.

Local memory to PCI transfers can generate two different PCI write commands: Memory Write, and Memory Write and Invalidate. The application software can use these PCI command types to improve system performance for these types of transfers.

Memory Write commands can be used for all data transfers to the PCI bus. There are no restrictions for these transfers and both bus interfaces are optimized for full 132 Mbytes/sec bandwidth. However, the PCI target may provide better system performance by using the Memory Write and Invalidate command.

The following describes a DMA transfer from 80960 local memory to the PCI bus:

- The DMA channel requests the 80960 local bus. Once the DMA channel has at least one WORD in the queue, it asserts the request for the PCI bus.
- The DMA channel reads data from local memory and fills the channel queue. If the PCI bus has not been granted and the queues become full, it removes the request from the 80960 local bus and continues to request the PCI bus. When the DMA channel reaches a 2 Kbyte address boundary, the DMA controller stops the current local bus transaction.
- When the DMA channel reaches a byte count of zero while filling the queues or reaches a queue full condition before acquiring the PCI bus, the DMA channel ends the data transfer and removes the request for the 80960 local bus.

- When the DMA channel acquires the PCI bus while filling the queues, the DMA channel transfers data from the channel queue to the PCI bus. At the same time, the DMA channel continues requesting data from local memory. This continues until one of four conditions occur: loss of PCI bus ownership, loss of 80960 local bus ownership, PCI bus error condition, 80960 local bus error condition.

NOTE: The loss of PCI bus ownership is determined by the ATU latency timer expiration and the removal of the PCI grant signal. Loss of 80960 local bus ownership is determined by the local arbitration described in CHAPTER 18, BUS ARBITRATION.

- Upon losing the PCI or 80960 local bus, the DMA channel completes the current data transfer in progress and terminates gracefully. The only exception is for the Memory Write and Invalidate cycle type. The DMA channel meets the requirements specified by the PCI local bus specification. For Memory Write and Invalidate, the DMA channel continues data transfers until reaching the next cacheline size boundary specified by the ATU Cacheline Size Register.
- Error conditions on either bus terminate data transfers on both interfaces, sets the corresponding status register bit, and generates an interrupt to the i960 core processor.

20.6.3 Local Memory to PCI Transfers using Memory Write and Invalidate

The second mechanism for performing local memory to PCI transfers may improve system performance based on the PCI target capabilities.

NOTE: Using the Memory Write and Invalidate (MWI) command improves system performance when the target is cacheable memory.

The DMA channel attempts to use the Memory Write and Invalidate command on the PCI bus when programmed by application software. However, a number of circumstances may prevent the DMA channel from actually initiating the MWI command. If any of the following three conditions are *not* met, the channel converts the MWI command to a Memory Write command for the complete DMA transfer:

1. The ATU Cacheline Size Register (ATUCLSR), located in ATU configuration space, must have a valid value other than zero. This register is programmed by host software.
2. The ATUCLSR must have a legal value which is less than or equal to the number of queue entries in the DMA channel queue. (The channel must guarantee an entire cache line can be transferred during an MWI bus transaction).
3. The Memory Write and Invalidate Enable bit must be set in either the:

   3.1. For Channels 0 and 1: Primary ATU Command Register

   3.2. For Channel 2: Secondary ATU Command Register

When the above conditions are met, the DMA channel provides full Memory Write and Invalidate support. For example, to transfer an 80 byte block to a PCI address of 8001CH while the ATUCLSR is 8 DWORDs, the DMA channel performs three PCI transactions:

1. Transfer of 4 bytes at address 8001CH using the Memory Write command.
2. Transfer of 64 bytes at address 80020H using the MWI command.
3. Transfer of 12 bytes at address 80060H using the Memory Write command.

20.6.4 Exclusive Access

The DMA Controller does not support exclusive access through the PCI LOCK# signal.

20.7 REGISTER DEFINITIONS

The DMA controller contains registers for controlling each channel. Each channel has nine memory-mapped control registers for independent operation. The CCR, CSR, and the Next Descriptor Address Registers have a read/write access. All other DMA registers are read-only and are loaded with new values from the chain descriptor when the channel reads a chain descriptor from memory.

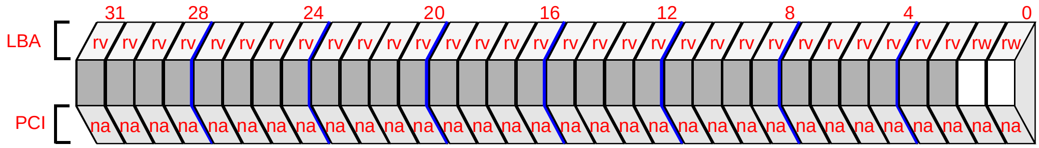
Table 20-2. DMA Controller Register Summary

Section	Register Name - Acronym	Page	Size (Bits)	DMA Channel	80960 Local Bus Address	PCI Config Addr Offset
20.7.1	Channel Control Register - CCRx	20-25	32	0 1 2	0000 1400H 0000 1440H 0000 1480H	NA
20.7.2	Channel Status Register - CSRx	20-26	32	0 1 2	0000 1404H 0000 1444H 0000 1484H	NA
20.7.3	Descriptor Address Register - DARx	20-28	32	0 1 2	0000 140CH 0000 144CH 0000 148CH	NA
20.7.4	Next Descriptor Address Register - NDARx	20-29	32	0 1 2	0000 1410H 0000 1450H 0000 1490H	NA
20.7.5	PCI Address Register - PADRx	20-30	32	0 1 2	0000 1414H 0000 1454H 0000 1494H	NA
20.7.6	PCI Upper Address Register - PUADRx	20-31	32	0 1 2	0000 1418H 0000 1458H 0000 1498H	NA
20.7.7	80960 Local Address Register - LADRx	20-32	32	0 1 2	0000 141CH 0000 145CH 0000 149CH	NA
20.7.8	Byte Count Register - BCRx	20-33	32	0 1 2	0000 1420H 0000 1460H 0000 14A0H	NA
20.7.9	Descriptor Control Register - DCRx	20-34	32	0 1 2	0000 1424H 0000 1464H 0000 14A4H	NA

20.7.1 Channel Control Register - CCRx

The Channel Control Register (CCR) specifies parameters that dictate the overall channel operating environment. The CCR should be initialized prior to any other DMA register following a system reset. This register can be read or written while the DMA channel is active.

Table 20-3. Channel Control Register - CCRx


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: CH.0-1400H
CH.1-1440H
CH.2-1480H
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

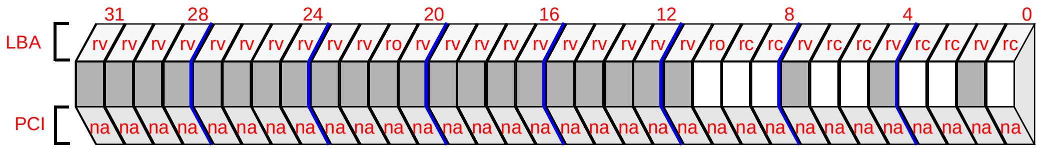
Bit	Default	Description
31:02	0000 0000H	Reserved.
01	0_2	Chain Resume - when set, causes the channel to resume chaining by re-reading the current descriptor located at the address in the Descriptor Address Register when the channel is idle (Channel Active bit in the CSR is clear) or when the channel completes a DMA transfer. This bit is cleared by the hardware when either: • The channel completes a DMA transfer and the Next Descriptor Address Register is zero. In this case, the channel proceeds to the next descriptor in the chain. • The channel re-reads the chain descriptor located at the address in the Descriptor Address Register and loads the Next Descriptor Address of that descriptor into the Next Descriptor Address Register
00	0_2	Channel Enable - When set, the channel enables DMA transfers. When clear, the channel disables DMA transfers. Clearing this bit when the channel is active immediately suspends the current DMA transfer by halting all local bus transactions. The PCI interface may continue with the current transfer until the data queue either fills or empties. The channel does not initiate any new DMA transfers when this bit is cleared. Data held in queues remains valid. Setting this bit after the channel is suspended causes the channel to resume the DMA transfer. The Channel Enable bit works in conjunction with the Bus Master Enable bit of the Primary ATU Command Register for DMA Channel 0 and 1 and with the Bus Master Enable bit of the Secondary ATU Command Register for DMA Channel 2. The respective Bus Master Enable bit must be set for the DMA channel to start a transaction on the PCI bus.

20.7.2 Channel Status Register - CSRx

The Channel Status Register (CSRx) contain status flags that indicate the channel status (see Table 20-4). This register is typically read by software to examine the source of an interrupt. See section 20.8, INTERRUPTS (pg. 20-36) for a description of DMA channel interrupts.

When a DMA error occurs, application software should check the status of Channel Active flag before processing the interrupt. It is possible that the channel may still be completing any outstanding PCI transactions.

Table 20-4. Channel Status Register - CSRx (Sheet 1 of 2)

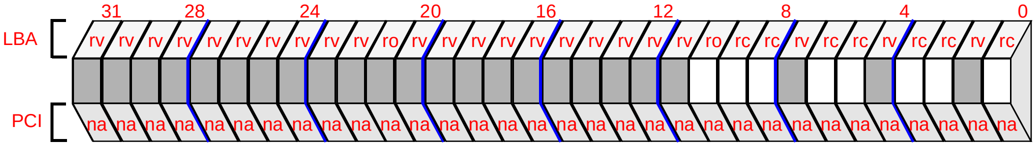

31 28 24 20 16 12 8 4 0
LBA
rv rv rv rv rv rv rv rv rv rv ro rv rv rv rv rv rv rv rv rv rv ro rc rc rv rc rc rv rc rc rv rc
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH.0-1404H
	CH.1-1444H
	CH.2-1484H
		Legend: NA = Not Accessible, RO = Read Only
		RV = Reserved, PR = Preserved, RW = Read/Write
		RS = Read/Set, RC = Read Clear
		LBA = 80960 Local Bus Address, PCI = PCI Configuration Address Offset

PCI:	NA	

Bit	Default	Description
31:11	0000 00H	Reserved.
10	0_2	Channel Active Flag - indicates the channel is either active (in use) or inactive (available). When set, indicates the channel is in use and actively performing DMA data transfers. When clear, indicates the channel is inactive and available to be configured to transfer data. The channel clears the Channel Active flag when the previously configured DMA transfer completes as a result of: • byte count reached zero and last chain descriptor is encountered (NULL value detected for Next Descriptor Address in chain descriptor) • PCI Master-abort occurred on the PCI interface • PCI Target-abort occurred on the PCI interface • PCI parity error occurred on the PCI interface • 80960 parity error signalled from the Memory Controller • 80960 local bus fault signalled from the Memory Controller The Channel Active flag is set when a Chain Descriptor is read from memory.
09	0_2	End of Transfer Interrupt Flag - set when the channel has signalled an interrupt to the i960 core processor after successfully completing an error-free DMA transfer but it is not the last descriptor in a chain.
08	0_2	End of Chain Interrupt Flag - set when the channel has signalled an interrupt to the i960 core processor after successfully completing an error-free DMA transfer that is the last of a chain.
07	0_2	Reserved.

Table 20-4. Channel Status Register - CSRx (Sheet 2 of 2)


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv ro rv rv rv rv rv rv rv rv rv rv ro rc rc rv rc rc rv rc rc rv rc
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: CH.0-1404H
CH.1-1444H
CH.2-1484H

PCI: NA

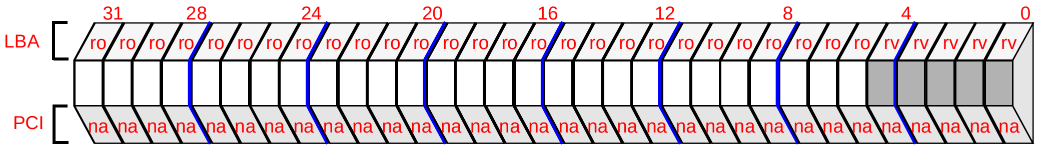
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
06	0_2	80960 Memory Fault Error Flag - set when the channel detects a parity error when reading data from the 80960 local bus or when reading the Chain Descriptor or NDAR value. The Memory Controller verifies data parity (when enabled) on memory reads from the 80960 local bus and notifies the DMA Controller upon detecting invalid parity.
05	0_2	80960 local bus Fault Error Flag - set when the channel detects a Bus Fault when attempting to read or write data to the 80960 local bus or when reading the Chain Descriptor or NDAR value.
04	0_2	Reserved.
03	0_2	PCI Master Abort Flag - set when the channel has initiated a transaction on the PCI bus and has detected a Master-abort.
02	0_2	PCI Target Abort Flag - set when the channel has initiated a transaction on the PCI bus and has detected a Target-abort.
01	0_2	Reserved.
00	0_2	PCI Parity Error Flag - is set when the following three conditions are met: • DMA channel asserted PERR# or has observed PERR# asserted • DMA channel was the master for the transaction in which the error occurred • Parity Checking Enable bit is in the PATUCMD is set (for channel 0 and 1) or in the SATUCMD (for channel 2)

20.7.3 Descriptor Address Register - DARx

The Descriptor Address Register (DARx, Table 20-5) contains the current chain descriptor's address in 80960 local memory for a DMA transfer. This read-only register is loaded when a new chain descriptor is read. All chain descriptors are required to be aligned on an eight 32-bit word boundary.

Table 20-5. Descriptor Address Register - DARx


LBA
PCI
31
28
24
20
16
12
8
4
0
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro rv rv rv rv rv
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH.0-140CH
	CH.1-144CH
	CH.2-148CH
		Legend:
		RV = Reserved
		RS = Read/Set
		LBA = 80960 Local Bus Address
			NA = Not Accessible RO = Read Only
			PR = Preserved RW = Read/Write
			RC = Read Clear
			PCI = PCI Configuration Address Offset

PCI:	NA		

Bit	Default	Description
31:05	0000 000H	Current Descriptor Address - 80960 local bus memory address of the current chain descriptor that was read by the channel.
04:00	00H	Reserved.

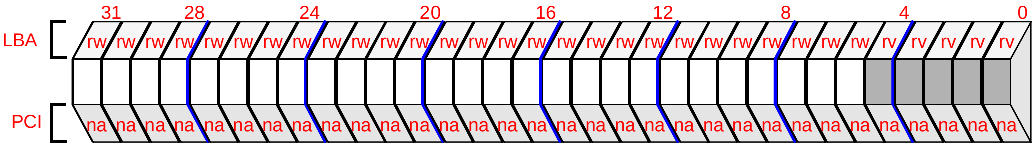
20.7.4 Next Descriptor Address Register - NDARx

The Next Descriptor Address Register (NDARx, see Table 20-6) contains the address of the next chain descriptor in 80960 local memory for a DMA transfer. When starting a DMA transfer, this register contains the first chain descriptor's address.

All chain descriptors are required to be aligned on an eight 32-bit word boundary. The channel may set bits 04:00 to zero when loading this register.

NOTE: The CCR Channel Enable bit and CSR Channel Active bit must both be clear prior to writing the Next Descriptor Address Register. Writing a value to this register while the channel is active may result in undefined behavior.

Table 20-6. Next Descriptor Address Register - NDARx


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rv rv rv rv rv
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: CH.0-1410H
CH.1-1450H
CH.2-1490H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:05	0000 000H	Next Descriptor Address - 80960 local bus memory address of the next chain descriptor to be read by the channel.
04:00	00H	Reserved.

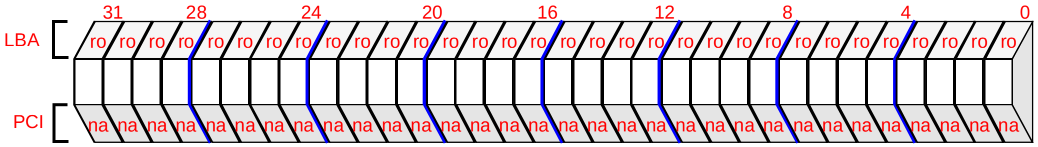
20.7.5 PCI Address Register - PADRx

The PCI Address Register (PADR, Table 20-7) contains the 32-bit PCI address for SAC cycles or the lower 32-bit PCI address of a 64-bit PCI address for DAC cycles. This address is the DMA transfer's source or destination. This read-only register is loaded when a chain descriptor is read from memory.

The channel drives PAD1:0 or SAD1:0 to a value of 00_2 indicating linear or sequential addressing. Refer to the *PCI Local Bus Specification*, revision 2.1 for additional information.

NOTE: Application software must not program the channel to transfer data across a 4 Gbyte boundary (i.e., the lower 32-bit address must not increment past the maximum address of FFFF FFFFH). The channel does not notify the application of this condition.

Table 20-7. PCI Address Register - PADRx


31
28
24
20
16
12
8
4
0
LBA
PCI
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH.0-1414H
	CH.1-1454H
	CH.2-1494H
		Legend:
		RV = Reserved
		RS = Read/Set
		LBA = 80960 Local Bus Address
			NA = Not Accessible
			PR = Preserved
			RC = Read Clear
			PCI = PCI Configuration Address Offset

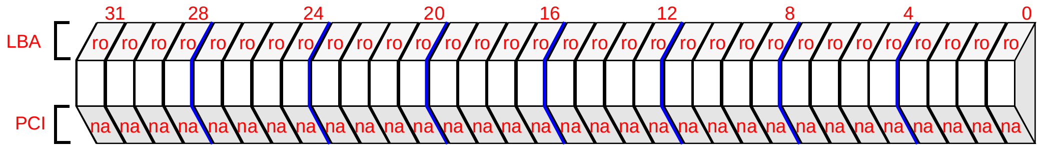
PCI:	NA		RO = Read Only
			RW = Read/Write

Bit	Default	Description
31:00	0000 0000H	PCI Address - is the PCI source/destination address.

20.7.6 PCI Upper Address Register - PUADRx

The PCI Upper Address Register (PUADRx, Table 20-8) contains the upper 32-bit address of a 64-bit address. This register is read-only and is loaded when a chain descriptor is read from memory.

Table 20-8. PCI Upper Address Register - PUADRx


31
28
24
20
16
12
8
4
0
LBA
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH.0-1418H	**Legend:**	NA = Not Accessible	RO = Read Only
	CH.1-1458H	RV = Reserved	PR = Preserved	RW = Read/Write
	CH.2-1498H	RS = Read/Set	RC = Read Clear	
PCI:	NA	LBA = 80960 Local Bus Address	PCI = PCI Configuration Address Offset	

Bit	Default	Description
31:00	0000 0000H	PCI Upper Address - is the PCI source/destination upper address.

20.7.7 80960 Local Address Register - LADRx

The 80960 Local Address Register (LADRx, Table 20-9) contains the 32-bit 80960 local bus address. The 80960 local bus address space is a 32-bit, byte addressable address space. This register is read-only and is loaded when a chain descriptor is read from memory.

NOTE: Access to the Peripheral Memory-Mapped Registers through a DMA transfer is not allowed. Do not program LADRx with values less than 1800H; this address space is reserved. Hardware must ensure that local bus accesses to this space are properly terminated.

Table 20-9. 80960 Local Address Register - LADRx

	31			28				24				20				16				12				8				4				0
LBA	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

LBA:	CH.0-141CH	**Legend:**	NA = Not Accessible	RO = Read Only
	CH.1-145CH	RV = Reserved	PR = Preserved	RW = Read/Write
	CH.2-149CH	RS = Read/Set	RC = Read Clear	
PCI:	NA	LBA = 80960 Local Bus Address	PCI = PCI Configuration Address Offset	

Bit	Default	Description
31:00	0000 0000H	80960 local bus address - the 80960 local bus source/destination address.

20.7.8 Byte Count Register - BCRx

The Byte Count Register contains the number of bytes to transfer for a DMA transfer. This is a read-only register that is loaded from the Byte Count word in a chain descriptor. It allows for a maximum DMA transfer of 16 Mbytes. A value of zero is a valid byte count and results in no data words being transferred and no cycles generated on either the PCI bus or the 80960 local bus.

When the i960 core processor reads this register, it contains the number of bytes left to transfer on the 80960 local bus. The channel's data queue may contain valid data. This register decrements by 1, 2, 3 or 4 for each successful operand transfer from the source to destination locations.

- When the operand size is byte, the register byte count decrements by 1
- When the operand is a 2-byte transfer, the byte count decrements by 2
- When the operand is a 3-byte transfer, the byte count decrements by 3
- When the operand is a word (32-bit data) the byte count decrements by 4

NOTE: The byte count value is not required to be aligned to a 32-bit word boundary (i.e., the byte count value can be a word aligned, short aligned, or byte aligned).

Table 20-10. Byte Count Register - BCRx

	31			28				24				20				16				12				8				4				0
LBA	rv	rv	rv	rv	rv	rv	rv	rv	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

LBA:	CH.0-1420H	**Legend:**	NA = Not Accessible	RO = Read Only
	CH.1-1460H	RV = Reserved	PR = Preserved	RW = Read/Write
	CH.2-14A0H	RS = Read/Set	RC = Read Clear	
PCI:	NA	LBA = 80960 Local Bus Address	PCI = PCI Configuration Address Offset	

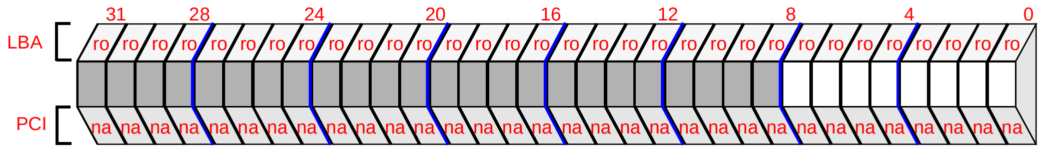
Bit	Default	Description
31:24	00H	Reserved
23:00	00 0000H	Byte Count - is the number of bytes to transfer for a DMA transfer.

20.7.9 Descriptor Control Register - DCRx

The Descriptor Control Register (DCR, Table 20-11) contains control values for the DMA transfer on a per-chain descriptor basis. These values may vary from chain descriptor to chain descriptor.

Table 20-12 lists the PCI commands that are supported and not supported for DCR bits 3:0.

Table 20-11. Descriptor Control Register - DCRx


LBA
PCI
31
28
24
20
16
12
8
4
0
ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA:	CH.0-1424H
	CH.1-1464H
	CH.2-14A4H
		Legend:
		RV = Reserved
		RS = Read/Set
			NA = Not Accessible
			PR = Preserved
			RC = Read Clear
				RO = Read Only
				RW = Read/Write

PCI:	NA	LBA = 80960 Local Bus Address	PCI = PCI Configuration Address Offset	

Bit	Default	Description
31:08	0000 00H	Reserved
07	0_2	80960 local bus address Increment Hold Enable - instructs DMA Channel 0 to hold the 80960 local bus address at a fixed value. This bit works in conjunction with demand mode DMA (section 20.4) and is ignored when the Demand Mode Enable bit is clear. When set, Channel 0 holds the 80960 local bus address to the fixed value in the 80960 Local Address Register. When clear, Channel 0 increments the 80960 local bus address on every byte transferred.
06	0_2	Demand Mode Enable - enables DMA Channel 0 to use the demand mode DMA interface for data transfers between an external device on the 80960 local bus and the PCI bus. When set, Channel 0 samples DREQ# to determine when the external device has data to transfer. When the channel is ready to transfer data, it asserts DACK# to notify the transfer is in progress. Refer to section 20.4, DEMAND MODE DMA. When clear, demand mode DMA transfers are disabled.
05	0_2	Dual Address Cycle Enable - determines the address cycle type generated on the PCI bus. When set, the channel uses Dual Address Cycle (DAC) to transfer a 64-bit address. When clear, the channel uses Single Address Cycle (SAC) to transfer a 32-bit address. For DAC, the PCI Address Register (PADRx) contains the lower 32-bit address used on the first address cycle. The PCI Upper Address Register (PUADRx) contains the upper 32 bits address cycle used on the second address cycle. The upper 32 bit address of a DAC transaction is required to be non-zero.
04	0_2	Interrupt Enable - when set, the channel generates an interrupt to the i960 core processor upon completion of this DMA transfer. When clear, no interrupt is generated.
03:00	0H	PCI Command - determines PCI bus command type on the PCI bus for this DMA transfer. This value is used directly for the PCI bus command; e.g., when PCI Command is 0000_2, the PCI Command is 0000_2, a reserved command type. See Table 20-12. Hardware does not check for reserved or unsupported command types.

Table 20-12. PCI Commands

C/BE3:0#	PCI Command Type	Description
0000_2	Reserved	Not Supported
0001_2	Reserved	Not Supported
0010_2	I/O Read	Not Supported
0011_2	I/O Write	Not Supported
0100_2	Reserved	Not Supported
0101_2	Reserved	Not Supported
0110_2	Memory Read	Memory Read of less than one cacheline
0111_2	Memory Write	Memory Write
1000_2	Reserved	Not Supported
1001_2	Reserved	Not Supported
1010_2	Configuration Read	Not Supported
1011_2	Configuration Write	Not Supported
1100_2	Memory Read Multiple	Memory Read of more than one cacheline
1101_2	Reserved	Not Supported
1110_2	Memory Read Line	Memory Read of one cacheline
1111_2	Memory Write and Invalidate	Memory Write which guarantees the transfer of a complete cache line during the current transaction

20.8 INTERRUPTS

Each channel can generate an interrupt to the i960 core processor. The Descriptor Control Register's Interrupt Enable bit (DCRx.ie) determines when the channel generates an interrupt upon successful error-free completion of a DMA transfer. Each channel has one interrupt output connected to the PCI and Peripheral Interrupt Controller described in CHAPTER 8, INTERRUPTS. Table 20-13 summarizes the conditions when interrupts are generated and status flags found in the Channel Status Register (CSRx).

Table 20-13. DMA Interrupt Summary

Interrupt Condition	Channel Status Flags								Interrupt Generated?	
	Active	End of Descriptor	End of Chain	PCI Master Abort	PCI Target Abort	PCI Parity Error	Local Bus Parity Error	Local Bus Fault Error	DCR.ie Set	DCR.ie Clear
Byte count == 0 && NDARx != NULL (End of Transfer)	1	1	0	0	0	0	0	0	Y	N
Byte Count == 0 && NDARx == NULL (End of Chain)	0	0	1	0	0	0	0	0	Y	N
PCI Master-abort	0	0	0	1	0	0	0	0	Y	Y
PCI Target-abort	0	0	0	0	1	0	0	0	Y	Y
PCI Parity Error	0	0	0	0	0	1	0	0	Y	Y
Local Bus Parity Error	0	0	0	0	0	0	1	0	Y	Y
Local Bus Fault Error	0	0	0	0	0	0	0	1	Y	Y

When abort or error interrupt conditions occur, the channel terminates data transfers for the current chain descriptor and clears the CSR Channel Active flag. The channel invalidates or clears any data in the channel data queues and does not read any new chain descriptors. The channel signals an interrupt to the i960 core processor and stops. The channel sets the appropriate error flag in the CSR. For PCI errors, the channel takes the appropriate actions on the PCI bus specified by the control bits found in the ATU Control Register (ATUCR). During an MWI transaction, the channel completes the cache line transfer before stopping. Refer to CHAPTER 16, ADDRESS TRANSLATION UNIT for additional information on the PCI error conditions.

The channel cannot restart a DMA transfer after an error condition. Software must configure the channel to complete the remaining transfers, if any.

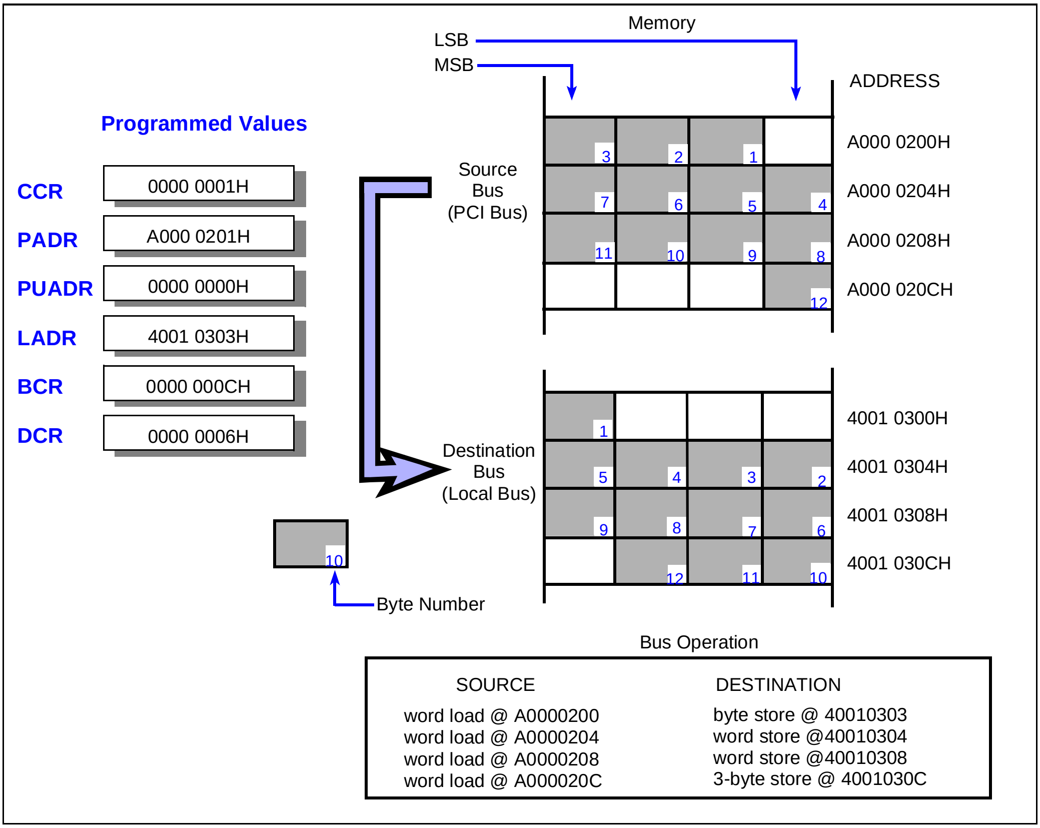
For local bus parity errors, data with incorrect parity is never transferred to the PCI bus. For PCI parity errors, data with incorrect parity is never transferred to the local memory.

When a Memory Fault Error or Bus Fault Error occurs while reading the Chain Descriptor or Next Descriptor Address, the channel sets the appropriate CSR error flag, loads the CCRs (if possible), and stops.

NOTE: The channel never reports an End of Descriptor Interrupt or End of Chain Interrupt along with any PCI error condition. End of Descriptor Interrupt and End of Chain Interrupt can only be reported in the CSR when the DMA transfer completes without any reportable errors. However, multiple error conditions may occur and be reported together. Also, because the channel does not stop after reporting the End of Descriptor Interrupt, the End of Chain Interrupt or local bus errors may occur before the End of Descriptor Interrupt is acknowledged and cleared.

20.9 PACKING AND UNPACKING

Each channel contains a data hardware packing and unpacking unit to support unaligned data transfers between the source and destination busses. The packing unit optimizes data transfers to and from 32-bit memory. The channel reformats data words for the correct bus data path. When the channel must pack or unpack data, the data is held internally to the channel and does not need to be re-read.



SOURCE	DESTINATION
word load @ A0000200	byte store @ 40010303
word load @ A0000204	word store @40010304
word load @ A0000208	word store @40010308
word load @ A000020C	3-byte store @ 4001030C

Figure 20-15. Optimization of an Unaligned DMA

20.10 DMA CHANNEL PROGRAMMING EXAMPLES

Software is required for each of the following DMA channel functions:

- Channel initialization
- Start DMA transfer
- Suspend channel

Examples for each function is shown in the following sections as pseudocode.

20.10.1 Software DMA Controller Initialization

The DMA Controller has independent control of interrupts, enables, and control. Initialization consists of virtually no overhead as shown in Figure 20-16.

```
CCR0 = 0x0000 0000 ; Disable channel
Call setup_channel
```

Figure 20-16. Software Example for Channel Initialization

20.10.2 Software Start DMA Transfer

The DMA channel control register provides independent control per channel based on each time the DMA channel is configured. This provides the most flexibility to the application programmer.

20.10.3 Software Suspend Channel

The channel may need to be suspended for various reasons. The channel provides the ability to suspend the channel state without losing the current status. The channel resumes DMA operation without requiring the software to save the channel configuration. The example shown in Figure 20-17 describes the pseudocode for suspending channel 0.

```
CCR0 = 0x0000 0000; Suspend Channel 0

     Channel suspended.....

CCR0 = 0x0000 0001; Resume Channel 0
```

Figure 20-17. Software Example for Channel Suspend

21

I^{2}C BUS INTERFACE UNIT

CHAPTER 21
I^{2}C BUS INTERFACE UNIT

This chapter describes the I^{2}C (Inter-Integrated Circuit) bus interface unit of the i960® Rx I/O processor, including the operation modes and setup. Throughout this manual, this peripheral is referred to as the I^{2}C unit.

Figure 21-1 shows a block diagram of the I^{2}C unit and its interface to the 80960 local bus.

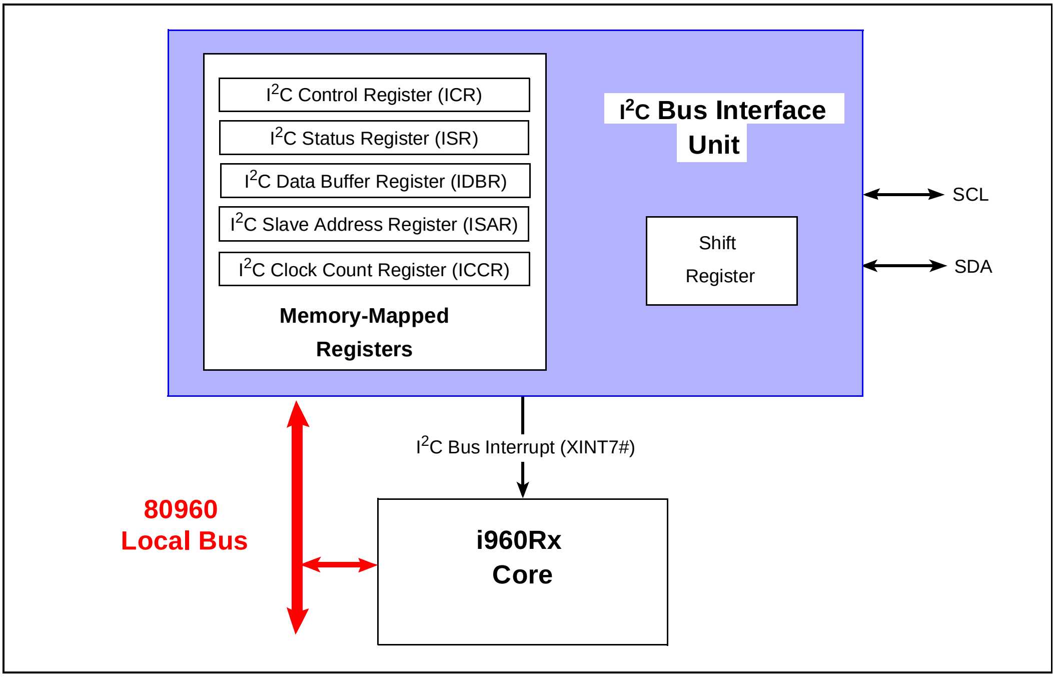

I2C Control Register (ICR)
I2C Status Register (ISR)
I2C Data Buffer Register (IDBR)
I2C Slave Address Register (ISAR)
I2C Clock Count Register (ICCR)
Memory-Mapped Registers
I2C Bus Interface Unit
Shift Register
SCL
SDA
I2C Bus Interrupt (XINT7#)
80960 Local Bus
i960Rx Core

Figure 21-1. I^{2}C Unit Block Diagram

21.1 OVERVIEW

The I^{2}C bus allows the i960 Rx I/O processor to interface to other I^{2}C peripherals and microcontrollers for system management functions. The serial bus requires hardware and software to create an economical system for relaying status and reliability information from the i960 Rx I/O processor subsystem to an external device.

Data transfers to and from the I^{2}C bus via a buffered interface. Control and status information are relayed through a set of 80960 memory-mapped registers. An interrupt mechanism notifies the i960 Rx I/O processor of I^{2}C activity. Refer to any of the following sources for details on I^{2}C bus operation:

- *I^{2}C Peripheral for Microcontrollers* – Philips Semiconductor
- *I^{2}C Bus and How to Use It (Including Specifications)* – Philips Semiconductor
- *I^{2}C Peripherals for Microcontrollers (Including Fast Mode)* – Signetics

The I^{2}C unit allows the i960 Rx I/O processor to serve as a master or slave device residing on the I^{2}C bus. The I^{2}C unit consists of:

- A Serial Data/Address (SDA) pin for input and output functions.
- A Serial Clock Line (SCL) pin for reference and control of the I^{2}C bus
- An 8-bit buffer for passing data to and from the i960 Rx I/O processor
- A shift register for parallel/serial data conversions
- A set of control and status registers
- A dedicated interrupt to inform the i960 Rx I/O processor of activity on the I^{2}C bus

21.2 THEORY OF OPERATION

The I^{2}C bus defines a complete serial protocol for passing information between agents on the I^{2}C bus using only a two pin interface. The interface consists of a Serial Data/Address (SDA) line and a Serial Clock Line (SCL). Each device on the I^{2}C bus is recognized by a unique 7-bit address and can operate as a transmitter or as a receiver. In addition to transmitter and receiver, the I^{2}C bus uses the concept of master and slave. Table 21-1 defines terms used in this chapter.

Table 21-1. I^{2}C Bus Definitions

I^{2}C Device	Definition
Transmitter	Sends data to the I^{2}C bus.
Receiver	Receives data from the I^{2}C bus.
Master	Initiates a transfer, generates the clock signal, and terminates the transactions.
Slave	The device addressed by a master.
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message.
Arbitration	Procedure to ensure that, when more than one master simultaneously tries to control the bus, only one is allowed. This procedure ensures that messages are not corrupted.

As an example of I^{2}C bus operation, consider the case of an i960 Rx I/O processor acting as a master on the bus (see Figure 21-2). The i960 Rx I/O processor, as a master, addresses an EEPROM as a slave to receive data. The i960 Rx I/O processor is a master-transmitter and the EEPROM is a slave-receiver. When the i960 Rx I/O processor reads data, the i960 Rx I/O processor is a master-receiver and the EEPROM is a slave-transmitter. In both cases, the master generates the clock, initiates the transaction and terminates it.

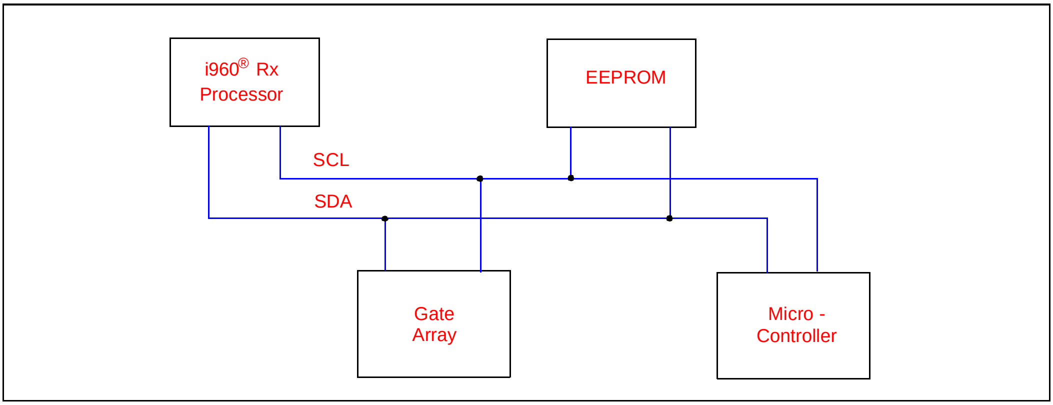


Figure 21-2. I^{2}C Bus Configuration Example

The I^{2}C bus allows for a multi-master system, which means more than one device can initiate data transfers at the same time. To support this feature, the I^{2}C bus arbitration relies on the wired-AND connection of all I^{2}C interfaces to the I^{2}C bus. Two masters can drive the bus simultaneously provided they are driving identical data. The first master to go high when other produces a low signal on the SDL line loses the arbitration. The SCL line consists of a synchronized combination of clocks generated by the masters using the wired-AND connection to the SCL line.

The I^{2}C bus serial operation uses an open-drain wired-AND bus structure, which allows multiple devices to drive the bus lines and to communicate status about events such as arbitration, wait states, error conditions and so on. For example, when a master drives the clock (SCL) line during a data transfer, it transfers a bit on every instance that the clock is high (see Figure 21-3). When the slave is unable to accept or drive data at the rate that the master is requesting, the slave can hold the clock line low between the high states to insert a wait interval. The master's clock can only be altered by a slow slave peripheral keeping the clock line low or by another master during arbitration. For more information on multi-master support, see Section 21.6, ARBITRATION (pg. 21-9).

The I^{2}C unit supports both fast mode operation at 400 Kbits/sec and standard mode at 100 Kbits/sec. Fast mode logic levels, formats, capacitive loading and protocols function the same in both modes. Refer to I^{2}C *Peripheral for Microcontrollers* by Philips Semiconductor for details. I^{2}C unit does not support I^{2}C 10-bit addressing or CBUS.

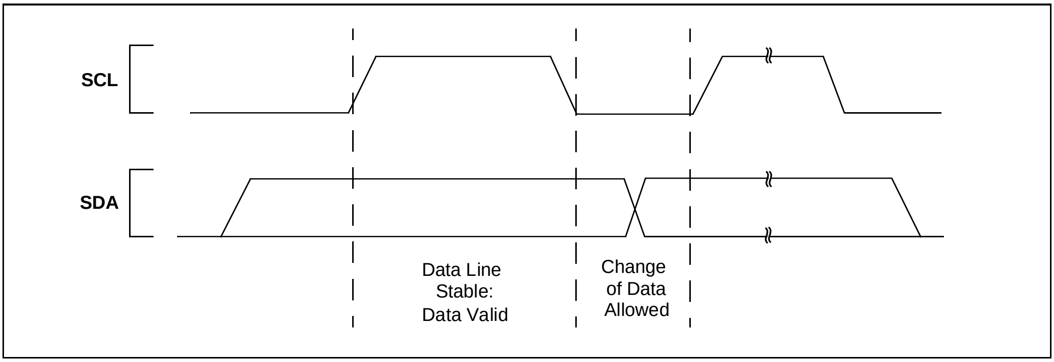


Figure 21-3. Bit Transfer on the I^{2}C Bus

21.3 START AND STOP BUS STATES

The i960 Rx I/O processor uses the START and STOP bits (bits 1:0) in the ICR (Table 21-6) to:

- Initiate a START condition on the I^{2}C bus.
- Enable data chaining (repeated START).
- Initiate a STOP condition on the I^{2}C bus.

Figure 21-4 shows the relationship between the SDA and SCL lines for a START and STOP condition.

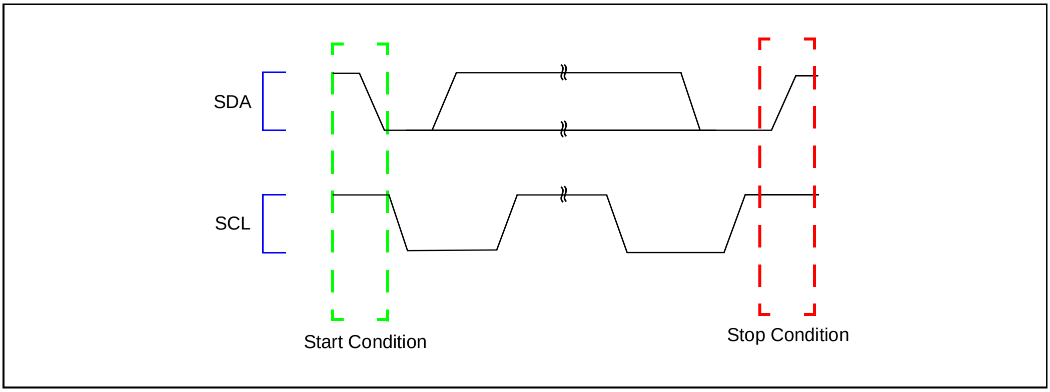


Figure 21-4. Start and Stop Conditions

21.3.1 START Condition

The START condition (bits 1:0 of the ICR set to 01_2) initiates a master transaction or repeated START. Software must load the target slave address and the R/W# bit in the IDBR (Table 21-9., I^{2}C Data Buffer Register – IDBR (pg. 21-26)) before setting the START ICR bit (see Figure 21-4). The START and the IDBR contents are transmitted on the I^{2}C bus when the ICR transfer byte bit is set. The I^{2}C bus stays in master-transmit mode when a write is requested or enters master-receive mode when a read is requested. For a repeated start (a change in read or write or a change in the target slave address), the IDBR contains the updated target slave address and the R/W# bit. This enables multiple transfers to different slaves without giving up the bus.

The START condition is not cleared by the I^{2}C unit when arbitration is lost. While initiating a START and the arbitration is lost, the I^{2}C unit may re-attempt the START when the bus becomes free - see section 21.6.2, SDA Arbitration (pg. 21-10). See section 21.6, ARBITRATION (pg. 21-9) for details on how the I^{2}C unit functions under those circumstances.

21.3.2 No START or STOP Condition

The START or STOP condition (bits 1:0 of the ICR set to 00_2) is used in master-transmit mode while the i960 Rx I/O processor is transmitting multiple data bytes (see Figure 21-4). When the IDBR buffer empty interrupt occurs, software clears the IDBR transmit empty bit to clear the interrupt. The software then initiates the repeated START as a master, by writing to the IDBR the target slave address and the R/W# bit. The software then sets the START bit in the ICR, clears the STOP bit, and disables the Arbitration Loss Interrupt bit in the ICR. To initiate the repeated START the software sets the transfer byte bit. The I^{2}C unit then waits for the IDBR transmit empty interrupt in the ISR.

The software writes a new byte to the IDBR and sets the Transfer Byte ICR bit, which initiates the new byte transmission. This continues until the software sets the START or STOP bit. The START and STOP bits in the ICR are not automatically cleared by the I^{2}C unit after the transmission of a START, STOP or repeated START.

After each byte transfer (including the Ack/Nack bit) the I^{2}C unit holds the SCL line low (inserting wait states) until the transfer byte bit in the ICR is set. This action notifies the I^{2}C unit to release the SCL line and allow the next information transfer to proceed.

21.3.3 STOP Condition

The STOP condition (bits 1:0 of the ICR set to 10_2) terminates a data transfer. In master-transmit mode, the software must write the last databyte to be transferred to the IDBR. The STOP bit and the transfer byte bit in the ICR must be set to initiate the last byte transfer (see Figure 21-4). In master-receive mode, to initiate the last transfer the i960 Rx I/O processor must set the Ack/Nack bit, the STOP bit, and the transfer byte bit in the ICR. Software must clear the STOP bit after it is transmitted.

21.4 SERIAL CLOCK LINE (SCL) MANAGEMENT

The i960 Rx I/O processor's I^{2}C clock (SCL) is programmed via the I^{2}C Clock Count Register (ICCR). The following subsections describe how the SCL works and is programmed.

21.4.1 SCL Clock Generation

The i960 Rx I/O processor's I^{2}C unit is required to generate the I^{2}C clock output when in master mode (either receive or transmit). SCL clock generation is accomplished through the use of the ICCR value, which is programmed at initialization. The ICCR value is used in the following equation to determine the SCL transition period:

SCL Transition Period =
ICCR Decimal Value * i960 Rx I/O Processor Local Bus Clock Period

The SCL transition period is the amount of time the clock spends in the high or low state. When wait states are inserted or synchronization with another master is necessary, the I^{2}C unit performs the necessary clock synchronization. The ICCR provides a simple method for determining I^{2}C clock frequencies. Table 21-2 details sample programming values for the ICCR.

Table 21-2. ICCR Programming Values

ICCR Value			i960 Rx I/O Processor Local Bus Frequency	SCL Transition Period	I^{2}C Clock Frequency = [1/(SCL Transition Per. * 2)]
00101010_2	2AH	42	33 MHz	1.27 µs	392.86 KHz
10100111_2	A7H	167	33 MHz	5.06 µs	98.88 KHz
00100000_2	20H	32	25 MHz	1.28 µs	390.63 KHz
01111101_2	7DH	125	25 MHz	5.00 µs	100.00 KHz

Programming a value less than 1EH results in undefined behavior.

21.5 DATA AND ADDRESSING MANAGEMENT

Data and slave addressing is managed via the I^{2}C Data Buffer Register (IDBR) and the I^{2}C Slave Address Register (ISAR). The IDBR (see Table 21-9., I^{2}C Data Buffer Register – IDBR (pg. 21-26)) contains data or a slave address and R/W# bit (Figure 21-5). The ISAR contains the i960 Rx I/O processor's programmable slave address. Data coming into the I^{2}C unit shift register is acknowledged and placed into the IDBR after a full byte is received. To transmit data, the processor writes to the IDBR, and the I^{2}C unit passes this onto the serial bus when the transfer byte bit in the ICR is set. See section 21.10.1, I^{2}C Control Register - ICR (pg. 21-19).

When the I^{2}C unit is in transmit mode (master or slave):

1. Software writes data to the IDBR over the 80960 local bus. This typically occurs to initiate a master transaction or to send the next data byte, after the IDBR transmit empty bit is sent.
2. The I^{2}C unit transmits the data from the IDBR when the transfer byte bit in the ICR is set.
3. When enabled, an IDBR transmit empty interrupt is signaled when a byte is transferred on the I^{2}C bus and the acknowledge cycle is complete.
4. When the I^{2}C bus is ready to transfer the next byte before the processor has written the IDBR (and a STOP condition is not in place), the I^{2}C unit inserts wait states until the processor writes a new value into the IDBR and sets the ICR transfer byte bit.

When the I^{2}C unit is in receive mode (master or slave):

1. The processor reads the IDBR data over the 80960 local bus after the IDBR receive full interrupt is signaled.
2. The I^{2}C unit transfers data from the shift register to the IDBR after the Ack cycle completes.
3. The I^{2}C unit inserts wait states until the IDBR is read. Refer to section 21.7, I^{2}C ACKNOWLEDGE (pg. 21-11) for acknowledge pulse information in receiver mode.
4. After the processor reads the IDBR, the I^{2}C unit sets the ICR's Ack/Nack Control bit and the transfer byte bit, allowing the next byte transfer to proceed.

21.5.1 Addressing a Slave Device

As a master device, the I^{2}C unit must compose and send the first byte of a transaction. This byte consists of the slave address for the intended device and a R/W# bit for transaction definition. The slave address and the R/W# bit are written to the IDBR (see Figure 21-5).

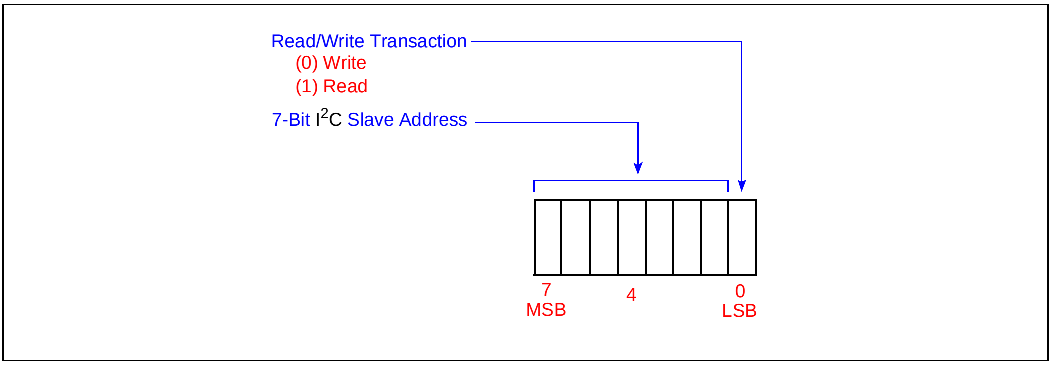


Figure 21-5. Data Format of First Byte in Master Transaction

The first byte transmission must be followed by an Ack pulse from the addressed slave. When the transaction is a write, the I^{2}C unit remains in master-transmit mode and the addressed slave device stays in slave-receive mode. When the transaction is a read, the I^{2}C unit transitions to master-receive mode immediately following the Ack and the addressed slave device transitions to slave-transmit mode. When a Nack is returned, the I^{2}C unit aborts the transaction by automatically sending a STOP and setting the ISR bus error bit.

When the I^{2}C unit is enabled and idle (no bus activity), it stays in slave-receive mode and monitors the I^{2}C bus for a START signal. Upon detecting a START pulse, the I^{2}C unit reads the first seven bits and compares them to those in the I^{2}C Slave Address Register (ISAR) and the general call address (00H). When the bits match those of the ISAR register, the I^{2}C unit reads the eighth bit (R/W# bit) and transmits an Ack pulse. The I^{2}C unit either remains in slave-receive mode (R/W# = 0) or transitions to slave-transmit mode (R/W# = 1). See section 21.8.3, General Call Address (pg. 21-16) for actions when a general call address is detected.

21.6 ARBITRATION

Arbitration on the I^{2}C bus is required due to the multi-master capabilities of the I^{2}C bus. Arbitration is used when two or more masters simultaneously generate a START condition within the minimum I^{2}C hold time of the START condition. The following sections describe the arbitration on the SCL and SDA lines.

21.6.1 SCL Arbitration

Each master on the I^{2}C bus generates its own clock on the SCL line for data transfers. With masters generating their own clocks, clocks with different frequencies may be connected to the SCL line. Since data is valid when the clock is in the high period, a defined clock synchronization procedure is needed during bit-by-bit arbitration.

Clock synchronization is accomplished by using the wired-AND connection of the I^{2}C interfaces to the SCL line. When a master's clock transitions from high to low, this causes the master to hold down the SCL line for its associated period (see Figure 21-6). The low to high transition of the clock may not change when another master has not completed its period. Therefore, the master with the longest low period holds down the SCL line. Masters with shorter periods are held in a high wait-state during this time. Once the master with the longest period completes, the SCL line transitions to the high state, masters with the shorter periods can continue the data cycle.

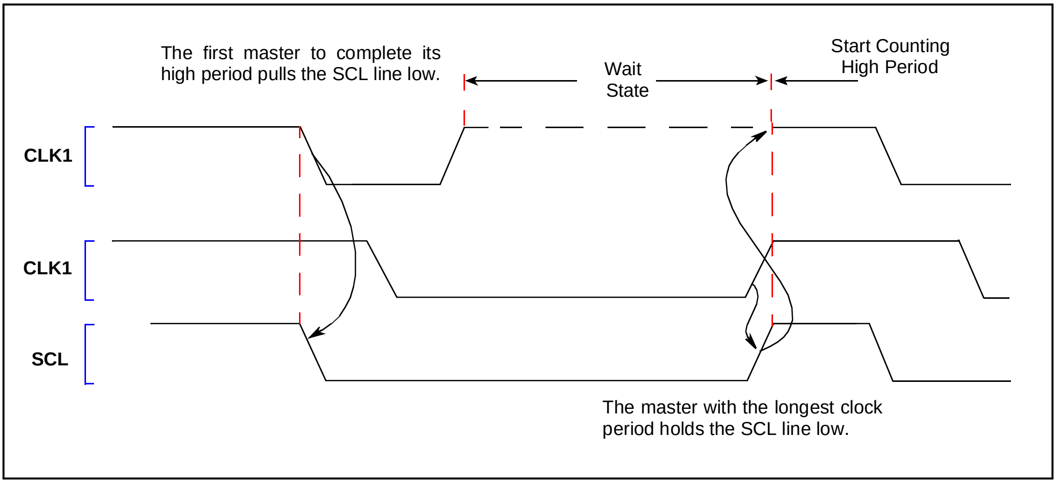

The first master to complete its high period pulls the SCL line low.
Wait State
Start Counting High Period
CLK1
CLK1
SCL
The master with the longest clock period holds the SCL line low.

Figure 21-6. Clock Synchronization During the Arbitration Procedure

21.6.2 SDA Arbitration

Arbitration on the SDA line can continue for a long period starting with the address and R/W# bits and continuing with the data bits. Figure 21-7 shows the arbitration procedure for two masters (more than two may be involved depending on how many masters are connected to the bus). When the address bit and the R/W# are the same, the arbitration moves to the data. Due to the wired-AND nature of the I^{2}C bus, no data is lost when both (or all) masters are outputting the same bus states. When the address, R/W# bit, or data is different, the master that output the first high data bit loses arbitration and shuts its data drivers off. When the I^{2}C unit loses arbitration, it shuts off the SDA or SCL drivers for the remainder of the byte transfer, sets the arbitration loss detected ISR bit, then returns to idle (Slave-Receive) mode.

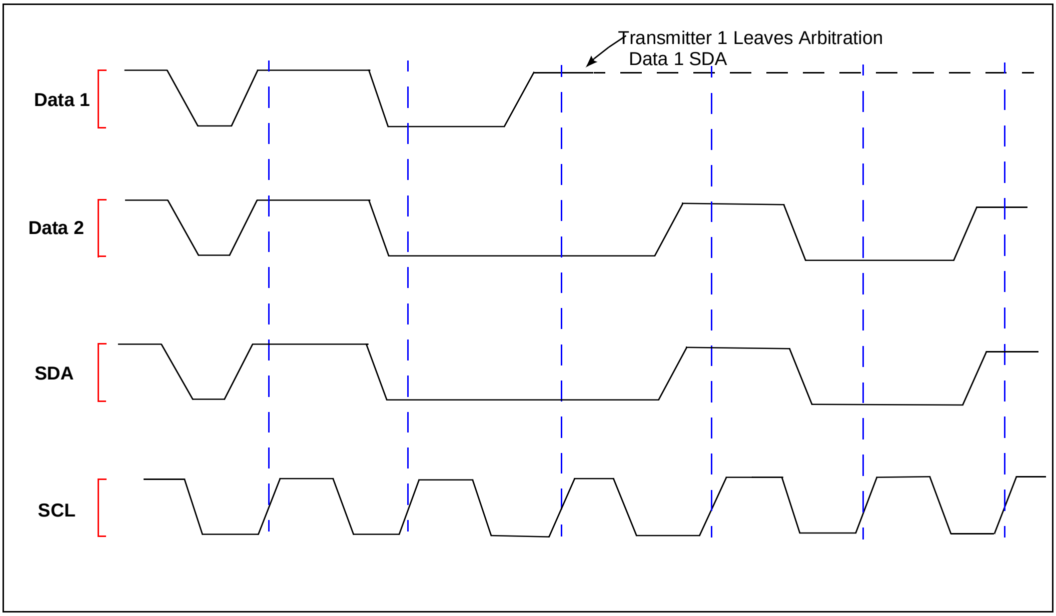


Figure 21-7. Arbitration Procedure of Two Masters

When the I^{2}C unit loses arbitration during transmission of the seven address bits and the i960 Rx I/O processor is not being addressed as a slave device, the I^{2}C unit resends the address when the I^{2}C bus becomes free. This is possible because the IDBR and ICR registers are not overwritten when arbitration is lost.

When the arbitration loss is to due to another bus master addressing the i960 Rx I/O processor as a slave device, the I^{2}C unit switches to slave-receive mode and the original data in the I^{2}C data buffer register is overwritten. Software is responsible for clearing the start and reinitiating the master transaction at a later time.

NOTE: Software must not allow the I^{2}C unit to write to its own slave address. This can cause the I^{2}C bus to enter an indeterminate state.

Boundary conditions exist for arbitration when an arbitration process is in progress and a repeated START or STOP condition is transmitted on the I^{2}C bus. To prevent errors, the I^{2}C unit, acting as a master, provides for the following sequences:

- No arbitration takes place between a repeated START condition and a data bit
- No arbitration takes place between a data bit and a STOP condition
- No arbitration takes place between a repeated START condition and a STOP condition

These situations arise only when different masters write the same data to the same target slave simultaneously and arbitration is not resolved after the first data byte transfer.

NOTE: Typically software protocol is responsible for ensuring arbitration is lost soon after the transaction begins. For example, the protocol might insist that all masters transmit their I^{2}C address as the first data byte of any transaction ensuring arbitration is ended. A restart is then sent to begin a valid data transfer (the slave can then discard the master's address).

21.7 I^{2}C ACKNOWLEDGE

Every I^{2}C byte transfer must be accompanied by an acknowledge pulse, which is always generated by the receiver (master or slave). The transmitter must release the SDA line for the receiver to transmit the acknowledge pulse (see Figure 21-8).

In master-transmit mode, when the target slave receiver device cannot generate the acknowledge pulse, the SDA line remains high. This lack of acknowledge (Nack) causes the I^{2}C unit to set the bus error detected bit in the ISR and generate the associated interrupt (when enabled). The I^{2}C unit aborts the transaction by generating a STOP automatically.

In master-receive mode, the I^{2}C unit signals the slave-transmitter to stop sending data by using the negative acknowledge (Nack). The Ack/Nack bit value driven by the I^{2}C bus is controlled by the Ack/Nack control bit in the ICR. The bus error detected bit in the ISR is not set for a master-receive mode Nack (as required by the I^{2}C bus protocol). When the transmit bit is set in the ICR, the I^{2}C unit automatically transmits the Ack pulse, based on the Ack/Nack control bit, after receiving each byte from the serial bus. Before receiving the last byte, software must set the Ack/Nack Control bit to Nack. Nack is then sent after the next byte is received to indicate the last byte.

In slave mode, the I^{2}C unit automatically acknowledges its own slave address, independent of the Ack/Nack control bit setting in the ICR. As a slave-receiver, an Ack response is automatically given to a data byte, independent of the Ack/Nack control bit setting in the ICR. The I^{2}C unit sends the Ack value after receiving the eighth data bit of the byte.

In slave-transmit mode, receiving a Nack from the master indicates the last byte is transferred. The master then sends either a STOP or repeated START. The ISR's unit busy bit (2) remains set until a STOP or repeated START is received.

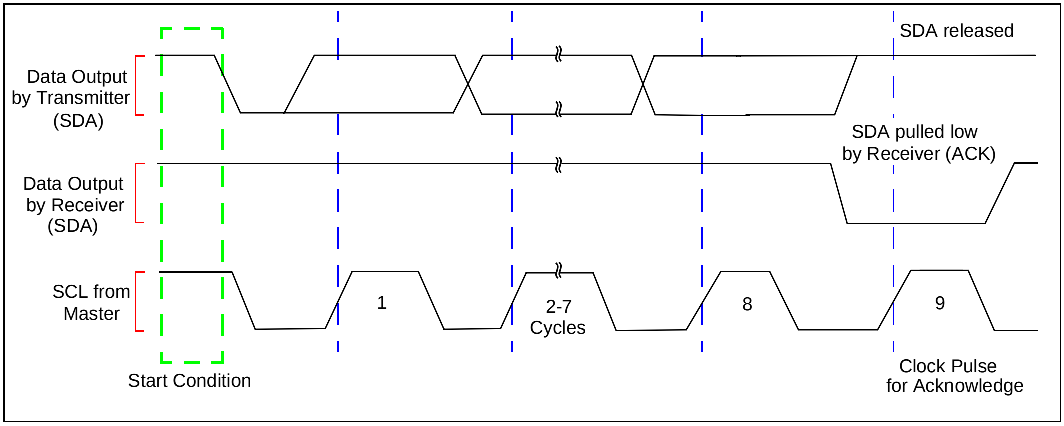

SDA released
Data Output by Transmitter (SDA)
SDA pulled low by Receiver (ACK)
Data Output by Receiver (SDA)
SCL from Master
1
2-7 Cycles
8
9
Start Condition
Clock Pulse for Acknowledge

Figure 21-8. Acknowledge on the I^{2}C Bus

21.8 I^2C MASTER AND SLAVE OPERATIONS

The I^2C unit can be in different modes of operation to accomplish a transfer. Table 21-3 summarizes the different modes.

Table 21-3. Operation Modes

Mode	Definition
Master - Transmit	• Used for a write operation on the bus. • I^2C unit sends the data. • I^2C unit is responsible for clocking. • Slave device must be in slave-receive mode.
Master - Receive	• Used for a read operation on the bus. • I^2C unit receives the data. • I^2C unit is responsible for clocking. • Slave device must be in slave-transmit mode.
Slave - Transmit	• Used for a write operation on the bus. • I^2C unit sends the data. • Master device must be in master-receive mode.
Slave - Receive (default)	• Used for a read operation on the bus. • I^2C unit receives the data. • Master device must be in master-transmit mode.

The I^2C unit enable bit (6) in the ICR must be set and the reset bit (14) cleared before the I^2C unit may act as a master or slave device. When the I^2C unit is in an idle mode (neither receiving or transmitting serial data), the unit defaults to slave-receive mode. This allows the interface to monitor the bus and receive any slave addresses that might be intended for the i960 Rx I/O processor.

The I^2C unit transfers in 1-byte increments. A data transfer on the I^2C bus always follows the sequence:

1) START

2) 7-bit slave address

3) R/W# bit

4) Acknowledge

5) 8 bits of data

6) Acknowledge or No Acknowledge (NACK)

7) Repeat of step 5 and 6 for required number of bytes

8) STOP condition or a repeated START (for repeated START repeat steps 1-8)

21.8.1 Master Operations

When software initiates a read or write on the I^{2}C bus, the I^{2}C unit transitions from the default slave-receive mode to master-transmit mode. The start pulse is sent followed by the 7-bit slave address and the R/W bit. After the master receives an acknowledge, the I^{2}C unit has the option of being one of two master modes:

- Master-Transmit — The i960 Rx I/O processor writes data
- Master-Receive — The i960 Rx I/O processor reads data

The i960 Rx I/O processor sets up a master transaction by writing to the slave address and the R/W# bit to the IDBR. To initiate this transaction, the START bit and the TRANSMIT bit are set. Data is read and written from the I^{2}C unit through the memory-mapped registers. When the i960 Rx I/O processor needs to read data, the I^{2}C unit transitions from slave-receive mode to master-transmit mode to transmit the start address and immediately following the ACK pulse transitions to master-receive mode to wait for the reception of the read data from the slave device (see Figure 21-9). It is also possible to have multiple transactions during an I^{2}C operation such as transitioning from master-receive to master-transmit through a repeated start or Data Chaining (see Figure 21-10). Figure 21-11 shows the wave forms of SDA and SCL for a complete data transfer.

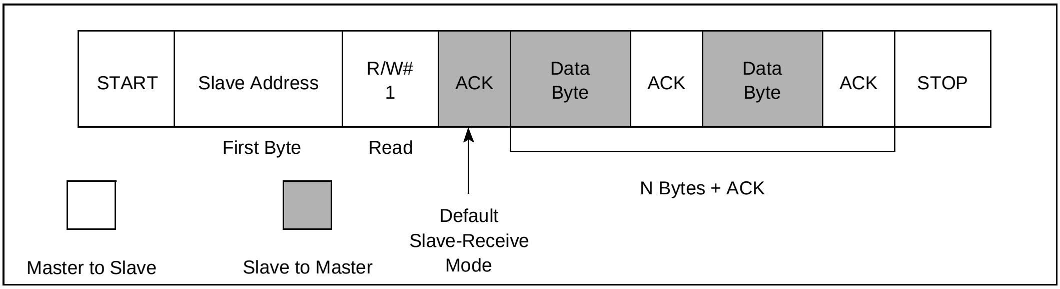


Figure 21-9. Master-Receiver Read from Slave-Transmitter

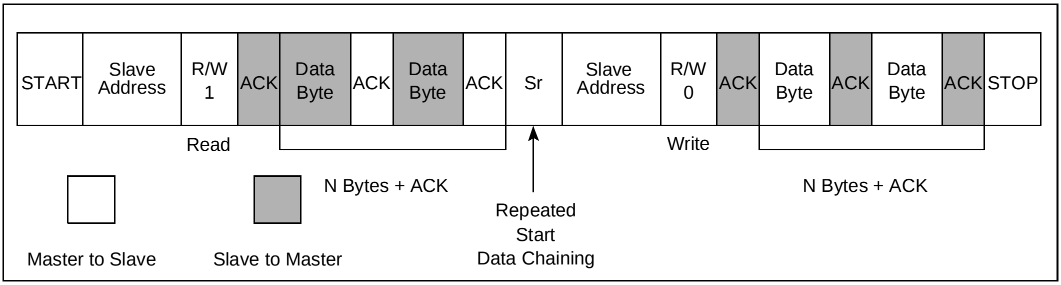


Figure 21-10. Master-Receiver Read from Slave-Transmitter / Repeated Start / Master-Transmitter Write to Slave-Receiver

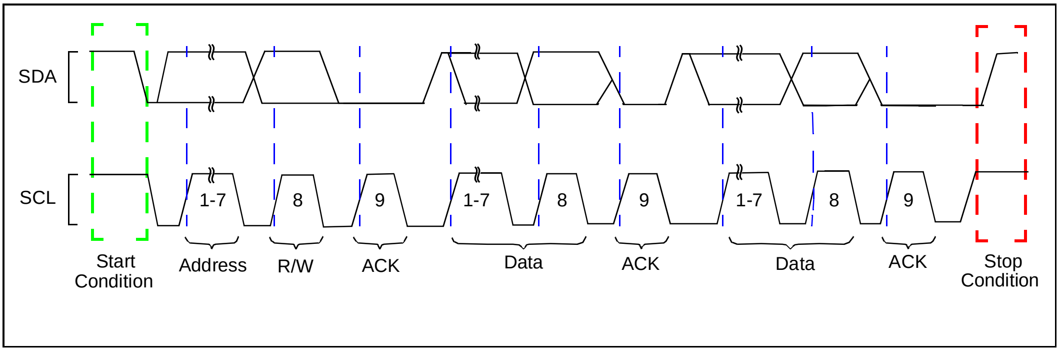


Figure 21-11. A Complete Data Transfer

The i960 Rx I/O processor initiates a master transaction by writing to the ICR register. Table 21-4., General Call Address Second Byte Definitions (pg. 21-17) describes the I^2C unit responsibilities as a master device.

21.8.2 Slave Operations

Figure 21-12 through Figure 21-14 are examples of I^2C transactions. These show the relationships between master and slave devices.

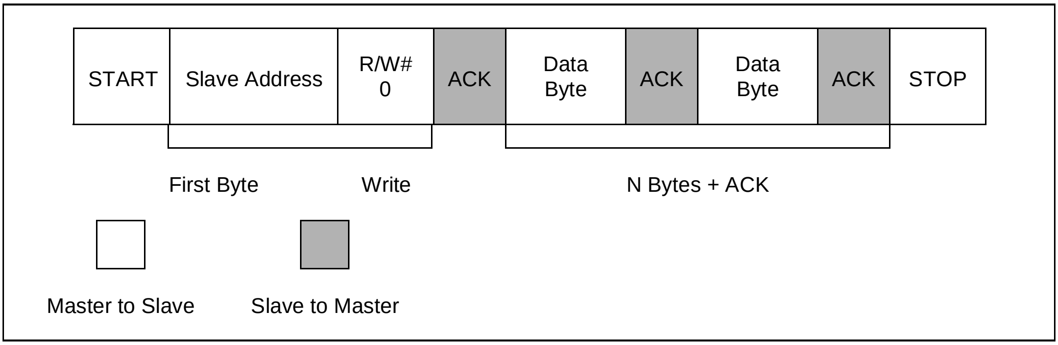


Figure 21-12. Master-Transmitter Write to Slave-Receiver

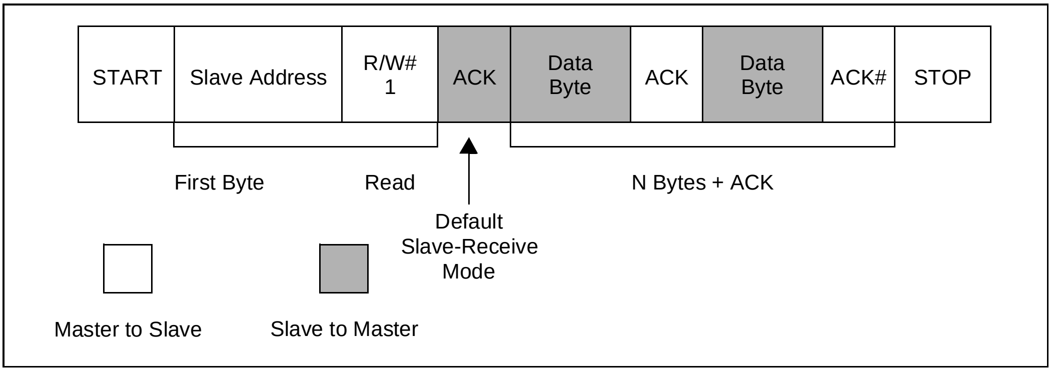


Figure 21-13. Master-Receiver Read to Slave-Transmitter

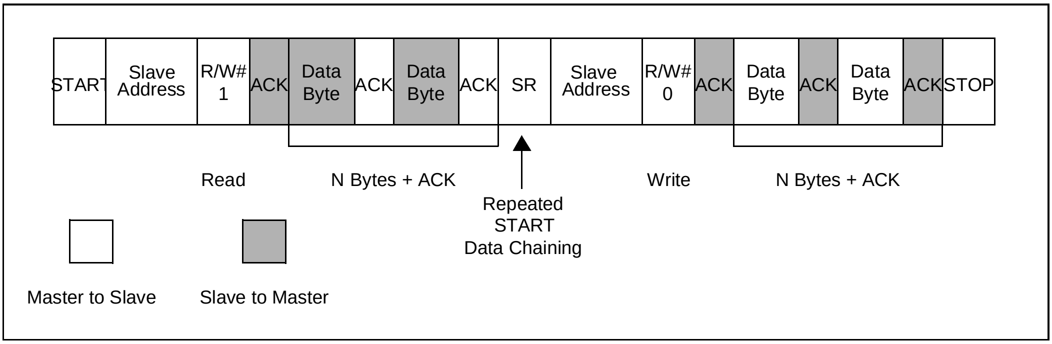


Figure 21-14. Master-Receiver Read to Slave-Transmitter, Repeated START, Master-Transmitter Write to Slave-Receiver

21.8.3 General Call Address

The I^{2}C unit supports both sending and receiving general call address transfers on the I^{2}C bus. When sending a general call message from the I^{2}C unit, software must set the general call disable bit in the ICR to keep the I^{2}C unit from responding as a slave. Failure to do this causes the I^{2}C Bus to enter an indeterminate state.

A general call address is defined as a transaction with a slave address of 00H. When a device requires the data from a general call address, it Acks the transaction and stays in slave-receiver mode. Otherwise, the device can ignore the general call address. The second and following bytes of a general call transaction are acknowledged by every device using it on the bus. Any device not using these bytes must not Ack. The meaning of a general call address is defined in the second byte sent by the master-transmitter. Figure 21-15 shows a general call address transaction. The least significant bit of the second byte defines the transaction. Table 21-4., General Call Address Second Byte Definitions (pg. 21-17) shows the valid values and definitions when B = 0.

When the i960 Rx I/O processor is acting as a slave, and the I^{2}C unit receives a general call address and the ICR general call disable bit is clear the I^{2}C unit:

- Sets the ISR general call address detected bit.
- Sets the ISR slave address detected bit.
- Signals an interrupt (when enabled) to the i960 Rx I/O processor.

When the I^{2}C unit receives a general call address and the ICR general call disable bit is set, the I^{2}C unit will ignore the general call address.

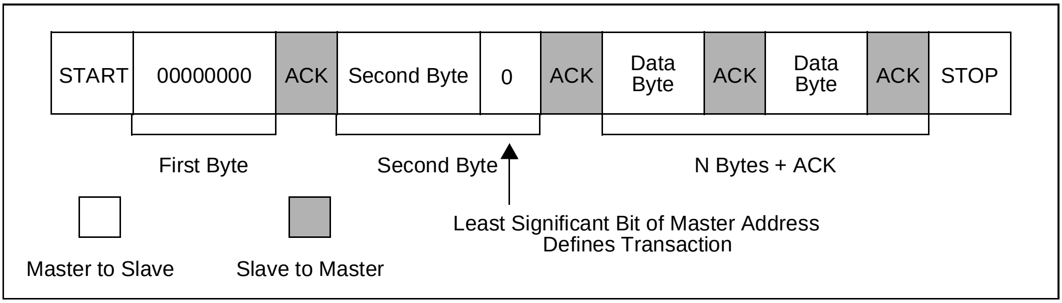


Figure 21-15. General Call Address

Table 21-4. General Call Address Second Byte Definitions

Least Significant Bit of Second Byte (B)	Second Byte Value	Definition
0	06H	2-byte transaction where the second byte tells the slave to reset and then store this value in the programmable part of their slave address.
0	04H	2-byte transaction where the second byte tells the slave to store this value in the programmable part of their slave address. No reset.

21.9 THE I^{2}C BUS UNIT AND RESET

The I^{2}C unit is reset by the local bus reset signal that is active when P_RST# is asserted or when reset local bus bit in EBCR is set. Software is responsible for ensuring the I^{2}C unit is not busy (unit busy is clear) before asserting reset. Software is also responsible for ensuring the I^{2}C bus is idle when the unit is enabled after reset. When directed to reset, the I^{2}C unit must return to its default reset condition with the exception of the ISAR. ISAR is not affected by a reset.

When the unit reset bit in the ICR is set, only the i960 Rx I/O processor I^{2}C unit resets, the associated I^{2}C MMRs remain intact. When resetting the I^{2}C unit with the ICR's unit reset, use the following guidelines:

1. In the ICR register, set the reset bit and clear the remainder of the register
2. Clear the ISR register
3. Clear reset in the ICR

21.10 I^{2}C REGISTERS

Table 21-5 identifies all I^{2}C unit registers. Subsections identify all registers and define bit settings.

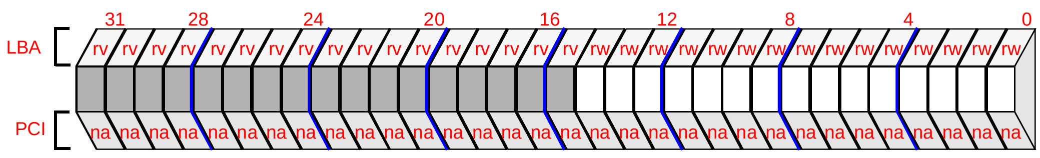
Table 21-5. I^{2}C Register Summary

Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
21.10.1	I^{2}C Control Register - ICR	21-19	32	0000 1680H	NA
21.10.2	I^{2}C Status Register- ISR	21-22	32	0000 1684H	NA
21.10.3	I^{2}C Slave Address Register – ISAR	21-25	32	0000 1688H	NA
21.10.4	I^{2}C Data Buffer Register – IDBR	21-26	32	0000 168CH	NA
21.10.5	I^{2}C Clock Count Register – ICCR	21-27	32	0000 1690H	NA

21.10.1 I^2C Control Register - ICR

The i960 Rx I/O processor uses the bits in the I^2C Control Register (ICR) to control the I^2C unit.

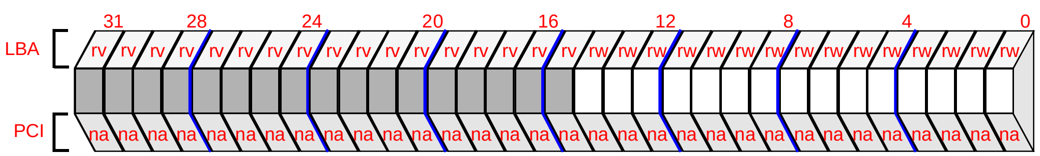
Table 21-6. I^2C Control Register – ICR (Sheet 1 of 3)


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1680H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 local bus address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
			RO = Read Only
			RW = Read/Write

Bit	Default	Description
31:15	0000 0H	Reserved
14	0_2	**Unit Reset**: 1 = Reset the i960 Rx I/O processor I^2C unit only. 0 = No reset.
13	0_2	**Slave Address Detected Interrupt Enable**: 1 = Enables the I^2C unit to signal an interrupt to the i960 Rx I/O processor upon detecting a slave address match or a general call address. 0 = Disable interrupt.
12	0_2	**Arbitration Loss Detected Interrupt Enable**: 1 = Enables the I^2C unit to signal an interrupt upon losing arbitration while in master mode. 0 = Disable interrupt.
11	0_2	**Slave STOP Detected Interrupt Enable**: 1 = Enables the I^2C unit to signal an interrupt when it detects a STOP condition while in slave mode. 0 = Disable interrupt.
10	0_2	**Bus Error Interrupt Enable**: 1 = Enables the I^2C unit to signal an interrupt for the following I^2C bus errors: • As a master transmitter, no Ack was detected after a byte was sent. • As a slave receiver, the I^2C unit generated a Nack pulse. **Note:** Software is must guarantee that misplaced START and STOP conditions do not occur. See section 13.6. 0 = Disable interrupt.
09	0_2	**IDBR Receive Full Interrupt Enable**: 1 = Enables the I^2C unit to signal an interrupt to the i960 Rx I/O processor when the IDBR has received a data byte from the I^2C bus. 0 = Disable interrupt.

Table 21-6. I^{2}C Control Register – ICR (Sheet 2 of 3)

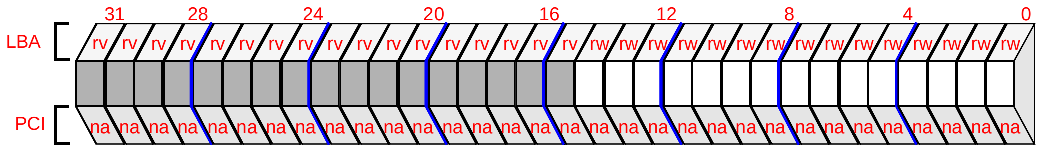


LBA: 1680H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
08	0_2	**IDBR Transmit Empty Interrupt Enable**: 1 = Enables the I^{2}C unit to signal an interrupt to the i960 Rx I/O processor after transmitting a byte onto the I^{2}C bus. 0 = Disable interrupt.
07	0_2	**General Call Interrupt Disable**: 1 = Disables I^{2}C unit response to general call messages as a slave. 0 = Enables the I^{2}C unit to respond to general call messages. This bit must be set when sending a master mode general call message from the I^{2}C unit.
06	0_2	**I^{2}C Unit Enable**: 1 = Enables the I^{2}C unit (defaults to slave-receive mode). 0 = Disables the unit and does not master any transactions or respond to any slave transactions. Software must guarantee the I^{2}C bus is idle before setting this bit.
05	0_2	**SCL Enable**: 1 = Enables the I^{2}C clock output for master mode operation. The ICCR (see section 21.10.5) must be programmed with a valid value before setting this bit. 0 = Disables the I^{2}C unit from driving the SCL line.
04	0_2	**Master Abort**: used by the I^{2}C unit when in master mode to generate a STOP without transmitting another data byte. 1 = The I^{2}C unit sends STOP without data transmission. 0 = The I^{2}C unit transmits STOP using the STOP ICR bit only. When in Master transmit mode, after transmitting a data byte, the ICR's transfer byte bit is clear and IDBR transmit empty bit is set. When no more data bytes need to be sent, setting master abort bit sends the STOP. The transfer byte bit (03) must remain clear. In master-receive mode, when a Nack is sent without a STOP (STOP ICR bit was not set) and the i960 Rx I/O processor does not send a repeated START, setting this bit sends the STOP. Once again, the transfer byte bit (03) must remain clear.

Table 21-6. I^{2}C Control Register – ICR (Sheet 3 of 3)


LBA
PCI
31
28
24
20
16
12
8
4
0
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1680H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

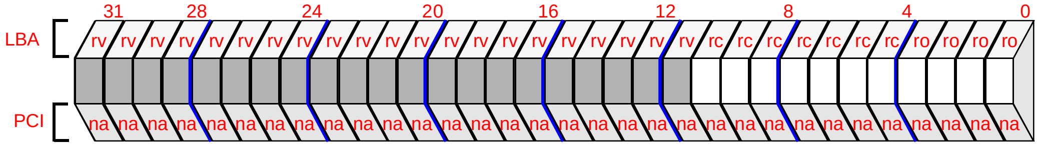
Bit	Default	Description
03	0_2	**Transfer Byte**: used to send/receive a byte on the I^{2}C bus. 1 = send/receive a byte. 0 = cleared by I^{2}C unit when the byte is sent/received. The i960 Rx I/O processor can monitor this bit to determine when the byte transfer has completed. In master or slave mode, after each byte transfer including Ack/Nack bit, the I^{2}C unit holds the SCL line low (inserting wait states) until the transfer byte bit is set.
02	0_2	**Ack/Nack Control**: defines the type of Ack pulse sent by the I^{2}C unit when in master or slave receive mode. 1 = The I^{2}C unit sends a negative Ack (Nack) after receiving a data byte. 0 = The I^{2}C unit sends an Ack pulse after receiving a data byte. The I^{2}C unit automatically sends an Ack pulse when responding to its slave address, independent of the Ack/Nack control bit setting.
01	0_2	**STOP**: used to initiate a STOP condition after transferring the next data byte on the I^{2}C bus when in master mode. In master-receive mode, the Ack/Nack control bit must be set in conjunction with this bit. See section 21.3. for more details on the STOP state. 1 = Send a STOP 0 = Do not send a STOP
00	0_2	**START**: used to initiate a START condition to the I^{2}C unit when in master mode. See section 21.3. for more details on the START state. 1 = Send a START 0 = Do not send a START

21.10.2 I^{2}C Status Register- ISR

I^{2}C interrupts are signaled through XINT7# and the XINT7 Interrupt Status Register (X7ISR), which shows the pending XINT7 interrupts (see CHAPTER 8, INTERRUPTS). XINT7# is set by the I^{2}C Interrupt Status Register (ISR). Software uses the ISR bits to check the status of the I^{2}C unit and bus. ISR bits (bits 5-9) are updated after the Ack/Nack bit has completed on the I^{2}C bus. The ISR is also used to clear interrupts signaled from the I^{2}C unit. They are:

- IDBR receive full
- IDBR transmit empty
- slave address detected
- bus error detected
- STOP condition detect
- arbitration lost

Table 21-7. I^{2}C Status Register – ISR (Sheet 1 of 3)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc rc rc ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1684H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:11	0000 00H	Reserved
10	0_2	**Bus Error Detected**: 1 = The I^{2}C unit sets this bit when it detects one of the following error conditions: • As a master transmitter, no Ack was detected on the interface after a byte was sent. • As a slave receiver, the I^{2}C unit generates a Nack pulse. **Note:** When an error occurs, I^{2}C bus transactions continue. Software must guarantee that misplaced START and STOP conditions do not occur. See section 21.6, ARBITRATION (pg. 21-9). 0 = no error detected.
09	0_2	**Slave Address Detected**: 1 = I^{2}C unit detected a 7-bit address that matches the general call address or ISAR. An interrupt is signaled when enabled in the ICR. 0 = No slave address detected.

Table 21-7. I²C Status Register – ISR (Sheet 2 of 3)

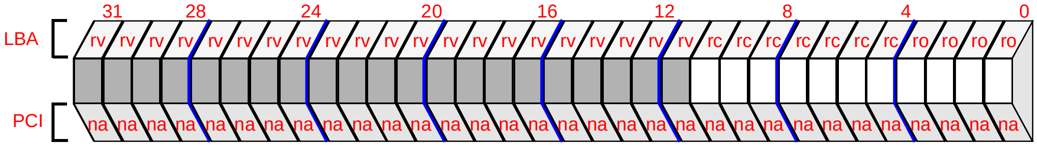
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LBA	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rv	rc	rc	rc	rc	rc	rc	rc	ro	ro	ro	ro
PCI	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

LBA: 1684H
PCI: NA

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 local bus address
PCI = PCI Configuration Address Offset

Bit	Default	Description
08	0_2	**General Call Address Detected**: 1 = I²C unit received a general call address. An interrupt is signaled when enabled in the ICR. 0 = No general call address received.
07	0_2	**IDBR Receive Full**: 1 = The IDBR register received a new data byte from the I²C bus. An interrupt is signaled when enabled in the ICR. 0 = The IDBR has not received a new data byte or the I²C unit is idle.
06	0_2	**IDBR Transmit Empty:** 1 = The I²C unit has finished transmitting a data byte on the I²C bus. An interrupt is signaled when enabled in the ICR. 0 = The data byte is still being transmitted.
05	0_2	**Arbitration Loss Detected**: used during multi-master operation. 1 = Set when the I²C unit loses arbitration. 0= Cleared when arbitration is won or never took place.
04	0_2	**Slave STOP Detected**: 1 = Set when the I²C unit detects a STOP while in slave-receive or slave-transmit mode. 0 = No STOP detected.
03	0_2	**I²C Bus Busy**: 1 = Set when the I²C bus is busy but the i960 Rx I/O processor's I²C unit is not involved in the transaction. 0 = I²C bus is idle or the I²C unit is using the bus (i.e., unit busy).
02	0_2	**Unit Busy:** 1 = Set when the i960 Rx I/O processor's I²C unit is busy. This is defined as the time between the first START and STOP. 0 = I²C unit not busy.

Table 21-7. I^{2}C Status Register – ISR (Sheet 3 of 3)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc rc rc ro ro ro ro
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1684H

PCI: NA

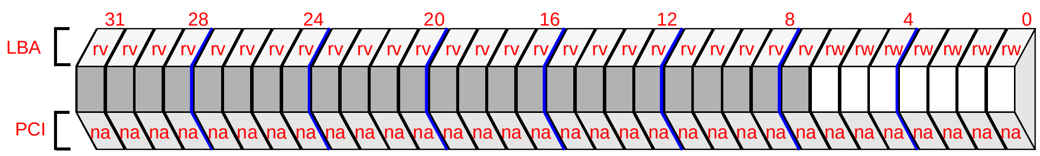
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
01	0_2	**Ack/Nack Status**: 1 = The I^{2}C unit received a Nack. 0 = The I^{2}C unit received an Ack on the bus. This bit is used in slave transmit mode to determine when the byte transferred is the last one. This bit is updated after each byte and Ack/Nack information is received.
00	0_2	**R/W Mode:** 1 = The I^{2}C unit is in receive mode. 0 = The I^{2}C unit is in transmit mode. This is the R/W# received after a slave address match. It is automatically cleared by hardware after a stop state.

21.10.3 I^2C Slave Address Register – ISAR

The I^2C Slave Address Register (see Table 21-8) defines the I^2C unit's 7-bit slave address to which the i960 Rx I/O processor responds when in slave-receive mode. This register is written by the i960 Rx I/O processor before enabling I^2C operations. The register is fully programmable (no address is assigned to the I^2C unit) so it can be set to a value other than those of hard-wired I^2C slave peripherals that might exist in the system. The ISAR is not affected by the i960 Rx I/O processor being reset. The ISAR register default value is 00H.

Table 21-8. I^2C Slave Address Register – ISAR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

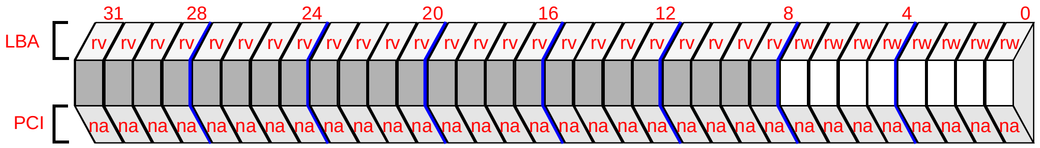
LBA: 1688H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 local bus address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
		PCI = PCI Configuration Address Offset
			RO = Read Only
			RW = Read/Write

Bit	Default	Description
31:07	0000 000H	Reserved
06:00	0H	**I^2C Slave Address**: The 7-bit address to which the I^2C unit responds when in slave-receive mode.

21.10.4 I^{2}C Data Buffer Register – IDBR

The IDBR (see Table 21-9) receives and sends data and transmits the slave address for the intended slave. The IDBR register defaults to 00H after reset.

Table 21-9. I^{2}C Data Buffer Register – IDBR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv
rw rw rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 168CH
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 local bus address PCI = PCI Configuration Address Offset

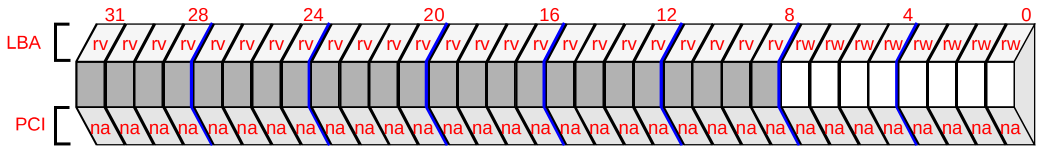
Bit	Default	Description
31:08	0000 00H	Reserved
07:00	00H	**I^{2}C Data Buffer**: Buffer for I^{2}C bus send/receive data.

21.10.5 I^2C Clock Count Register – ICCR

The I^2C Clock Count Register (ICCR) defines the multiplier used to generate the I^2C SCL clock. This register is used with an internal 8-bit down counter. When the SCL enable bit in the ICR is set, this counter decrements from the programmed ICCR value to zero, then resets to the programmed ICCR value and begins to decrement again. This continues until the SCL enable bit in the ICR is cleared. Each time the counter reaches zero, the SCL line transitions from low to high or vice versa, depending on the current state. This creates the I^2C clock output used during I^2C master operations.

Changing this register while the SCL enable bit is set results in undefined behavior.

Table 21-10. I^2C Clock Count Register – ICCR


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1690H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write

	LBA = 80960 local bus address	PCI = PCI Configuration Address Offset	

Bit	Default	Description
31:08	0000 00H	Reserved
07:00	00H	**I^2C Clock Count**: 8 bit count value used to generate an I^2C clock from the i960 Rx I/O processor local bus clock.

22

I/O APIC BUS INTERFACE UNIT

CHAPTER 22
I/O APIC BUS INTERFACE UNIT

This chapter details the operation modes, setup and implementation of the i960® Rx I/O processor’s I/O Advanced Peripheral Interrupt Controller (APIC) Bus Interface Unit.

22.1 Overview

The APIC interrupt architecture is specified as the interrupt architecture for all Intel Multiprocessor Specification* (MPS) compatible systems. The APIC Bus Interface Unit provides a mechanism for communication between the local bus and the 3-wire APIC bus. It provides two basic functions:

- It gives the i960 core processor the ability to send an interrupt message out onto the APIC bus and optionally be interrupted when the message has been sent. The i960 core processor can then read the resulting status of the message transmission to check for errors.
- It can also receive EOI messages from the APIC bus and optionally interrupt the i960 core processor to inform it that an EOI vector is available.

22.2 Theory of Operation

Structurally, the APIC partitions into two portions (Figure 22-1):

- One residing in the I/O Subsystem.
- One residing in the Host CPU.

The portion that resides in the I/O subsystem is known as I/O APIC Unit and the portion that resides in the CPU is known as Local APIC Unit. The local APIC and the I/O APIC communicate over a dedicated APIC bus.

The I/O APIC Unit provides the interrupt input pins on which I/O devices send interrupts into the system in the form of an edge or a level detect. The I/O APIC also contains a Redirection Table with an entry for each interrupt input pin.

* Intel *MultiProcessor Specification*, Order Number 242016

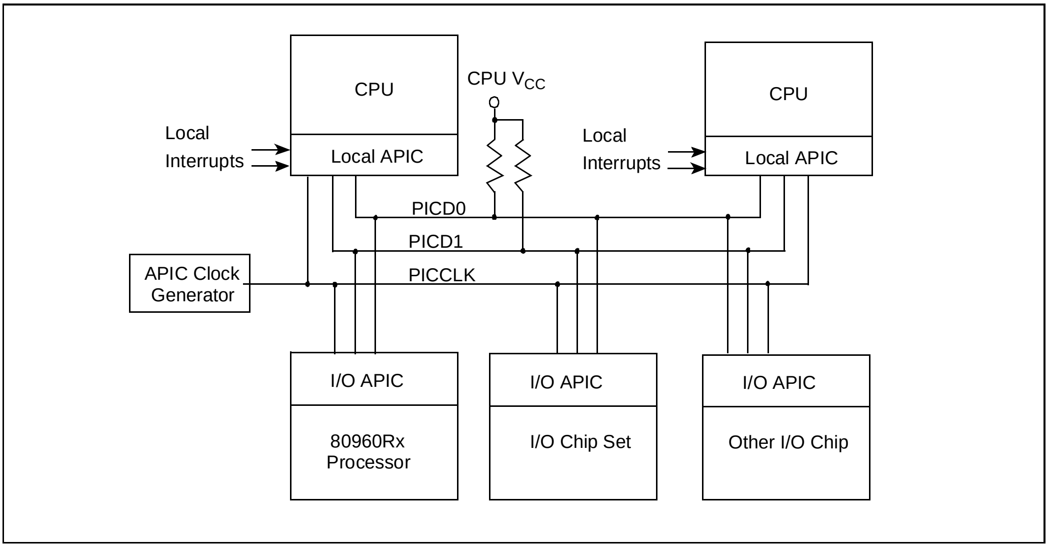


Figure 22-1. APIC System Interface

22.3 Physical Characteristics of an APIC

The APIC bus is a 3-wire synchronous bus connecting all APICs (I/O and Local units). Two of these wires (PICD0, PICD1) are used for data transmission, and one wire is a clock (PICCLK). For bus arbitration, the APIC uses only one of the data wires (Figure 22-1).

The bus is logically a wire-OR and electrically an open-drain connection providing arbitration for both bus use and for lowest priority interrupt delivery. Being open-drain, the bus is run at a nominal speed such that design-specific termination tuning is not required. Furthermore, each APIC receiving a message or participating in an arbitration must be given enough time in a single bus cycle to latch the bus and perform some simple logic operations on the latched information in order to determine whether the next drive cycle must be inhibited. The maximum APIC bus speed is 16 MHz.

All values mentioned in the protocol description are logical values; i.e. “Bus Driven” is logical 1 and “Bus Not Driven” is logical 0. The electrical values are 0 for logical one and 1 for logical zero.

22.4 I/O APIC EMULATION

A basic I/O APIC Unit is emulated by the i960 Rx I/O processor with the APIC Bus Interface Unit and emulation software. APIC unit consists of:

- A set of interrupt input pins
- Interrupt Redirection Table
- An I/O APIC bus interface unit for sending and receiving APIC messages from the APIC bus

The I/O APIC bus interface unit is a dedicated hardware unit in the i960 Rx I/O processor and acts as an interface from the 80960 local bus to the APIC bus. The I/O APIC Unit provides the interrupt input pins on which I/O devices inject interrupts into the system in the form of an edge or a level. The I/O APIC also contains a Redirection Table with an entry for each interrupt input pin. Each entry in the Redirection Table can be individually programmed to indicate whether an interrupt on the pin is recognized as either an edge or a level; what vector and also what priority the interrupt has; and which of all possible processors should service the interrupt and how to select that processor (statically or dynamically). The information in the table is used to broadcast a message to all Local APIC units. Overall control of the I/O APIC is handled by emulation software executing within the i960 Rx I/O processor.

When the i960 core processor receives an interrupt that it determines should be sent as an APIC message, the emulation software looks up the information related to that interrupt in the Interrupt Redirection Table stored in local memory and writes that information to the APIC Bus Interface Unit which then sends the correct message on the APIC bus. The contents of the Redirection Table are under software control and defaults to a disabled state upon reset. The block diagram in Figure 22-2 shows how an I/O APIC Unit can be emulated in a i960 Rx I/O processor.

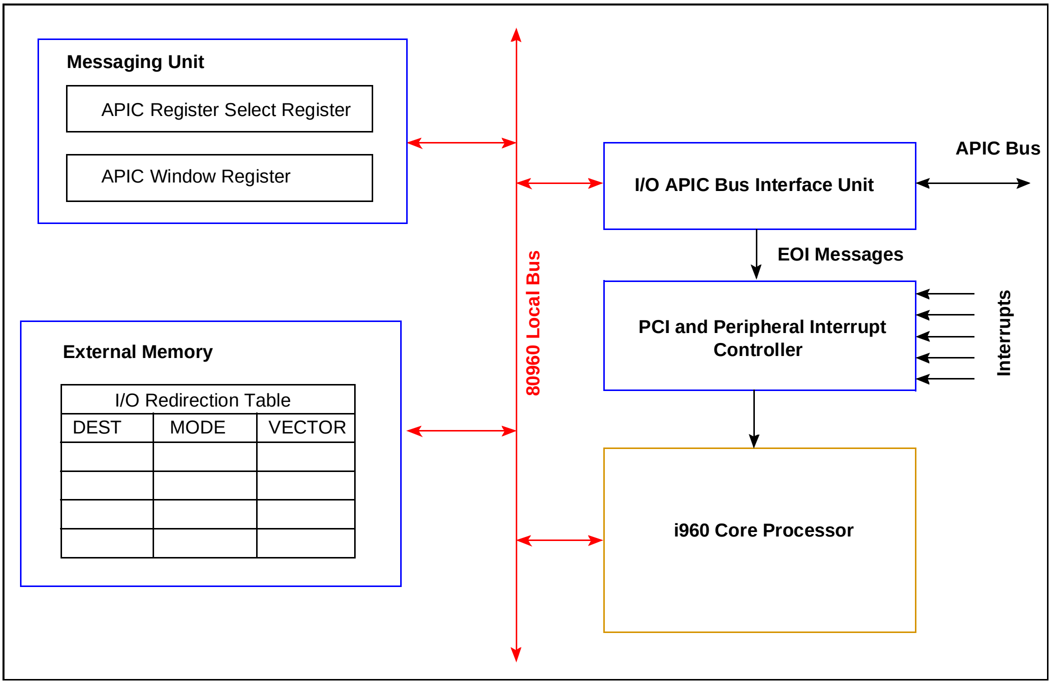


Figure 22-2. I/O APIC Emulation Block Diagram

The I/O Register Select Register and I/O Window Register are the only registers directly visible to the host software in the APIC architecture. These two registers are implemented in the Messaging Unit. The APIC Bus Interface Unit also has a set of Memory-mapped Registers (MMR) for transferring and receiving data over the APIC bus.

When the APIC register select register in the messaging unit is written, an interrupt is asserted to the i960 Rx I/O processor and the messaging unit locks out all other PCI accesses to the two messaging unit APIC registers. The emulation software then reads the logical APIC register at the offset contained in the APIC register select register (as defined by the APIC Architecture - not the local bus address mapping) and stores the value back into the APIC window register. The emulation software then clears the interrupt and the messaging unit releases the interlock mechanism to allow additional accesses to the APIC registers. The emulation software must also keep the value of the APIC window register updated when the redirection table changes due to interrupt activity.

When the APIC Window Register is written, an interrupt is asserted to the i960 core processor and the Messaging Unit locks out all other PCI accesses to the two Messaging Unit APIC registers. The emulation software reads the values of the APIC Register Select Register and APIC Window Register, updates the appropriate register and then clears the interrupt to release the interlock.

22.5 REGISTER DEFINITIONS

The APIC bus interface unit implements five registers that are mapped into local bus memory address space.

Table 22-1. I/O APIC Bus Interface Unit Register Summary

Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
22.5.1	APIC ID Register - APIC ID	22-6	32	0000 1780H	NA
22.5.2	APIC Arbitration Register - APIC ArbID	22-7	32	0000 1784H	NA
22.5.3	EOI Vector Register - EVR	22-8	32	0000 1788H	NA
22.5.4	Interrupt Message Register - IMR	22-8	32	0000 178CH	NA
22.5.5	APIC Control/Status Register - APIC CSR	22-11	32	0000 1790H	NA

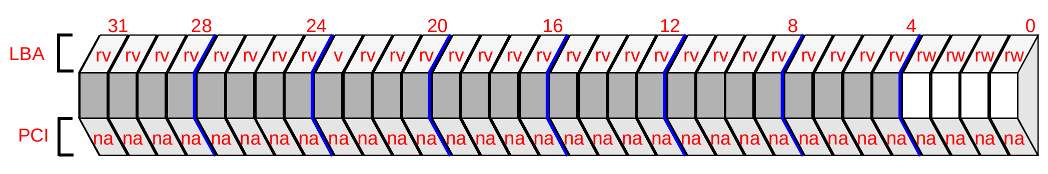
See APPENDIX C, MEMORY-MAPPED REGISTERS for details on where these registers are mapped. Two related registers from the messaging unit are:

- 17.7.1, APIC Register Select Register - ARSR (pg. 17-15)
- 17.7.2, APIC Window Register - AWR (pg. 17-15)

22.5.1 APIC ID Register - APIC ID

Each APIC unit has a register that contains the unit’s APIC ID. The ID serves as a physical name of the APIC unit. All APIC units using the APIC bus must have a unique four bit APIC ID. The APIC bus arbitration ID for the I/O unit is derived from its APIC ID. The APIC architecture allows for an 8 bit APIC ID but the APIC Bus is limited to 4 bits by its arbitration scheme. The APIC ID is read-write by software and must be programmed to a valid ID value before using the APIC bus for message transmission.

Table 22-2. APIC ID Register – APIC ID


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv v rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1780H
PCI: NA

Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:04	0000 000H	Reserved
03:00	0H	APIC ID - Unique APIC ID for the APIC bus interface unit.

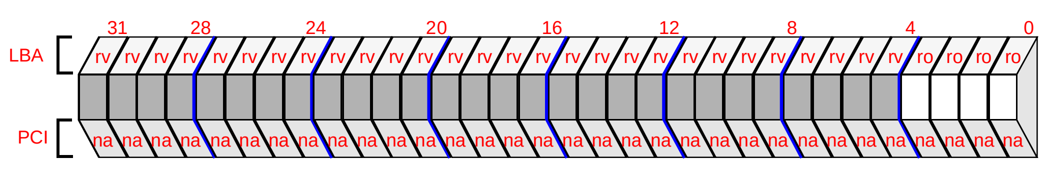
22.5.2 APIC Arbitration Register - APIC ArbID

The APIC Arbitration register (APIC ArbID) contains the current bus arbitration priority for the APIC bus interface register. It is written when the APIC ID register is written. It must be loaded with the new APIC ID value by the emulation software whenever APIC ID register is written. This register is valid only for those I/O APIC units that are directly connected to the APIC bus. When the I/O APIC is not connected to the APIC bus, this register returns the I/O APIC ID. A rotating priority scheme is used for APIC bus arbitration. The winner of the arbitration becomes the lowest priority agent and assumes an arbitration ID of 0.

When an APIC bus message successfully completes, all other agents, except the agent whose arbitration ID is 15, increments their arbitration IDs by one. The agent whose ID was 15 takes the winner's arbitration ID and increments it by one. Arbitration IDs are changed (incremented or assumed) only for messages that are transmitted successfully (except in the case of lowest priority local unit messages where arbitration ID is changed even when message was not successfully transmitted). A message is transmitted successfully when no CS error or acceptance error was reported for that message. I/O APIC arbitration ID register is always loaded with I/O APIC ID during a "level triggered INIT with deassert" message. The APIC Arbitration register is updated after the status 1 cycle. It determines the new Arb ID according to the following priorities:

- Write of the APIC ID Register. (ArbID = APICID)
- Successful INIT message. (ArbID = APICID)
- Successful message and Arb ID is the same as the winner ID. (ArbID = 0)
- Successful message and Arb ID = 1111. (ArbID = WinnerID + 1)
- Successful message. (ArbID = ArbID + 1)
- Otherwise, ArbID remains the same.

Table 22-3. APIC Arbitration ID Register – APIC ArbID


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1784H
PCI: NA
	Legend:
	RV = Reserved
	RS = Read/Set
	LBA = 80960 Local Bus Address
		NA = Not Accessible
		PR = Preserved
		RC = Read Clear
			RO = Read Only
			RW = Read/Write
			
			PCI = PCI Configuration Address Offset

Bit	Default	Description
31:04	0000 000H	Reserved
03:00	0H	APIC Arbitration ID - Current bus arbitration priority for the APIC bus interface unit.

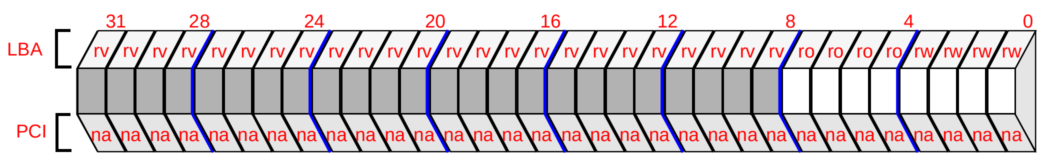
22.5.3 EOI Vector Register - EVR

The EOI Vector Register (EVR) is a read-write register although it is not written by the emulation software in normal operation. It is set to the EOI Vector by the APIC bus interface unit when an EOI message is received on the APIC bus.

To ensure that the value read is valid, the EOI flow control bit should be set and the EOI received bit should indicate that an EOI has been received. When the EOI flow control bit is not set, another EOI could be received just as the EVR is being read resulting in corrupt read data. When the value of the EOI vector is not important, then this is not an issue.

The data read from the APIC bus is only written into the EVR register when the checksum was OK (Status0) and an Accept status was written (Status1).

Table 22-4. EOI Vector Register – EVR


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv ro ro ro ro rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1788H
PCI: NA

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 Local Bus Address
PCI = PCI Configuration Address Offset

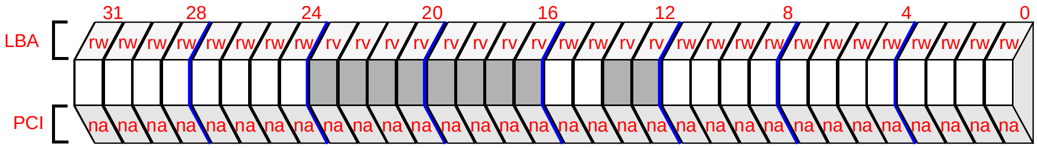
Bit	Default	Description
31:08	0000 00H	Reserved
07:00	00H	EOI Vector - EOI Vector received from the APIC Bus.

22.5.4 Interrupt Message Register - IMR

The Interrupt Message Register (IMR - Table 22-5) is a 32-bit register used to provide the data to be sent in the APIC interrupt message. To the APIC Bus Interface Unit, the IMR register is a data register, holding data that may need to be sent on the APIC bus.

IMR fields have a 1-to-1 correspondence with the fields defined for the APIC redirection table entries and the bits sent in the APIC interrupt message itself. This register should only be written when the send message bit and message sent bit in the APIC CSR indicate that the APIC send unit is Idle. When emulation software aborts a message by clearing the send message bit before the message sent bit is set, the emulation software should wait for at least 30 APIC clocks (1.8 ms at 16 MHz) before attempting to write a new value to the IMR.

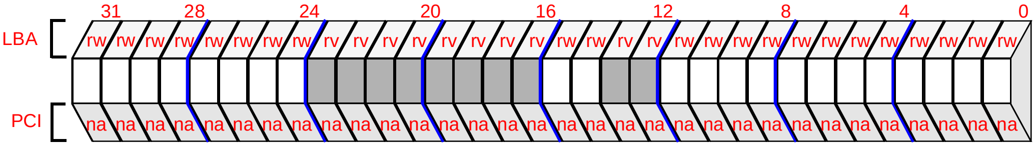
Table 22-5. Interrupt Message Register – IMR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
rw rw rw rw rw rw rw rw rv rv rv rv rv rv rv rv rw rw rv rv rw rw rw rw rw rw rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 178CH
PCI: NA
	Legend: NA = Not Accessible RO = Read Only
	RV = Reserved PR = Preserved RW = Read/Write
	RS = Read/Set RC = Read Clear
	LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:24	00H	Destination - contains an APIC ID or destination address. When the destination mode of this entry is physical mode, then the bits 27:24 contain an APIC ID. When logical mode, then the destination field potentially defines a set of processors. Bits 31:24 of the destination field specify the logical destination address.
23:16	00H	Reserved
15	0_2	Trigger Mode - indicates the type of signal on the interrupt pin that triggers an interrupt. A value of 0 means the input is edge sensitive and a value of 1 means the input is level sensitive.
14	0_2	Level - only used when delivery mode is set to INIT (101). When the delivery mode is not set to INIT, this bit must be set to 0. The meaning for INIT is: • 0 - INIT deasserted • 1 - INIT asserted
13:12	00_2	Reserved
11	0_2	Destination Mode - determines the interpretation of the destination field: • 0 (Physical Mode): a destination APIC is identified by its ID. Bits 24 through 27 of the destination field specify the 4-bit APIC ID. • 1 (Logical Mode): destinations are identified by matching on logical destination under the control of the destination format register and logical destination register in each local APIC. Bits 24 through 31 (8 MSB) of the destination field specify the 8-bit APIC ID

Table 22-5. Interrupt Message Register – IMR (Sheet 2 of 2)



	31			28				24				20				16				12				8				4				0
LBA	rw	rw	rw	rw	rw	rw	rw	rw	rv	rv	rv	rv	rv	rv	rv	rw	rw	rv	rv	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
PCI	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

LBA: 178CH
PCI: NA

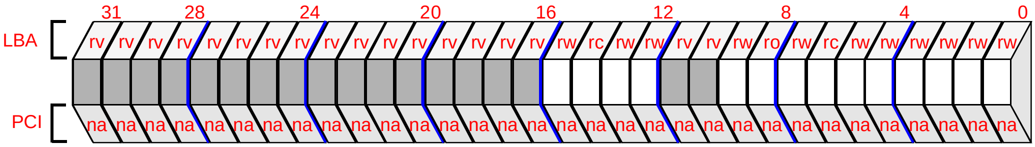
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
10:08	000_2	Delivery Mode - a 3-bit field that specifies how the APICs listed in the destination field should act upon reception of this signal. Note that certain delivery modes only operates as intended when used in conjunction with a specific trigger mode. These restrictions are indicated in the table below for each delivery mode. • 000_2 (Fixed): Delivers the signal on the INTR signal of all processor cores listed in the destination. trigger mode for “fixed” delivery mode can be edge or level. • 001_2 (Lowest Priority Local Unit): Delivers the signal on the INTR signal of the processor core that is executing at the lowest priority among all the processors listed in the destination. trigger mode for “lowest priority” delivery mode can be edge or level. • 010_2 System Management Interrupt (SMI): A delivery mode equal to “SMI” requires an “edge” trigger mode. The vector information is ignored but must be programmed to all zeroes for future compatibility. • 100_2 (NMI): Delivers the signal on the NMI signal of all processor cores listed in the destination; vector information is ignored. “NMI” is treated as an edge-triggered interrupt even when programmed as a level-triggered interrupt. • 101_2 (INIT): Delivers the signal to all processor cores listed in the destination by asserting the INIT signal. All addressed local APICs assume their INIT state. INIT is always treated as an edge-triggered interrupt even when programmed otherwise. • 111_2 (ExtINT): Delivers the signal to the INTR signal of all processor cores listed in the destination as an interrupt that originated in an externally connected (8259A-compatible) interrupt controller. The INTA cycle that corresponds to this ExtINT delivery is routed to the external controller that is expected to supply the vector. A delivery mode of “ExtINT” requires an “edge” trigger mode.
07:00	00H	Vector - the interrupt vector for this interrupt. Vector values range between 10H and FEH.

22.5.5 APIC Control/Status Register - APIC CSR

The APIC Control/Status Register (APIC CSR - Table 22-6) is used to control and monitor the status of the APIC bus interface unit. The lower byte is used for sending APIC interrupt messages and the upper byte is used for receiving APIC EOI messages. When the i960 core processor tries to update this register at the same time the APIC Bus Interface Unit is updating the register, the Unit update has precedence.

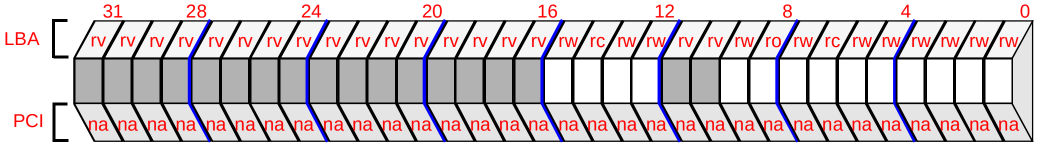
Table 22-6. APIC Control/Status Register – APIC CSR (Sheet 1 of 2)


31
28
24
20
16
12
8
4
0
LBA
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rc rw rw rv rv rw ro rw rc rw rw rw rw rw rw
PCI
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1790H
PCI: NA
		Legend: NA = Not Accessible RO = Read Only
		RV = Reserved PR = Preserved RW = Read/Write
		RS = Read/Set RC = Read Clear
		LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	**Default**	**Description**
31:16	0000H	Reserved
15	0_2	EOI Flow Control - Determines whether or not to accept an EOI message from the APIC bus. • 0 - The APIC bus interface unit always accepts an EOI message. • 1 - The unit waits for the i960 Rx I/O processor to process the previous EOI before accepting another. See Table 22-4., EOI Vector Register – EVR (pg. 22-8).
14	0_2	EOI Received - Used to indicate that an APIC EOI message has been received. This is a read/clear bit.
13	0_2	EOI Interrupt Enable: • 0 - An APIC EOI does not interrupt the i960 Rx I/O processor. • 1 - An APIC EOI can interrupt the i960 Rx I/O processor.
12	0_2	APIC Bus Interface Enable - controls the pin multiplexers for the APIC bus pins. • 0 - The APIC pins are disabled. • 1 - The APIC bus are enabled.
11:10	00_2	Reserved
09	0_2	APIC Bus Interface Unit Reset: • 0 - The APIC unit is not reset. • 1 - The APIC unit is reset. The APIC PMMR registers are not reset.

Table 22-6. APIC Control/Status Register – APIC CSR (Sheet 2 of 2)


31
28
24
20
16
12
8
4
0
LBA
PCI
rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rw rc rw rw rv rv rw ro rw rc rw rw rw rw rw rw
na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na na

LBA: 1790H
PCI: NA

Legend:
NA = Not Accessible
RO = Read Only
RV = Reserved
PR = Preserved
RW = Read/Write
RS = Read/Set
RC = Read Clear
LBA = 80960 Local Bus Address
PCI = PCI Configuration Address Offset

Bit	Default	Description
08	0_2	PICCLK Active - used by software to determine when the APIC Bus Interface Unit is connected to an APIC bus. • 0 - No PICCLK is present. • 1 - The PICCLK input is active (i.e. has toggled at least 3 times since reset was deasserted).
07	0_2	Send Message - tells the APIC bus interface unit to take the information in the IMR and send an interrupt message on the APIC bus. The APIC Bus Interface attempts to send a message whenever this bit is set and the message sent bit is reset.
06	0_2	Message Sent - indicates when an APIC interrupt message is sent. To ensure that it was received by a local APIC, the status field must be checked.
05	0_2	Message Sent Interrupt Enable: • 0 - The APIC bus interface does not interrupt the i960 Rx I/O processor after sending the message. • 1 - The APIC bus interface interrupts the i960 Rx I/O processor • after the message sent interrupt.
04:00	0H	APIC Message Status - Are read/write for testability but are normally just set by the APIC bus interface unit after sending an APIC interrupt message. Bits 3:2 are status0 and bits 1:0 are status1. Bit 4 is set when an error on status2 occurs during a lowest priority local unit arbitration message. Cleared for all other messages.

23

TEST FEATURES

CHAPTER 23
TEST FEATURES

This chapter describes the i960® Rx I/O processor test features, including ONCE (On-Circuit Emulation) and boundary-scan (JTAG). Together these two features create a powerful environment for design debug and fault diagnosis.

23.1 ON-CIRCUIT EMULATION (ONCE)

On-circuit emulation aids board-level testing. This feature allows a mounted i960 Rx I/O processor to electrically "remove" itself from a circuit board. This allows for system-level testing where a remote tester exercises the processor system. In ONCE mode, the processor presents a high impedance on every pin, except for the JTAG test data Output (TDO). All pullup transistors present on input pins are also disabled and internal clocks stop. In this state the processor's power demands on the circuit board are nearly eliminated. Once the processor is electrically removed, a functional tester such as an In-Circuit Emulator (ICE) system can emulate the mounted processor and execute a test of the i960 Rx I/O processor system.

NOTE: Do not use ONCE mode with boundary-scan (JTAG). See section 23.1.2, ONCE Mode and Boundary-Scan (JTAG) are Incompatible (pg. 23-2).

23.1.1 Entering/Exiting ONCE Mode

The ONCE# pin, in concert with the RESET# pin, invokes ONCE mode.

To invoke ONCE mode, assert the ONCE# pin (low) while the processor is in the reset state. (The processor recognizes the ONCE# pin signal only while RESET# is asserted.) The processor enters ONCE mode immediately. The rising edge of RESET# latches the ONCE# pin state until RESET# goes true again.

Enter ONCE mode by asserting the following sequence with an external tester:

1. Drive the ONCE# pin low (overcoming the internal pull-up resistor).
2. Initiate a normal reset cycle.
3. After the RESET# pin goes high again, the ONCE# pin can be deasserted.

Exit ONCE mode, by performing a normal reset with the RESET# pin while holding the ONCE# pin high. A power off-on cycle is not necessary to exit ONCE mode.

See the *80960RP/RD Intelligent I/O Microprocessor* Data Sheets (Intel Literature order #272737) for specific timing of the ONCE# pin and the characteristics of the on-circuit emulation mode.

23.1.2 ONCE Mode and Boundary-Scan (JTAG) are Incompatible

Permanent damage can occur when an in-circuit emulator is used concurrently with boundary-scan (JTAG). Do not use any system that relies on ONCE mode when using boundary-scan. Signal contentions and resultant damage may occur if an external system, such as an emulator development system, invokes ONCE mode and manipulates the i960 Rx I/O processor signals while JTAG is active.

Since the i960 Rx I/O processor i960 Rx I/O processor complies fully with IEEE Std. 1149.1, JTAG boundary-scan instructions always override ONCE mode. While ONCE mode intends to disable all processor outputs so an external emulator can drive them, JTAG boundary-scan can enable those outputs, causing contention with the external emulator.

To avoid damage, and as a general design rule, force TRST# low to disable boundary-scan whenever ONCE mode is active.

23.1.3 How to use the Data Enable (DEN#) Signal with an In-Circuit Emulator

When using an ICE in an 80960Rx system, the use of the Data Enable signal (DEN#) is not recommended. This section describes how DEN# operates and a recommended solution for using it with an In-Circuit Emulator (ICE).

DEN# *Operation:* When asserted, DEN# indicates data transfer cycles during a bus access. DEN# asserts at the start of the first data cycle in a bus access and de-asserts at the end of the last data cycle. DEN# can be used in conjunction with DT/R# to provide control for data transceivers connected to the data bus.

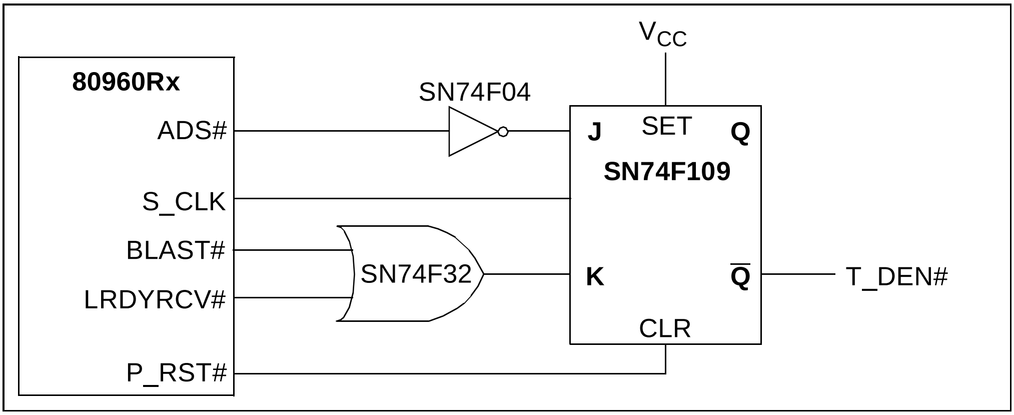
Using DEN# *with an In-Circuit Emulator:* For ICE users, it is not recommended to use the 80960Rx's DEN# signal directly to transceivers**.** When executing an ICE microcode transaction, the expected behavior is that DEN# would remain de-asserted during the entire transaction. However, DEN# asserts as described above. If the design uses DEN# to enable transceivers, the transceivers will be enabled. This may result in bus contention.

The use of DEN# in 80960Jx designs was possible because the 80960Jx was on a "POD". The POD was cabled to the target board where it plugged in to a socket. The POD masks out DEN# during ICE microcode transactions. The 80960Rx's package does not allow the use of a POD; consequently, the ICE signals connect directly to the target system and the DEN# signal cannot be masked.

23.1.3.1 DEN# Alternatives

To use an ICE with your 80960Rx design, alternatives to DEN# are:

- Ground the OE# pin of the transceiver
- Re-create a DEN# signal with the circuit shown below



The circuit asserts T_DEN# (Q#) at the start of the first data cycle when ADS# asserts and BLAST# and LRDYRCV# de-asserts. T_DEN# deasserts at the end of the last data cycle when ADS# de-asserts and BLAST# and LRDYRCV# assert. During RESET, T_DEN# de-asserts.

Equivalent components may be used in place of the components shown.

23.2 BOUNDARY-SCAN (JTAG)

The i960 Rx I/O processor provides test features compliant to IEEE standard test access port and boundary-scan architecture (IEEE Std. 1149.1). JTAG ensures that components function correctly, connections between components are correct, and components interact correctly on the printed circuit board.

To date, the i960 Hx, Jx and Rx processors implement IEEE 1149.1 standard test access port and boundary-scan architecture, and i960 Kx, Sx and Cx processors do not. For information about using JTAG in a design, refer to IEEE Std. 1149.1 (available from the Institute of Electrical and Electronics Engineers Inc., 345 E. 47th St., New York, NY 10017).

NOTE: Do not use ONCE mode with boundary-scan (JTAG). See section 23.1.2, ONCE Mode and Boundary-Scan (JTAG) are Incompatible (pg. 23-2).

23.2.1 Boundary-Scan Architecture

Boundary-scan test logic consists of a boundary-scan register and support logic. These are accessed through a Test Access Port (TAP). The TAP provides a simple serial interface that allows all processor signal pins to be driven and/or sampled, thereby providing direct control and monitoring of processor pins at the system level.

This mode of operation is valuable for design debugging and fault diagnosis since it permits examination of connections not normally accessible to the test system. The following subsections describe the boundary-scan test logic elements: TAP pins, instruction register, test data registers and TAP controller. Figure 23-1 illustrates how these pieces fit together to form the JTAG unit.

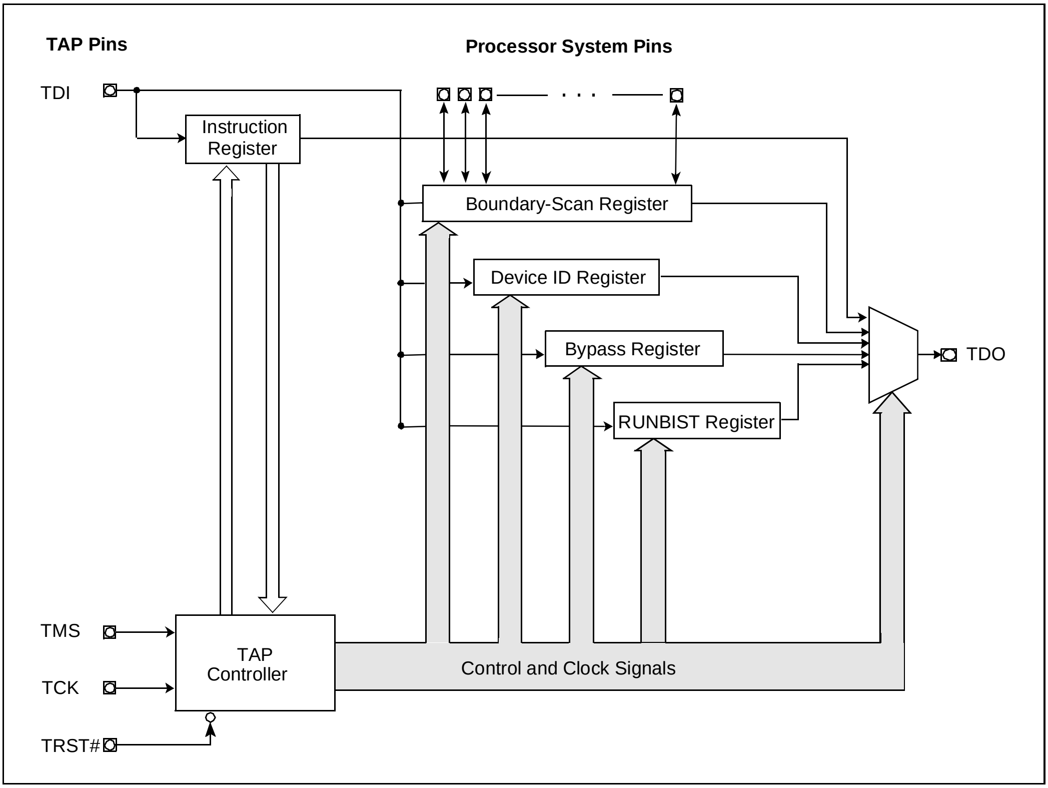


Figure 23-1. Test Access Port Block Diagram

23.2.2 TAP Pins

The i960 Rx I/O processor's TAP pins form a serial port composed of four input connections (TMS, TCK, TRST# and TDI) and one output connection (TDO). These pins are described in Table 23-1. The TAP pins provide access to the instruction register and the test data registers.

Table 23-1. TAP Controller Pin Definitions

Pin Name	Type	Definition
TCK	Input	**Test Clock** provides the clock for the JTAG logic. The JTAG test logic retains its state indefinitely when TCK is stopped at "0" or "1".
TMS	Input	**Test Mode** is decoded by the TAP controller state machine to control test operations. TMS is sampled by the test logic on the rising edge of TCK. TMS is pulled high internally when not driven.
TDI	Input	**Test Data Input** is the serial port where test instructions and data is received by the test logic. Signals presented at TDI are sampled into the test logic on the rising edge of TCK. TDI is pulled high internally when not driven. Data shifted into TDI is not inverted on its way to the TDO input.
TDO	Output	**Test Data Output** is the serial output for test instructions and data from the JTAG test logic. Changes in the state of TDO occur only on the falling edge of TCK. The TDO output is active only during data shifting (SHDR or SHIR); it is inactive (high-Z) at all other times.
TRST#	Input	**Test Reset** provides for an asynchronous initialization of the TAP controller. Asserting a logic "0" on this pin puts the TAP controller state machine and all other test logic on the processor in the *Test-Logic-Reset* (initial) state. TRST# is pulled high internally when not driven. **Note:** The system must ensure that TRST# is asserted after power-up in order to put the TAP controller in a known state. Failure to do so may cause improper processor operation.

23.2.3 Instruction Register

The Instruction Register (IR) holds instruction codes. These codes are shifted in through the Test Data Input (TDI) pin. The instruction codes are used to select the specific test operation to be performed and the test data register to be accessed.

The instruction register is a parallel-loadable, master/slave-configured 4-bit wide, serial-shift register with latched outputs. Data is shifted into and out of the IR serially through the TDI pin clocked by the rising edge of TCK when the TAP controller is in the Shift_IR state. The shifted-in instruction becomes active upon latching from the master stage to the slave stage in the Update_IR state. At that time the IR outputs along with the TAP finite state machine outputs are decoded to select and control the test data register selected by that instruction. Upon latching, all actions caused by any previous instructions will terminate.

The instruction determines the test to be performed, the test data register to be accessed, or both (see Table 23-2). The IR is four bits wide. When the IR is selected in the Shift_IR state, the most significant bit is connected to TDI, and the least significant bit is connected to TDO. The value presented on the TDI pin is shifted into the IR on each rising edge of TCK, as long as the TAP controller remains in the Shift_IR state. When the TAP controller changes to the Capture_IR state, fixed parallel data (0001_2) is captured. During Shift_IR, when a new instruction is shifted in through TDI, the value 0001_2 is always shifted out through TDO, least significant bit first. This helps identify instructions in a long chain of serial data from several devices.

Upon activation of the TRST# reset pin, the latched instruction asynchronously changes to the **idcode** instruction. When the TAP controller moves into the Test_Logic_Reset state other than by reset activation, the opcode changes as TDI is shifts, and becomes active on the falling edge of TCK. See Figure 23-4 for an example of loading the instruction register.

23.2.3.1 Boundary-Scan Instruction Set

The i960 Rx I/O processor supports three mandatory boundary-scan instructions (**bypass**, **sample**/**preload** and **extest)** plus four additional public instructions (**idcode**, **clamp**, **highz** and **runbist)**. Table 23-2 lists the i960 Rx I/O processor's boundary-scan instruction codes. Those codes listed as "not used" or "private" should not be used.

Table 23-2. Boundary-Scan Instruction Set

Instruction Code	Instruction Name	Instruction Code	Instruction Name
0000_2	**extest**	1000_2	**highz**
0001_2	**sample/preload**	1001_2	not used
0010_2	**idcode**	1010_2	not used
0011_2	not used	1011_2	private
0100_2	**clamp**	1100_2	private
0101_2	not used	1101_2	not used
0110_2	not used	1110_2	not used
0111_2	**runbist**	1111_2	**bypass**

Table 23-3. IEEE Instructions (Sheet 1 of 2)

Instruction / Requisite	Opcode	Description
extest IEEE 1149.1 Required	0000_2	**extest** initiates testing of external circuitry, typically board-level interconnects and off chip circuitry. **extest** connects the boundary-scan register between TDI and TDO in the Shift_DR state only. When **extest** is selected, all output signal pin values are driven by values shifted into the boundary-scan register and may change only on the falling edge of TCK in the Update_DR state. Also, when **extest** is selected, all system input pin states must be loaded into the boundary-scan register on the rising-edge of TCK in the Capture_DR state. Values shifted into input latches in the boundary-scan register are never used by the processor's internal logic.
sample/ preload IEEE 1149.1 Required	0001_2	**sample**/**preload** performs two functions: • When the TAP controller is in the Capture-DR state, the **sample** instruction occurs on the rising edge of TCK and provides a snapshot of the component's normal operation without interfering with that normal operation. The instruction causes boundary-scan register cells associated with outputs to sample the value being driven by or to the processor. • When the TAP controller is in the Update-DR state, the **preload** instruction occurs on the falling edge of TCK. This instruction causes the transfer of data held in the boundary-scan cells to the slave register cells. Typically the slave latched data is applied to the system outputs via the **extest** instruction.
idcode IEEE 1149.1 Optional	0010_2	**idcode** is used in conjunction with the device identification register. It connects the device identification register between TDI and TDO in the Shift_DR state. When selected, **idcode** parallel-loads the hard-wired identification code (32 bits) into the device identification register on the rising edge of TCK in the Capture_DR state. **NOTE**: The device identification register is not altered by data being shifted in on TDI.
runbist i960 Rx I/O processor Optional	0111_2	**runbist** selects the one-bit RUNBIST register, loads a value of 1 into it and connects it to TDO. It also initiates the processor's built-in self test (BIST) feature which is able to detect approximately 82% of all the possible stuck-at faults on the device. The processor AC/DC specifications for V_{CC} and CLKIN must be met and RESET# must be de-asserted prior to executing **runbist**. After loading **runbist** instruction code into the instruction register, the TAP controller must be placed in the Run-Test/Idle state. BIST begins on the first rising edge of TCK after the Run-Test/Idle state is entered. The TAP controller must remain in the Run-Test/Idle state until BIST is completed. **runbist** requires approximately 414,000 core cycles to complete BIST and report the result to the RUNBIST register. The results are stored in bit 0 of the RUNBIST register. After the report completes, the value in the RUNBIST register is shifted out on TDO during the Shift-DR state. A value of 0 being shifted out on TDO indicates BIST completed successfully. A value of 1 indicates a failure occurred. After BIST completes, the processor must be cycled through the reset state to resume normal operation.

Table 23-3. IEEE Instructions (Sheet 2 of 2)

Instruction / Requisite	Opcode	Description
bypass IEEE 1149.1 Required	1111_2	**bypass** instruction selects the one-bit bypass register between TDI and TDO pins while in SHIFT_DR state, effectively bypassing the processor's test logic. 0_2 is captured in the CAPTURE_DR state. This is the only instruction that accesses the bypass register. While this instruction is in effect, all other test data registers have no effect on system operation. Test data registers with both test and system functionality perform their system functions when this instruction is selected.
highz	1000_2	Executing **highz** generates a signal that is read on the rising-edge of RESET#. When this signal is found asserted, the device is put into the ONCE mode (all output pins are floated). Also, when this instruction is active, the Bypass register is connected between TDI and TDO. This register can be accessed via the JTAG Test-Access Port throughout the device operation. Access to the Bypass register can also be obtained with the **bypass** instruction. **highz** provides an alternate method of entering ONCE mode.
clamp	0100_2	**clamp** instruction allows the state of the signals driven from the i960 Jx processor pins to be determined from the boundary-scan register while the BYPASS register is selected as the serial path between TDI and TDO. Signals driven from the component pins will not change while the **clamp** instruction is selected.

23.2.4 TAP Test Data Registers

The i960 Rx I/O processor contains four test data registers (device identification, bypass, RUNBIST and boundary-scan). Each test data register selected by the TAP controller is connected serially between TDI and TDO. TDI is connected to the test data register's most significant bit. TDO is connected to the least significant bit. Data is shifted one bit position within the register towards TDO on each rising edge of TCK. While any register is selected, data is transferred from TDI to TDO without inversion. The following sections describe each of the test data registers. See Figure 23-5 for an example of loading the data register.

23.2.4.1 Device Identification Register

The device identification register is a 32-bit register containing the manufacturer's identification code, part number code, version code and other information in the format shown in the *80960RP/RD Intelligent I/O Microprocessor* Data Sheets. The format of the register is discussed in section 11.5, DEVICE IDENTIFICATION ON RESET (pg. 11-23). The identification register is selected only by the **idcode** instruction. When the TAP controller's Test_Logic_Reset state is entered, **idcode** is asynchronously loaded into the instruction register. The device identification register loads the fixed parallel input value in the Capture_DR state.

23.2.4.2 Bypass Register

The required bypass register, a one-bit shift register, provides the shortest path between TDI and TDO when a **bypass** instruction is in effect. This allows rapid movement of test data to and from other components on the board. This path can be selected when no test operation is being performed on the processor.

23.2.4.3 RUNBIST Register

The RUNBIST register, a one-bit register, contains the result of the execution of the processor's BIST routine. After the built-in self-test completes, the processor must be cycled through the reset state to resume normal operation. See section 11.3.1, Self Test Function (STEST, FAIL#) (pg. 11-9) for details of the built-in self test algorithm. The processor runs the BIST routine when the TAP controller enters the Test_Logic_Reset state while the **runbist** instruction is selected.

23.2.4.4 Boundary-Scan Register

The boundary-scan register contains a cell for each pin as well as control cells for I/O and the HIGHZ pin.

Table 23-4 shows the bit order of the i960 Rx I/O processor boundary-scan register. All table cells that contain "Control" select the direction of bidirectional pins or HIGHZ output pins. When a "0" is loaded into the control cell, the associated pin(s) are HIGHZ or selected as input.

The boundary-scan register is a required set of serial-shiftable register cells, configured in master/slave stages and connected between each of the i960 Rx I/O processor's pins and on-chip system logic. The V_{CC}, V_{SS} and JTAG pins are NOT in the boundary-scan chain.

The boundary-scan register cells are dedicated logic and do not have any system function. Data may be loaded into the boundary-scan register master cells from the device input pins and output pin-drivers in parallel by the mandatory **sample**/**preload** and **extest** instructions. Parallel loading takes place on the rising edge of TCK in the Capture_DR state.

Data may be scanned into the boundary-scan register serially via the TDI serial input pin, clocked by the rising edge of TCK in the Shift_DR state. When the required data has been loaded into the master-cell stages, it can be driven into the system logic at input pins or onto the output pins on the falling edge of TCK in the Update_DR state. Data may also be shifted out of the boundary-scan register by means of the TDO serial output pin at the falling edge of TCK.

Table 23-4. i960® Rx I/O Processor Boundary Scan Register Bit Order (Sheet 1 of 3)

Bit	Signal	Input/Output	Bit	Signal	Input/Output
0	BLAST#	I/O	133	control	enable cell
1	DEN#	I/O	134	control	enable cell
2	DT/R#	O	135	control	enable cell
3	W/R#	I/O	136	control	enable cell
4	BE#(0)	I/O	137	P_AD(31)	I/O
5	BE#(1)	I/O	138	P_AD(30)	I/O
6	control	enable cell	139	P_AD(29)	I/O
7	control	enable cell	140	P_AD(28)	I/O
8	control	enable cell	141	P_AD(27)	I/O
9	BE#(2)	I/O	142	P_AD(26)	I/O
10	BE#(3)	I/O	143	P_AD(25)	I/O
11	ADS#	I/O	144	P_AD(24)	I/O
12	ALE	O	145	P_C/BE#(3)	I/O
13	LRDYRCV#	O	146	control	enable cell
14	RDYRCV#	I/O	147	P_IDSEL	I
15	control	enable cell	148	P_AD(23)	I/O
16	AD(0)	I/O	149	P_AD(22)	I/O
17	AD(1)	I/O	150	P_AD(21)	I/O
18	AD(2)	I/O	151	P_AD(20)	I/O
19	AD(3)	I/O	152	P_AD(19)	I/O
20	AD(4)	I/O	153	P_AD(18)	I/O
21	AD(5)	I/O	154	P_AD(17)	I/O
22	AD(6)	I/O	155	P_AD(16)	I/O
23	AD(7)	I/O	156	control	enable cell
24	AD(8)	I/O	157	control	enable cell
25	AD(9)	I/O	158	P_C/BE#(2)	I/O
26	AD(10)	I/O	159	P_FRAME#	I/O
27	AD(11)	I/O	160	control	enable cell
28	AD(12)	I/O	161	control	enable cell
29	AD(13)	I/O	162	P_IRDY#	I/O
30	AD(14)	I/O	163	P_TRDY#	I/O
31	AD(15)	I/O	164	P_DEVSEL#	I/O
32	control	enable cell	165	P_STOP#	I/O
33	AD(16)	I/O	166	P_LOCK#	I
34	AD(17)	I/O	167	P_PERR#	I/O
35	AD(18)	I/O	168	P_SERR#	O
36	AD(19)	I/O	169	P_PAR	I/O
37	AD(20)	I/O	170	P_C/BE#(1)	I/O
38	AD(21)	I/O	171	control	enable cell
39	AD(22)	I/O	172	control	enable cell
40	AD(23)	I/O	173	control	enable cell
41	AD(24)	I/O	174	P_AD(15)	I/O
42	AD(25)	I/O	175	P_AD(14)	I/O
43	AD(26)	I/O	176	P_AD(13)	I/O
44	AD(27)	I/O	177	P_AD(12)	I/O

Table 23-4. i960® Rx I/O Processor Boundary Scan Register Bit Order (Sheet 2 of 3)

Bit	Signal	Input/Output	Bit	Signal	Input/Output
45	AD(28)	I/O	178	P_AD(11)	I/O
46	AD(29)	I/O	179	P_AD(10)	I/O
47	AD(30)	I/O	180	P_AD(9)	I/O
48	AD(31)	I/O	181	P_AD(8)	I/O
49	control	enable cell	182	control	enable cell
50	MA(0)	O	183	P_C/BE#(0)	I/O
51	MA(1)	O	184	P_AD(7)	I/O
52	MA(2)	O	185	P_AD(6)	I/O
53	MA(3)	O	186	P_AD(5)	I/O
54	MA(4)	O	187	P_AD(4)	I/O
55	MA(5)	O	188	P_AD(3)	I/O
56	MA(6)	O	189	P_AD(2)	I/O
57	MA(7)	O	190	P_AD(1)	I/O
58	MA(8)	O	191	P_AD(0)	I/O
59	MA(9)	O	192	S_AD(0)	I/O
60	MA(10)	O	193	S_AD(1)	I/O
61	MA(11)	O	194	S_AD(2)	I/O
62	control	enable cell	195	S_AD(3)	I/O
63	DP(0)	I/O	196	S_AD(4)	I/O
64	DP(1)	I/O	197	S_AD(5)	I/O
65	DP(2)	I/O	198	S_AD(6)	I/O
66	DP(3)	I/O	199	S_AD(7)	I/O
67	RAS#(0)	I/O	200	control	enable cell
68	RAS#(1)	I/O	201	S_C/BE#(0)	I/O
69	RAS#(2)	I/O	202	S_AD(8)	I/O
70	RAS#(3)	I/O	203	S_AD(9)	I/O
71	CAS#(0)	I/O	204	S_AD(10)	I/O
72	CAS#(1)	I/O	205	S_AD(11)	I/O
73	CAS#(2)	I/O	206	S_AD(12)	I/O
74	CAS#(3)	I/O	207	S_AD(13)	I/O
75	CAS#(4)	I/O	208	S_AD(14)	I/O
76	CAS#(5)	I/O	209	S_AD(15)	I/O
77	CAS#(6)	I/O	210	S_C/BE#(1)	I/O
78	CAS#(7)	I/O	211	S_PAR	I/O
79	MWE#(0)	O	212	S_SERR#	I/O
80	MWE#(1)	O	213	control	enable cell
81	MWE#(2)	O	214	control	enable cell
82	MWE#(3)	O	215	control	enable cell
83	DWE#(0)	O	216	S_PERR#	I/O
84	DWE#(1)	O	217	S_LOCK#	I/O
85	CE#(0)	O	218	S_STOP#	I/O
86	CE#(1)	O	219	S_DEVSEL#	I/O
87	LEAF#(0)	O	220	S_TRDY#	I/O
88	LEAF#(1)	O	221	S_IRDY#	I/O
89	DALE(0)	O	222	S_FRAME#	I/O

Table 23-4. i960® Rx I/O Processor Boundary Scan Register Bit Order (Sheet 3 of 3)

Bit	Signal	Input/Output	Bit	Signal	Input/Output
90	DALE(1)	O	223	S_C/BE#(2)	I/O
91	WAIT#	I/O	224	control	enable cell
92	S_INTA/XINT#0	I	225	control	enable cell
93	S_INTB/XINT#1	I	226	control	enable cell
94	S_INTC/XINT#2	I	227	control	enable cell
95	S_INTD/XINT#3	I	228	control	enable cell
96	control	enable cell	229	S_AD(16)	I/O
97	XINT#4	I	230	S_AD(17)	I/O
98	XINT#5	I	231	S_AD(18)	I/O
99	XINT#6	I	232	S_AD(19)	I/O
100	XINT#7	I	233	S_AD(20)	I/O
101	NMI#	I	234	S_AD(21)	I/O
102	control	enable cell	235	S_AD(22)	I/O
103	control	enable cell	236	S_AD(23)	I/O
104	control	enable cell	237	control	enable cell
105	control	enable cell	238	S_IDSEL	I
106	PICD(0)	I/O	239	S_C/BE#(3)	I/O
107	PICD(1)	I/O	240	S_AD(24)	I/O
108	PICCLK	I	241	S_AD(25)	I/O
109	SCL	I/O	242	S_AD(26)	I/O
110	SDA	I/O	243	S_AD(27)	I/O
111	HOLDA	O	244	S_AD(28)	I/O
112	HOLD	I	245	S_AD(29)	I/O
113	DACK#	O	246	S_AD(30)	I/O
114	DREQ#	I	247	S_AD(31)	I/O
115	STEST	I	248	S_RST#	O
116	LOCK#/ONCE#	I/O	249	S_REQ0/S_GNT#	I
117	D/C#/RST_MODE#	I/O	250	S_GNT0/S_REQ#	O
118	FAIL#	O	251	S_REQ#(0)	I
119	WIDTH/HLTD0/SYNC	I/O	252	S_GNT#(0)	O
120	WIDTH/HLTD1/RETRY	I/O	253	S_REQ#(1)	I
121	LRST#	O	254	S_GNT#(1)	O
122	control	enable cell	255	S_REQ#(2)	I
123	control	enable cell	256	S_GNT#(2)	O
124	P_INTA#	O	257	control	enable cell
125	P_INTB#	O	258	control	enable cell
126	P_INTC#	O	259	S_CLK	I
127	P_INTD#	O	260	S_REQ#(3)	I
128	P_RST#	I	261	S_GNT#(3)	O
129	P_CLK	I	262	S_REQ#(4)	I
130	P_GNT#	I	263	S_GNT#(4)	O
131	P_REQ#	O	133	control	enable cell
132	control	enable cell	134	control	enable cell

23.2.5 TAP Controller

The TAP (Test Access Port) controller is a 16-state synchronous finite state machine that controls the sequence of test logic operations. The TAP can be controlled via a bus master. The bus master can be either automatic test equipment or a component (i.e., PLD) that interfaces to the TAP. The TAP controller changes state only in response to a rising edge of TCK. The value of the test mode state (TMS) input signal at a rising edge of TCK controls the sequence of state changes. The TAP controller is initialized after power-up by applying a low to the TRST# pin. In addition, the TAP controller can be initialized by applying a high signal level on the TMS input for a minimum of five TCK periods. See Figure 23-2 for the state diagram of the TAP controller. An uninitialized TAP controller can result in erratic processor behavior even when there is no intention to use the JTAG portion of the processor.

The behavior of the TAP controller and other test logic in each controller state is described in the following subsections. For greater detail on the state machine and the public instructions, refer to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture* document (available from the IEEE).

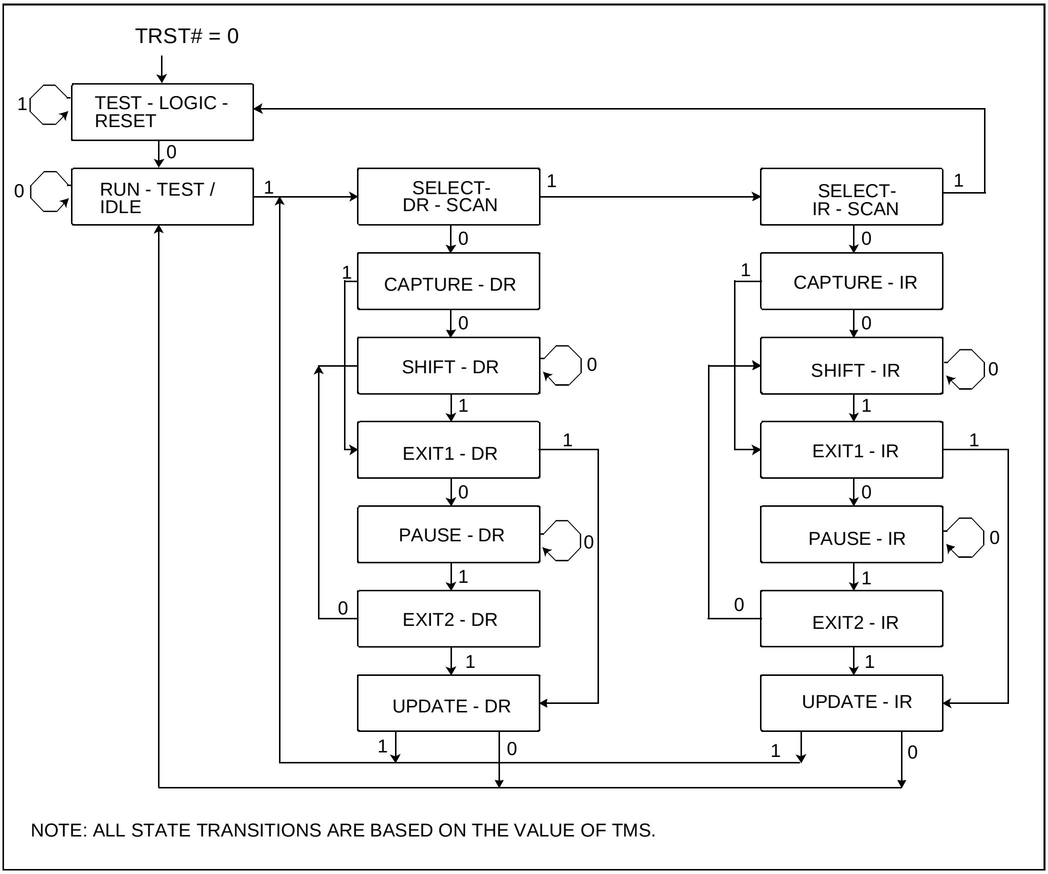


Figure 23-2. TAP Controller State Diagram

23.2.5.1 Test Logic Reset State

In this state, test logic is disabled to allow normal operation of the i960 Rx I/O processor. Upon entering the Test_Logic_Reset state, the device identification register is loaded. No matter what the present state of the controller, it enters Test-Logic-Reset state when the TMS input is held high (1_2) for at least five rising edges of TCK. The controller remains in this state while TMS is high. The TAP controller is also forced to enter this state asynchronously by asserting TRST#.

When the controller exits the Test-Logic-Reset controller state as a result of an erroneous low signal on the TMS line at the time of a rising edge on TCK (for example, a glitch due to external interference), it returns to the Test-Logic-Reset state following three rising edges of TCK with the TMS line at the intended high logic level.

23.2.5.2 Run-Test/Idle State

The TAP controller enters the Run-Test/Idle state between scan operations. The controller remains in this state as long as TMS is held low. When the **runbist** instruction is selected, it executes during the Run-Test/Idle state and the result is reported in the RUNBIST register. Instructions that do not call functions generate no activity in the test logic while the controller is in this state. The instruction register and all test data registers retain their current state. When TMS is high on the rising edge of TCK, the controller moves to the Select-DR-Scan state. The instruction register does not change while the TAP controller is in this state.

23.2.5.3 Select-DR-Scan State

The Select-DR-Scan state is a transitional controller state. While in the Select-DR-Scan state, the test data registers selected by the current instruction retain their previous states. When TMS is held low on the rising edge of TCK, the controller moves into the Capture-DR state. When TMS is held high on the rising edge of TCK, the controller moves into the Select-IR-Scan state. See Section 23.2.5.10, Select-IR Scan State (pg. 23-17). The instruction register does not change while the TAP controller is in this state.

23.2.5.4 Capture-DR State

In this state, the selected test data register is loaded with its parallel value on the rising edge of TCK. When the controller is in the Capture-DR state and the current instruction is **sample**/**preload**, the boundary-scan register captures input pin data on the rising edge of TCK. Test data registers that do not have a parallel input are not changed. The boundary-scan registers cannot be updated from the parallel inputs any other way. The instruction register does not change while the TAP controller is in this state.

When TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. When TMS is low on the rising edge of TCK, the controller enters the Shift-DR state.

23.2.5.5 Shift-DR State

In the Shift-DR state, the test data register selected by the current instruction shifts data one bit position nearer to the TDO serial output on each rising edge of TCK. All other test data registers retain their previous values during this state.

The instruction register does not change while the TAP controller is in this state.

When TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. When TMS is low on the rising edge of TCK, the controller remains in the Shift-DR state.

23.2.5.6 Exit1-DR State

Exit1-DR is a temporary controller state. When the TAP controller is in the Exit1-DR state and TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller enters the Pause-DR state.

The instruction register does not change while the TAP controller is in this state. All test data registers selected by the current instruction retain their previous value during this state.

23.2.5.7 Pause-DR State

The Pause-DR state allows the test controller to temporarily halt the shifting of data through the test data register in the serial path between TDI and TDO. The test data register selected by the current instruction retains its previous value during this state. The instruction register does not change in this state.

The controller remains in this state as long as TMS is low. When TMS is high on the rising edge of TCK, the controller moves to the Exit2-DR state.

23.2.5.8 Exit2-DR State

Exit2-DR is a temporary state. When TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller re-enters the Shift-DR state.

The instruction register does not change while the TAP controller is in this state. All test data registers selected by the current instruction retain their previous value during this state.

23.2.5.9 Update-DR State

The boundary-scan register is provided with a latched parallel output. This output prevents changes at the parallel output while data is shifted in response to the **extest**, **sample**/**preload** instructions. When the boundary-scan register is selected while the TAP controller is in the Update-DR state, data is latched onto the boundary-scan register's parallel output from the shift-register path on the falling edge of TCK. The data held at the latched parallel output does not change unless the controller is in this state.

While the TAP controller is in this state, all of the test data register's shift-register bit positions selected by the current instruction retain their previous values. The instruction register does not change while the TAP controller is in this state.

When the TAP controller is in this state and TMS is held high on the rising edge of TCK, the controller re-enters the Select-DR-Scan state. When TMS is held low on the rising edge of TCK, the controller re-enters the Run-Test/Idle state.

23.2.5.10 Select-IR Scan State

Select-IR is a temporary controller state. The test data registers selected by the current instruction retain their previous states. In this state, when TMS is held low on the rising edge of TCK, the controller enters the Capture-IR state and a scan sequence for the instruction register is initiated. When TMS is held high on the rising edge of TCK, the controller re-enters the Test-Logic-Reset state. The instruction register does not change in this state.

23.2.5.11 Capture-IR State

When the controller is in the Capture-IR state, the shift register contained in the instruction register appends the instruction with the fixed value 01_2 on the rising edge of TCK.

The test data register selected by the current instruction retains its previous value during this state. The instruction does not change in this state. While in this state, holding TMS high on the rising edge of TCK causes the controller to enter the Exit1-IR state. When TMS is held low on the rising edge of TCK, the controller enters the Shift-IR state.

23.2.5.12 Shift-IR State

When the controller is in this state, the shift register contained in the instruction register is connected between TDI and TDO and shifts data one bit position nearer to its serial output on each rising edge of TCK. The test data register selected by the current instruction retains its previous value during this state. The instruction register does not change.

When TMS is held high on the rising edge of TCK, the controller enters the Exit1-IR state. When TMS is held low on the rising edge of TCK, the controller remains in the Shift-IR state.

23.2.5.13 Exit1-IR State

This is a temporary state. When TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller enters the Pause-IR state.

The test data register selected by the current instruction retains its previous value during this state.

The instruction does not change and the instruction register retains its state.

23.2.5.14 Pause-IR State

The Pause-IR state allows the test controller to temporarily halt the shifting of data through the instruction register. The test data registers selected by the current instruction retain their previous values during this state. The instruction does not change and the instruction register retains its state.

The controller remains in this state as long as TMS is held low. When TMS is high on the rising edges of TCK, the controller enters the Exit2-IR state.

23.2.5.15 Exit2-IR State

This is a temporary state. When TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller re-enters the Shift-IR state.

This test data register selected by the current instruction retains its previous value during this state. The instruction does not change and the instruction register retains its state.

23.2.5.16 Update-IR State

The instruction shifted into the instruction register is latched onto the parallel output from the shift-register path on the falling edge of TCK. Once latched, the new instruction becomes the current instruction. Test data registers selected by the current instruction retain their previous values.

When TMS is held high on the rising edge of TCK, the controller re-enters the Select-DR-Scan state. When TMS is held low on the rising edge of TCK, the controller re-enters the Run-Test/Idle state.

23.2.6 Boundary-Scan Example

The following example describes two command actions. The example assumes the TAP controller starts in the Test-Logic-Reset state. The TAP controller then loads and executes a new instruction. See Figure 23-3 for an illustration of the waveforms involved in this example. The steps are:

1. Load the **sample**/**preload** instruction into the instruction register:

    1.1. Use TMS to select the Shift-IR state. While in the Shift-IR state, shift in the new instruction, least significant byte first.

    1.2. Use the Shift-IR state four times to read the least- through most-significant instruction bits into the instruction register (one does not care what old instruction is being shifted out of the TDO pin).

1.3. Enter the Update-IR state to make the instruction take effect.

2. Capture pin data and shift the data out through the TDO pin:

2.1. Use TMS to select the Select-DR-Scan state.

2.2. Transition the TAP controller to the Capture-DR state to latch pin data in the boundary-scan register cells.

2.3. Enter and stay in the Shift-DR state for 110 TCK cycles. These TDO values are compared against expected data to determine if component operation and connection are correct. Record the TDO values after each cycle. New serial data enters the boundary-scan register through the TDI pin, while old data is scanned out.

2.4. Pass through the Exit1-DR state to the Update-DR state. Here boundary-scan data to be driven out of the system output pins is latched and driven.

2.5. Transition back to the Select-DR state to begin another iteration.

This example does not use Pause states. These states allow software to pause the JTAG state machine to accommodate slow board-level data paths. The Pause states allow indefinite interruptions in the shifting while the external tester performs other tasks.

The old instruction was *abcd* in the example. The original instruction register value becomes the ID code since the example starts from the reset state. Other times it represents the previous opcode. The new instruction opcode is 0001_2 (**sample**/**preload**). All pins are captured into the serial boundary-scan register and the values are output to the TDO pin.

The TCK signal at the top of the diagram shows a continuous pulse train. In many designs, however, TCK is more irregular. In such cases, software controls TCK by writing to a port bit. Software writes the TMS and TDI signals and toggles the clock high. Typically, software drives TCK low quickly. The program monitors the TDO pin values as they are shifted out.

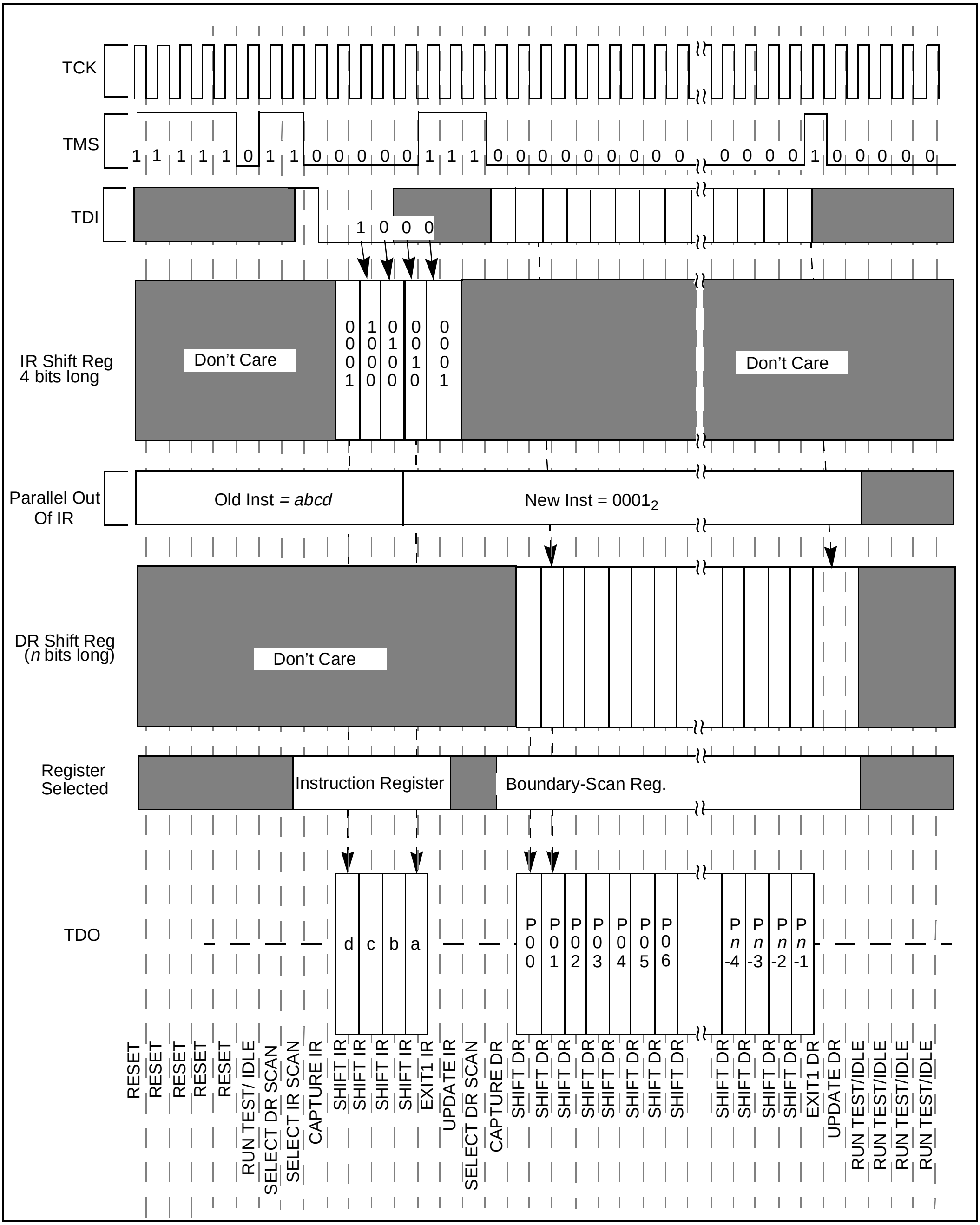


Figure 23-3. Example Showing Typical JTAG Operations

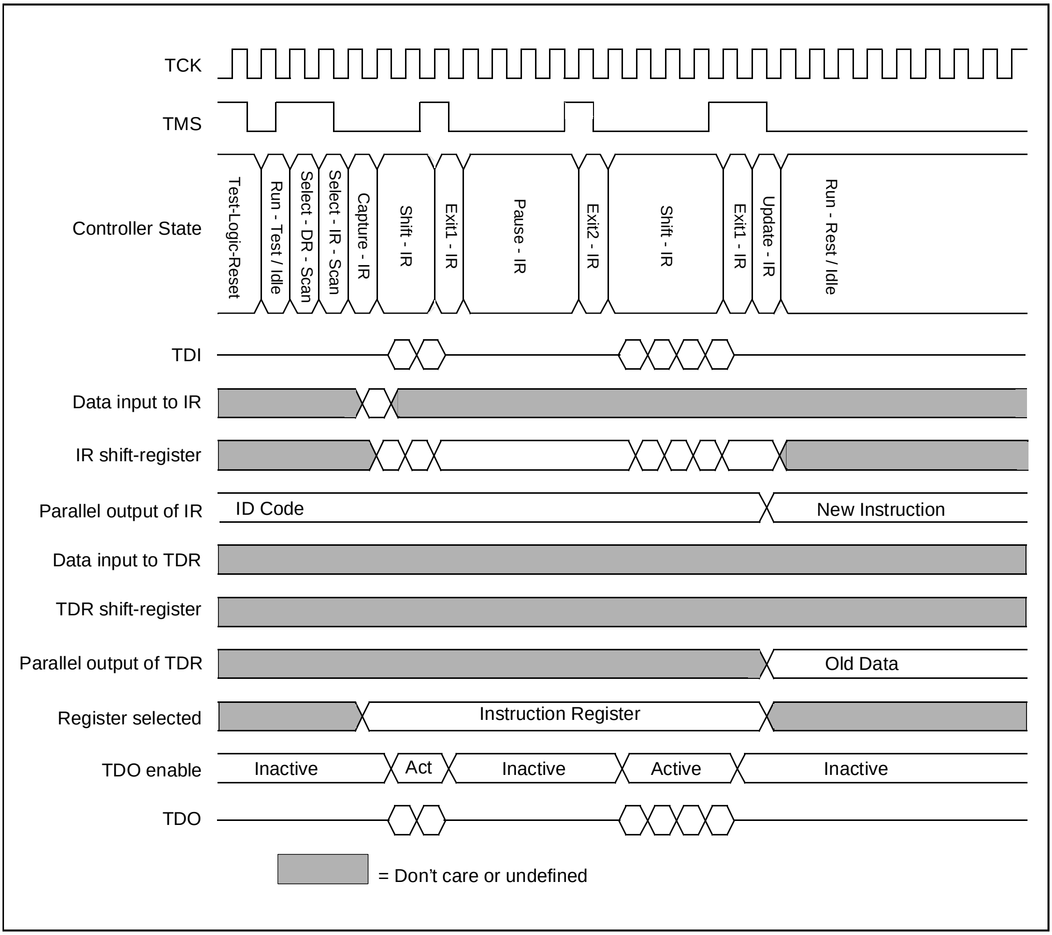


Figure 23-4. Timing Diagram Illustrating the Loading of Instruction Register

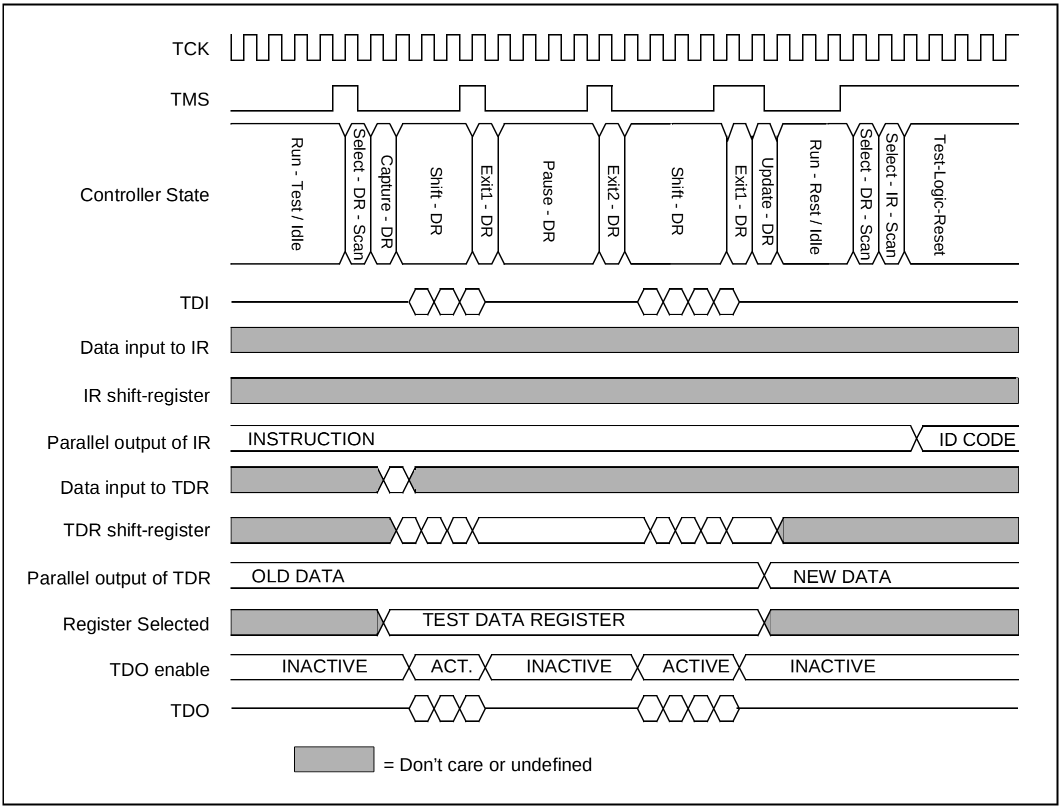


Figure 23-5. Timing Diagram Illustrating the Loading of Data Register

A

MACHINE-LEVEL INSTRUCTION FORMATS

APPENDIX A
MACHINE-LEVEL INSTRUCTION FORMATS

This appendix describes the encoding format for instructions used by the i960 processors. Included is a description of the four instruction formats and how the addressing modes relate to these formats. Refer also to APPENDIX B, OPCODES AND EXECUTION TIMES.

A.1 GENERAL INSTRUCTION FORMAT

The i960 architecture defines four basic instruction encoding formats: REG, COBR, CTRL and MEM (see Figure A-1). Each instruction uses one of these formats, which is defined by the instruction's opcode field. All instructions are one word long and begin on word boundaries. MEM format instructions are encoded in one of two sub-formats: MEMA or MEMB. MEMB supports an optional second word to hold a displacement value. The following sections describe each format's instruction word fields.

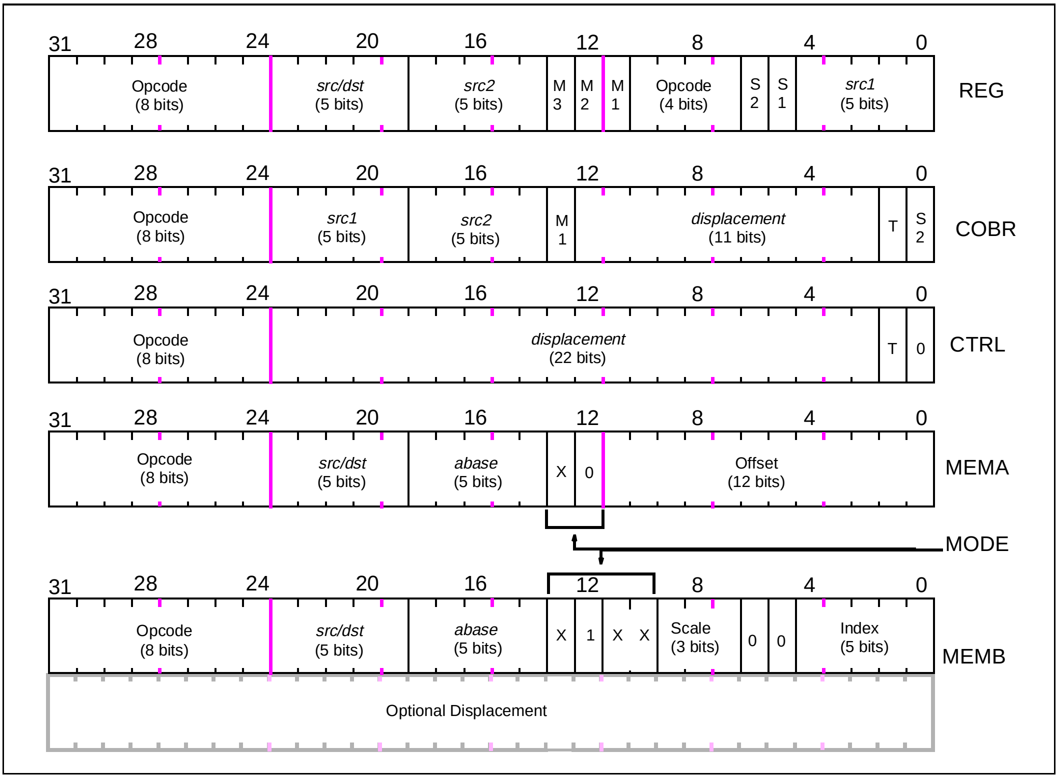


Figure A-1. Instruction Formats

Table A-1. Instruction Field Descriptions

Instruction Field	Description
Opcode	The opcode of the instruction. Opcode encodings are defined in section 6.1.8, Opcode and Instruction Format (pg. 6-5).
src1	An input to the instruction. This field specifies a value or address. In one case of the COBR format, this field is used to specify a register in which a result is stored.
src2	An input to the instruction. This field specifies a value or address.
src/dst	Depending on the instruction, this field can be (1) an input value or address, (2) the register where the result is stored, or (3) both of the above.
abase	A register whose value is used in computing a memory address.
INDEX	A register whose value is used in computing a memory address.
displacement	A signed two's complement number.
Offset	An unsigned positive number.
Optional Displacement	A signed two's complement number used in the two-word MEMB format.
MODE	A specification of how a memory address for an operand is computed and, for MEMB, specifies whether the instruction contains a second word to be used as a displacement.
SCALE	A specification of how a register's contents are multiplied for certain addressing modes (i.e., for indexing).
M1, M2, M3	These fields further define the meaning of the *src1*, *src2*, and *src/dst* fields respectively as shown in Table A-3.

When a particular instruction is defined as not using a particular field, the field is ignored.

A.2 REG FORMAT

REG format is used for operations performed on data contained in registers. Most of the i960 processor family's instructions use this format.

The opcode for the REG instructions is 12 bits long (three hexadecimal digits) and is split between bits 7 through 10 and bits 24 through 31. For example, the **addi** opcode is 591H. Here, bits 24 through 31 contain 59H and bits 7 through 10 contain 1H.

src1 and *src2* fields specify the instruction's source operands. Operands can be globa or local registers or literals. Mode bits (M1 for *src1* and M2 for *src2*) and the instruction type determine what an operand specifies. Table A-3 shows this relationship.

Table A-2. Encoding of *src1* and *src2* in REG Format

M1 or M2	Src1 or Src2 Operand Value	Register Number	Literal Value
0	00000 ... 01111	r0 ... r15	NA
	10000 ... 11111	g0 ... g15	NA
1	00000 ... 11111	NA	0 ... 31

The *src/dst* field can specify a source operand, a destination operand or both, depending on the instruction. Here again, mode bit M3 determines how this field is used. If M3 is clear, the *src/dst* operand is a global or local register that is encoded as shown in Table A-3. If M3 is set, the *src/dst* operand can be used as a source-only operand that is a literal.

When a literal is specified, it is always an unsigned 5-bit value that is zero-extended to a 32-bit value and used as the operand. When the instruction defines an operand to be larger than 32 bits, values specified by literals are zero-extended to the operand size.

Table A-3. Encoding of *src/dst* in REG Format

M3	*src/dst*	*src* Only	*dst* Only
0	g0 ... g15 r0 ... r15	g0 ... g15 r0 ... r15	g0 ... g15 r0 ... r15
1	Reserved	Reserved	reserved

A.3 COBR FORMAT

The COBR format is used primarily for compare-and-branch instructions. The test-if instructions also use the COBR format. The COBR opcode field is eight bits (two hexadecimal digits).

The *src1* and *src2* fields specify source operands for the instruction. The *src1* field can specify either a global or local register or a literal as determined by mode bit M1. The src2 field can only specify a global or local register. Table A-4 shows the M1, *src1* relationship and Table A-5 shows the S2, *src2* relationship.

Table A-4. Encoding of *src1* in COBR Format

M1	*src1*
0	g0 ... g15 r0 ... r15
1	Literal

Table A-5. Encoding of *src2* in COBR Format

S2	*src2*
0	g0 ... g15 r0 ... r15
1	reserved

The *displacement* field contains a signed two's complement number that specifies a word displacement. The processor uses this value to compute the address of a target instruction to which the processor branches as a result of the comparison. The displacement field's value can range from -2^{10} to 2^{10} -1. To determine the target instruction's IP, the processor converts the displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the current instruction.

A.4 CTRL FORMAT

The CTRL format is used for instructions that branch to a new IP, including the **BRANCH<cc>**, **bal** and **call** instructions. Note that **balx**, **bx** and **callx** do not use this format. **ret** also uses the CTRL format. The CTRL opcode field is eight bits (two hexadecimal digits).

A branch target address is specified with the displacement field in the same manner as COBR format instructions. The displacement field specifies a word displacement as a signed, two's complement number in the range -2^{21} to 2^{21}-1. The processor ignores the **ret** instruction's displacement field.

A.5 MEM FORMAT

The MEM format is used for instructions that require a memory address to be computed. These instructions include the **LOAD**, **STORE** and **lda** instructions. Also, the extended versions of the branch, branch-and-link and call instructions (**bx**, **balx** and **callx**) use this format.

The two MEM-format encodings are MEMA and MEMB. MEMB can optionally add a 32-bit displacement (contained in a second word) to the instruction. Bit 12 of the instruction's first word determines whether MEMA (clear) or MEMB (set) is used.

The opcode field is eight bits long for either encoding. The *src/dst* field specifies a global or local register. For load instructions, *src/dst* specifies the destination register for a word loaded into the processor from memory or, for operands larger than one word, the first of successive destination registers. For store instructions, this field specifies the register or group of registers that contain the source operand to be stored in memory.

The mode field determines the address mode used for the instruction. Table A-6 summarizes the addressing modes for the two MEM-format encodings. Fields used in these addressing modes are described in the following sections.

Table A-6. Addressing Modes for MEM Format Instructions

Format	MODE	Addressing Mode	Address Computation	# of Instr Words
MEMA	00	Absolute Offset	offset	1
	10	Register Indirect with Offset	(abase) + offset	1
MEMB	0100	Register Indirect	(abase)	1
	0101	IP with Displacement	(IP) + displacement + 8	2
	0110	Reserved	reserved	NA
	0111	Register Indirect with Index	(abase) + (index) * 2^{scale}	1
	1100	Absolute Displacement	displacement	2
	1101	Register Indirect with Displacement	(abase) + displacement	2
	1110	Index with Displacement	(index) * 2^{scale} + displacement	2
	1111	Register Indirect with Index and Displacement	(abase) + (index) * 2^{scale} + displacement	2

NOTES:

1. In these address computations, a field in parentheses indicates that the value in the specified register is used in the computation.
2. Usage of a reserved encoding may cause generation of an OPERATION.INVALID_OPCODE fault.

A.5.1 MEMA Format Addressing

The MEMA format provides two addressing modes:

- Absolute offset
- Register indirect with offset

The *offset* field specifies an unsigned byte offset from 0 to 4096. The abase field specifies a global or local register that contains an address in memory.

For the absolute-offset addressing mode (MODE = 00), the processor interprets the *offset* field as an offset from byte 0 of the current process address space; the abase field is ignored. Using this addressing mode along with the **lda** instruction allows a constant in the range 0 to 4096 to be loaded into a register.

For the register-indirect-with-offset addressing mode (MODE = 10), *offset* field value is added to the address in the abase register. Clearing the offset value creates a register indirect addressing mode; however, this operation can generally be carried out faster by using the MEMB version of this addressing mode.

A.5.2 MEMB Format Addressing

The MEMB format provides the following seven addressing modes:

- absolute displacement
- register indirect with displacement
- register indirect with index and displacement
- IP with displacement
- register indirect
- register indirect with displacement
- index with displacement

The abase and index fields specify local or global registers, the contents of which are used in address computation. When the index field is used in an addressing mode, the processor automatically scales the index register value by the amount specified in the SCALE field. Table A-7 gives the encoding of the scale field. The optional displacement field is contained in the word following the instruction word. The displacement is a 32-bit signed two's complement value.

Table A-7. Encoding of Scale Field

Scale	Scale Factor (Multiplier)
000	1
001	2
010	4
011	8
100	16
101 to 111	Reserved

NOTE:
Usage of a reserved encoding causes an unpredictable result.

For the IP with displacement mode, the value of the displacement field plus eight is added to the address of the current instruction.

B

OPCODES AND EXECUTION TIMES

APPENDIX B
OPCODES AND EXECUTION TIMES

B.1 INSTRUCTION REFERENCE BY OPCODE

This section lists the instruction encoding for each i960 Rx processor instruction. Instructions are grouped by instruction format and listed by opcode within each format.

Table B-1. Miscellaneous Instruction Encoding Bits

M3	M2	M1	S2	S1	T	Description
REG Format						
x	x	0	x	0	—	*src1* is a global or local register
x	x	1	x	0	—	*src1* is a literal
x	x	0	x	1	—	reserved
x	x	1	x	1	—	reserved
x	0	x	0	x	—	*src2* is a global or local register
x	1	x	0	x	—	*src2* is a literal
x	0	x	1	x	—	reserved
x	1	x	1	x	—	reserved
0	x	x	x	x	—	*src/dst* is a global or local register
1	x	x	x	x	—	*src/dst* is a literal when used as a source. M3 may not be 1 when *src/dst* is used as a destination only or is used both as a source and destination in an instruction (**atmod, modify, extract, modpc**).
COBR Format						
—	—	0	0	—	x	*src1*, *src2* and *dst* are global or local registers
—	—	1	0	—	x	*src1* is a literal, *src2* and *dst* are global or local registers
—	—	0	1	—	x	reserved
—	—	1	1	—	x0	reserved

Table B-2. REG Format Instruction Encodings (Sheet 1 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	*src/dst*	*src2*	Mode			Opcode (3-0)	Special Flags		*src1*
			31........ 24	23 ..19	18 ..14	13	12	11	10 ..7	6	5	4 0
58:0	**notbit**	1	0101 1000	*dst*	*src*	M3	M2	M1	0000	S2	S1	*bitpos*
58:1	**and**	1	0101 1000	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
58:2	**andnot**	1	0101 1000	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
58:3	**setbit**	1	0101 1000	*dst*	*src*	M3	M2	M1	0011	S2	S1	*bitpos*
58:4	**notand**	1	0101 1000	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
58:6	**xor**	1	0101 1000	*dst*	*src2*	M3	M2	M1	0110	S2	S1	*src1*
58:7	**or**	1	0101 1000	*dst*	*src2*	M3	M2	M1	0111	S2	S1	*src1*
58:8	**nor**	1	0101 1000	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
58:9	**xnor**	1	0101 1000	*dst*	*src2*	M3	M2	M1	1001	S2	S1	*src1*
58:A	**not**	1	0101 1000	*dst*		M3	M2	M1	1010	S2	S1	*src*
58:B	**ornot**	1	0101 1000	*dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*
58:C	**clrbit**	1	0101 1000	*dst*	*src*	M3	M2	M1	1100	S2	S1	*bitpos*
58:D	**notor**	1	0101 1000	*dst*	*src2*	M3	M2	M1	1101	S2	S1	*src1*
58:E	**nand**	1	0101 1000	*dst*	*src2*	M3	M2	M1	1110	S2	S1	*src1*
58:F	**alterbit**	1	0101 1000	*dst*	*src*	M3	M2	M1	1111	S2	S1	*bitpos*
59:0	**addo**	1	0101 1001	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
59:1	**addi**	1	0101 1001	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
59:2	**subo**	1	0101 1001	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
59:3	**subi**	1	0101 1001	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
59:4	**cmpob**	1	0101 1001		*src2*	M3	M2	M1	0100	S2	S1	*src1*
59:5	**cmpib**	1	0101 1001		*src2*	M3	M2	M1	0101	S2	S1	*src1*
59:6	**cmpos**	1	0101 1001		*src2*	M3	M2	M1	0110	S2	S1	*src1*
59:7	**cmpis**	1	0101 1001		*src2*	M3	M2	M1	0111	S2	S1	*src1*
59:8	**shro**	1	0101 1001	*dst*	*src*	M3	M2	M1	1000	S2	S1	*len*
59:A	**shrdi**	6	0101 1001	*dst*	*src*	M3	M2	M1	1010	S2	S1	*len*
59:B	**shri**	1	0101 1001	*dst*	*src*	M3	M2	M1	1011	S2	S1	*len*
59:C	**shlo**	1	0101 1001	*dst*	*src*	M3	M2	M1	1100	S2	S1	*len*
59:D	**rotate**	1	0101 1001	*dst*	*src*	M3	M2	M1	1101	S2	S1	*len*
59:E	**shli**	1	0101 1001	*dst*	*src*	M3	M2	M1	1110	S2	S1	*len*
5A:0	**cmpo**	1	0101 1010		*src2*	M3	M2	M1	0000	S2	S1	*src1*
5A:1	**cmpi**	1	0101 1010		*src2*	M3	M2	M1	0001	S2	S1	*src1*
5A:2	**concmpo**	1	0101 1010		*src2*	M3	M2	M1	0010	S2	S1	*src1*

1. Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 2 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	*src/dst*	*src2*	Mode			Opcode (3-0)	Special Flags		*src1*
			31 24	23 .. 19	18 . 14	13	12	11	10 ..7	6	5	4 0
5A:3	**concmpi**	1	0101 1010		*src2*	M3	M2	M1	0011	S2	S1	*src1*
5A:4	**cmpinco**	1	0101 1010	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
5A:5	**cmpinci**	1	0101 1010	*dst*	*src2*	M3	M2	M1	0101	S2	S1	*src1*
5A:6	**cmpdeco**	1	0101 1010	*dst*	*src2*	M3	M2	M1	0110	S2	S1	*src1*
5A:7	**cmpdeci**	1	0101 1010	*dst*	*src2*	M3	M2	M1	0111	S2	S1	*src1*
5A:C	**scanbyte**	1	0101 1010		*src2*	M3	M2	M1	1100	S2	S1	*src1*
5A:D	**bswap**	10	0101 1010	*dst*		M3	M2	M1	1101	S2	S1	*src1*
5A:E	**chkbit**	1	0101 1010		*src*	M3	M2	M1	1110	S2	S1	*bitpos*
5B:0	**addc**	1	0101 1011	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
5B:2	**subc**	1	0101 1011	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
5B:4	**intdis**	4	0101 1011			M3	M2	M1	0100	S2	S1	
5B:5	**inten**	4	0101 1011			M3	M2	M1	0101	S2	S1	
5C:C	**mov**	1	0101 1100	*dst*		M3	M2	M1	1100	S2	S1	*src*
5D:8	**eshro**	11	0101 1101	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
5D:C	**movl**	4	0101 1101	*dst*		M3	M2	M1	1100	S2	S1	*src*
5E:C	**movt**	5	0101 1110	*dst*		M3	M2	M1	1100	S2	S1	*src*
5F:C	**movq**	6	0101 1111	*dst*		M3	M2	M1	1100	S2	S1	*src*
61:0	**atmod**	24	0110 0010	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
61:2	**atadd**	24	0110 0010	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
64:0	**spanbit**	6	0110 0100	*dst*		M3	M2	M1	0000	S2	S1	*src*
64:1	**scanbit**	5	0110 0100	*dst*		M3	M2	M1	0001	S2	S1	*src*
64:5	**modac**	10	0110 0100	*mask*	*src*	M3	M2	M1	0101	S2	S1	*dst*
65:0	**modify**	6	0110 0101	*src/dst*	*src*	M3	M2	M1	0000	S2	S1	*mask*
65:1	**extract**	7	0110 0101	*src/dst*	*len*	M3	M2	M1	0001	S2	S1	*bitpos*
65:4	**modtc**	10	0110 0101	*mask*	*src*	M3	M2	M1	0100	S2	S1	*dst*
65:5	**modpc**	17	0110 0101	*src/dst*	*mask*	M3	M2	M1	0101	S2	S1	*src*
65:8	**intctl**	12-16	0110 0101	*dst*		M3	M2	M1	1000	S2	S1	*src1*
65:9	**sysctl**	10-100[1]	0110 0101	*src/dst*	*src2*	M3	M2	M1	1001	S2	S1	*src1*
65:B	**icctl**	10-100[1]	0110 0101	*src/dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*
65:C	**dcctl**	10-100[1]	0110 0101	*src/dst*	*src2*	M3	M2	M1	1100	S2	S1	*src1*

[1] Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 3 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	*src/dst*	*src2*	Mode			Opcode (3-0)	Special Flags		*src1*
			31........ 24	23 ..19	18 ..14	13	12	11	10 ..7	6	5	4 0
65:D	**halt**	∞	0110 0101			M3	M2	M1	1101	S2	S1	*src1*
66:0	**calls**	30	0110 0110			M3	M2	M1	0000	S2	S1	*src*
66:B	**mark**	8	0110 0110			M3	M2	M1	1011	S2	S1	
66:C	**fmark**	8	0110 0110			M3	M2	M1	1100	S2	S1	
66:D	**flushreg**	15	0110 0110			M3	M2	M1	1101	S2	S1	
66:F	**syncf**	4	0110 0110			M3	M2	M1	1111	S2	S1	
67:0	**emul**	7	0110 0111	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
67:1	**ediv**	40	0110 0111	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
70:1	**mulo**	2-4	0111 0000	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
70:8	**remo**	40	0111 0000	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
70:B	**divo**	40	0111 0000	*dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*
74:1	**muli**	2-4	0111 0100	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
74:8	**remi**	40	0111 0100	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
74:9	**modi**	40	0111 0100	*dst*	*src2*	M3	M2	M1	1001	S2	S1	*src1*
74:B	**divi**	40	0111 0100	*dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*
78:0	**addono**	1	0111 1000	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
78:1	**addino**	1	0111 1000	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
78:2	**subono**	1	0111 1000	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
78:3	**subino**	1	0111 1000	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
78:4	**selno**	1	0111 1000	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
79:0	**addog**	1	0111 1001	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
79:1	**addig**	1	0111 1001	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
79:2	**subog**	1	0111 1001	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
79:3	**subig**	1	0111 1001	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
79:4	**selg**	1	0111 1001	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
7A:0	**addoe**	1	0111 1010	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
7A:1	**addie**	1	0111 1010	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
7A:2	**suboe**	1	0111 1010	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
7A:3	**subie**	1	0111 1010	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
7A:4	**sele**	1	0111 1010	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
7B:0	**addoge**	1	0111 1011	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
7B:1	**addige**	1	0111 1011	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
7B:2	**suboge**	1	0111 1011	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*

1. Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 4 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	*src/dst*	*src2*	Mode			Opcode (3-0)	Special Flags		*src1*
			31 24	23 .. 19	18 . 14	13	12	11	10 ..7	6	5	4 0
7B:3	**subige**	1	0111 1011	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
7B:4	**selge**	1	0111 1011	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
7C:0	**addol**	1	0111 1100	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
7C:1	**addil**	1	0111 1100	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
7C:2	**subol**	1	0111 1100	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
7C:3	**subil**	1	0111 1100	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
7C:4	**sell**	1	0111 1100	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
7D:0	**addone**	1	0111 1101	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
7D:1	**addine**	1	0111 1101	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
7D:2	**subone**	1	0111 1101	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
7D:3	**subine**	1	0111 1101	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
7D:4	**selne**	1	0111 1101	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
7E:0	**addole**	1	0111 1110	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
7E:1	**addile**	1	0111 1110	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
7E:2	**subole**	1	0111 1110	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
7E:3	**subile**	1	0111 1110	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
7E:4	**selle**	1	0111 1110	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
7F:0	**addoo**	1	0111 1111	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
7F:1	**addio**	1	0111 1111	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
7F:2	**suboo**	1	0111 1111	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
7F:3	**subio**	1	0111 1111	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
7F:4	**sello**	1	0111 1111	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*

1. Execution time based on function performed by instruction.

Table B-3. COBR Format Instruction Encodings

Opcode	Mnemonic	Cycles to Execute	Opcode	*src1*	*src2*	M	Displacement	T	S2
			31......... 24	*23 . 19*	*18... 14*	13	12.......2	1	0
20	**testno**	4	0010 0000	*dst*		M1		T	S2
21	**testg**	4	0010 0001	*dst*		M1		T	S2
22	**teste**	4	0010 0010	*dst*		M1		T	S2
23	**testge**	4	0010 0011	*dst*		M1		T	S2
24	**testl**	4	0010 0100	*dst*		M1		T	S2
25	**testne**	4	0010 0101	*dst*		M1		T	S2
26	**testle**	4	0010 0110	*dst*		M1		T	S2
27	**testo**	4	0010 0111	*dst*		M1		T	S2
30	**bbc**	2 + 1[1]	0011 0000	*bitpos*	*src*	M1	*targ*	T	S2
31	**cmpobg**	2 + 1	0011 0001	*src1*	*src2*	M1	*targ*	T	S2
32	**cmpobe**	2 + 1	0011 0010	*src1*	*src2*	M1	*targ*	T	S2
33	**cmpobge**	2 + 1	0011 0011	*src1*	*src2*	M1	*targ*	T	S2
34	**cmpobl**	2 + 1	0011 0100	*src1*	*src2*	M1	*targ*	T	S2
35	**cmpobne**	2 + 1	0011 0101	*src1*	*src2*	M1	*targ*	T	S2
36	**cmpoble**	2 + 1	0011 0110	*src1*	*src2*	M1	*targ*	T	S2
37	**bbs**	2 + 1	0011 0111	*bitpos*	*src*	M1	*targ*	T	S2
38	**cmpibno**	2 + 1	0011 1000	*src1*	*src2*	M1	*targ*	T	S2
39	**cmpibg**	2 + 1	0011 1001	*src1*	*src2*	M1	*targ*	T	S2
3A	**cmpibe**	2 + 1	0011 1010	*src1*	*src2*	M1	*targ*	T	S2
3B	**cmpibge**	2 + 1	0011 1011	*src1*	*src2*	M1	*targ*	T	S2
3C	**cmpibl**	2 + 1	0011 1100	*src1*	*src2*	M1	*targ*	T	S2
3D	**cmpibne**	2 + 1	0011 1101	*src1*	*src2*	M1	*targ*	T	S2
3E	**cmpible**	2 + 1	0011 1110	*src1*	*src2*	M1	*targ*	T	S2
3F	**cmpibo**	2 + 1	0011 1111	*src1*	*src2*	M1	*targ*	T	S2

[1] Indicates that 2 cycles are required to execute the instruction plus an additional cycle to fetch the T_A get instruction when the branch is taken.

Table B-4. CTRL Format Instruction Encodings

Opcode	Mnemonic	Cycles to Execute	Opcode	*displacement*	T	0
			31............24	23...........2	1	0
08	**b**	1 + 1[1]	0000 1000	*targ*	T	0
09	**call**	7	0000 1001	*targ*	T	0
0A	**ret**	6	0000 1010		T	0
0B	**bal**	1 + 1	0000 1011	*targ*	T	0
10	**bno**	1 + 1	0001 0000	*targ*	T	0
11	**bg**	1 + 1	0001 0001	*targ*	T	0
12	**be**	1 + 1	0001 0010	*targ*	T	0
13	**bge**	1 + 1	0001 0011	*targ*	T	0
14	**bl**	1 + 1	0001 0100	*targ*	T	0
15	**bne**	1 + 1	0001 0101	*targ*	T	0
16	**ble**	1 + 1	0001 0110	*targ*	T	0
17	**bo**	1 + 1	0001 0111	*targ*	T	0
18	**faultno**	13	0001 1000		T	0
19	**faultg**	13	0001 1001		T	0
1A	**faulte**	13	0001 1010		T	0
1B	**faultge**	13	0001 1011		T	0
1C	**faultl**	13	0001 1100		T	0
1D	**faultne**	13	0001 1101		T	0
1E	**faultle**	13	0001 1110		T	0
1F	**faulto**	13	0001 1111		T	0

[1] Indicates that 2 cycles are required to execute the instruction plus an additional cycle to fetch the target instruction when the branch is taken.

Table B-5. Cycle Counts for sysctl Operations

Operation	Cycles to Execute
Post Interrupt	20
Purge I-cache	19
Enable I-cache	20
Disable I-cache	22
Software Reset	329+bus
Load Control Register Group	26
Request Breakpoint Resource	21-22

Table B-6. Cycle Counts for icctl Operations

Operation	Cycles to Execute
Disable I-cache	18
Enable I-cache	16
Invalidate I-cache	18
Load and Lock I-cache	5193
I-cache Status Request	21
I-cache Locking Status	20

Table B-7. Cycle Counts for dcctl Operations

Operation	Cycles to Execute
Disable D-cache	18
Enable D-cache	18
Invalidate D-cache	19
Load and Lock D-cache	19
D-cache Status Request	16
Quick Invalidate D-cache	14

Table B-8. Cycle Counts for intctl Operations

Operation	Cycles to Execute
Disable Interrupts	13
Enable Interrupts	13
Interrupt Status Request	8

Table B-9. MEM Format Instruction Encodings

31 24	23 ...19	18 14	13....... 12	11 0
Opcode	***src/dst***	**ABASE**	**Mode**	**Offset**

31 24	23 ...19	18 14	13....... 12 .. 11 10	9....... 7	6 .. 5	4 0
Opcode	***src/dst***	**ABASE**	**Mode**	**Scale**	**00**	**Index**
Displacement						

Effective Address

efa =										
offset	*opcode*	*dst*		0	0	*offset*				
offset(*reg*)	*opcode*	*dst*	*reg*	1	0	*offset*				
(*reg*)	*opcode*	*dst*	*reg*	0	1	0	0		00	
disp + 8 (IP)	*opcode*	*dst*		0	1	0	1		00	
	displacement									
(*reg1*)[*reg2* * *scale*]	*opcode*	*dst*	*reg1*	0	1	1	1	*scale*	00	*reg2*
disp	*opcode*	*dst*		1	1	0	0		00	
	displacement									
disp(*reg*)	*opcode*	*dst*	*reg*	1	1	0	1		00	
	displacement									
disp[*reg* * *scale*]	*opcode*	*dst*		1	1	1	0	*scale*	00	*reg*
	displacement									
disp(*reg1*)[*reg2***scale*]	*opcode*	*dst*	*reg1*	1	1	1	1	*scale*	00	*reg2*
	displacement									

Opcode	Mnemonic	Cycles to Execute	Opcode	Mnemonic	Cycles to Execute
80	**ldob**		9A	**stl**	
82	**stob**		A0	**ldt**	
84	**bx**	4-7	A2	**stt**	
85	**balx**	5-8			
86	**callx**	9-12	B0	**ldq**	
88	**ldos**		B2	**stq**	
8A	**stos**		C0	**ldib**	
8C	**lda**		C2	**stib**	
90	**ld**		C8	**ldis**	
92	**st**		CA	**stis**	
98	**ldl**				

1. The number of cycles required to execute these instructions is based on the addressing mode used (see Table B-10).

Table B-10. Addressing Mode Performance

Mode	Assembler Syntax	Memory Format	Number of Instruction words	Cycles to Execute
Absolute Offset	exp	MEMA	1	1
Absolute Displacement	exp	MEMB	2	2
Register Indirect	(reg)	MEMB	1	1
Register Indirect with Offset	exp(reg)	MEMA	1	1
Register Indirect with Displacement	exp(reg)	MEMB	2	2
Index with Displacement	exp[reg*scale]	MEMB	2	2
Register Indirect with Index	(reg)[reg*scale]	MEMB	1	6
Register Indirect with Index + Displacement	exp(reg)[reg*scale]	MEMB	2	6
Instruction Pointer with Displacement	exp(IP)	MEMB	2	6

C

MEMORY-MAPPED REGISTERS

APPENDIX C
MEMORY-MAPPED REGISTERS

This chapter describes the memory-mapped registers for the integrated peripherals.

C.1 OVERVIEW

The Peripheral Memory-Mapped Register (PMMR) interface gives software the ability to read and modify internal control registers. Each register is accessed as a memory-mapped 32-bit register with a unique memory address. Access is accomplished through regular memory-format instructions from the i960 core processor. These memory-mapped registers are specific to the i960® Rx I/O processor only.

C.2 SUPERVISOR SPACE FAMILY REGISTERS AND TABLES

Table C-1. Access Types

Access Type	Description	
R	Read	Read (**ld** instruction) accesses are allowed.
RO	Read Only	Only Read (**ld** instruction) accesses are allowed. Write (**st** instruction) accesses are ignored.
W	Write	Write (**st** instruction) accesses allowed.
R/W	Read/Write	**ld**, **st**, and **sysctl** instructions are allowed access.
WwG	Write when Granted	Writing or Modifying (through a **st** or **sysctl** instruction) the register is only allowed when modification-rights to the register have been granted. An OPERATION.UNIMPLEMENTED fault occurs if an attempt is made to write the register before rights are granted. section 10.2.7.2, Hardware Breakpoints (pg. 10-5) for details about getting modification rights to breakpoint registers.
Sysctl-RwG	**sysctl** Read when Granted	The value of the register can only be read by executing a **sysctl** instruction issued with the modify memory-mapped register message type. Modification rights to the register must be granted first or an OPERATION.UNIMPLEMENTED fault occurs when the **sysctl** is executed. A **ld** instruction to the register returns unpredictable results.
AtMod	**atmod** update	Register can be updated quickly through the **atmod** instruction. The **atmod** ensures correct operation by performing the update of the register in an atomic manner which provides synchronization with previous and subsequent operations. This is a faster update mechanism than **sysctl** and is optimized for a few special registers.

Table C-2. Supervisor Space Register Addresses (Sheet 1 of 2)

Section	Register Name - Acronym	Page	80960 Local Bus Address
	Reserved	—	FF00 8000H to FF00 80FFH
12.5.1	Default Logical Memory Configuration Register – DLMCON	12-9	FF00 8100H
	Reserved	—	FF00 8104H
12.5.1	Logical Memory Address Registers – LMADR0:1 - 0	12-8	FF00 8108H
12.5.1	Logical Memory Mask Registers – LMMR0:1 - 0	12-9	FF00 810CH
12.5.1	Logical Memory Address Registers – LMADR0:1 - 1	12-8	FF00 8110H
12.5.1	Logical Memory Mask Registers – LMMR0:1 - 1	12-9	FF00 8114H
	Reserved	—	FF00 8118H to FF00 83FFH
10.2.7.6	Instruction Breakpoint Register – IPBx	10-10	FF00 8400H to FF00 8404H
	Reserved	—	FF00 8408H to FF00 841FH
10.2.7.5	Data Address Breakpoint Register – DABx	10-10	FF00 8420H to FF00 8424H
	Reserved	—	FF00 8428H to FF00 843FH
10.2.7.4	Breakpoint Control Register – BPCON	10-8	FF00 8440H
	Reserved	—	FF00 8444H to FF00 84FFH
8.4.4	Interrupt Pending Register – IPND	8-37	FF00 8500H
8.4.4	Interrupt Mask Register – IMSK	8-38	FF00 8504H
	Reserved	—	FF00 8508H to FF00 850FH
8.4.2	Interrupt Control Register – ICON	8-34	FF00 8510H
	Reserved	—	FF00 8514H to FF00 851FH
8.4.3	Interrupt Map Register 0 – IMAP0	8-35	FF00 8520H
8.4.3	Interrupt Map Register 1 – IMAP1	8-36	FF00 8524H
8.4.3	Interrupt Map Register 2 – IMAP2	8-36	FF00 8528H

Table C-2. Supervisor Space Register Addresses (Sheet 2 of 2)

Section	Register Name - Acronym	Page	80960 Local Bus Address
	Reserved	—	FF00 852CH through FF00 85FFH
12.2	Physical Memory Control Register 0 – PMCON0_1	12-5	FF00 8600H
	Reserved	—	FF00 8604H
12.2	Physical Memory Control Register 1 – PMCON2_3	12-4	FF00 8608H
	Reserved	—	FF00 860CH
12.2	Physical Memory Control Register 2 – PMCON4_5	12-4	FF00 8610H
	Reserved	—	FF00 8614H
12.2	Physical Memory Control Register 3 – PMCON6_7	12-4	FF00 8618H
	Reserved	—	FF00 861CH
12.2	Physical Memory Control Register 4 – PMCON8_9	12-4	FF00 8620H
	Reserved	—	FF00 8624H
12.2	Physical Memory Control Register 5 – PMCON10_11	12-4	FF00 8628H
	Reserved	—	FF00 862CH
12.2	Physical Memory Control Register 6 – PMCON12_13	12-4	FF00 8630H
	Reserved	—	FF00 8634H
11.4.1 12.2	PMCON14_15 Register Bit Description in IBR Physical Memory Control Register 7 – PMCON14_15	11-16, 12-5	FF00 8638H
	Reserved	—	FF00 863CH through FF00 86F8H
12.3.1	Bus Control Register Bit Definitions – BCON	12-6	FF00 86FCH
11.4.2	Process Control Block – PRCB	11-18	FF00 8700H
8.1.5	Interrupt Stack And Interrupt Record	8-6	FF00 8704H
3.4	Supervisor Stack Pointer	3-9	FF00 8708H
	Reserved	—	FF00 870CH
11.5	Processor Device ID Register - PDIDR	11-23	FF00 8710H
11.5	i960® Core Processor Device ID Register - DEVICEID	11-24	FF00 8710H
	Reserved	—	FF00 8714H through FFFF FFFFH

Table C-3. Timer Registers

Section	Register Name	Page	80960 Local Bus Address
	Reserved		FF00 0000H to FF00 02FFH
19.1.3	Timer Reload Register – TRRx - 0	19-7	FF00 0300H
19.1.2	Timer Count Register – TCRx - 0	19-6	FF00 0304H
19.1.1	Timer Mode Register – TMRx - 0	19-3	FF00 0308H
	Reserved		FF00 030CH
19.1.3	Timer Reload Register – TRRx - 1	19-7	FF00 0310H
19.1.2	Timer Count Register – TCRx - 1	19-6	FF00 0314H
19.1.1	Timer Mode Register – TMRx - 1	19-3	FF00 0318H
	Reserved		FF00 031CH to FF00 7FFFH

C.3 PERIPHERAL MEMORY-MAPPED REGISTER ADDRESS SPACE

The PMMR address space is divided to support the integrated peripherals on the i960 Rx I/O processor. Table C-4 provides a summary of all of the PMMR registers.

They support the DMA Controller, Memory Controller, PCI And Peripheral Interrupt Controller, Messaging Unit, Local Bus Arbitration Unit, PCI-to-PCI Bridge Unit, and PCI Address Translation Unit, I^2C Bus Interface Unit, and the APIC Bus Interface Unit.

Portions of the i960 core processor address space are already reserved by the i960 core processor. Addresses 0000 0000H through 0000 03FFH are reserved for the processor internal data RAM. This memory is dedicated to the i960 core processor only and inaccessible from local bus masters. Addresses FF00 0000H through FFFF FFFFH are reserved for the processor specific memory-mapped registers. Accesses to this address space do not generate external bus cycles.

The PMMR interface provides full accessibility from the Primary ATU, Secondary ATU, and the i960 core processor. Addresses 0000 0800H through 0000 0FFFH are allocated to the PMMR interface.

Table C-4. 80960 Local Addresses Assigned to Integrated Peripherals

Integrated Peripheral	80960 Address Block
PCI-to-PCI Bridge Unit	0000 1000H through 0000 10FFH
Reserved	0000 1100H through 0000 11FFH
Address Translation Unit	0000 1200H through 0000 12FFH
Messaging Unit	0000 1300H through 0000 13FFH
DMA Controller	0000 1400H through 0000 14FFH
Memory Controller	0000 1500H through 0000 15FFH
Local Bus Arbitration Unit	0000 1600H through 0000 167FH
I^2C Bus Interface Unit	0000 1680H through 0000 16FFH
PCI And Peripheral Interrupt Controller	0000 1700H through 0000 177FH
APIC Bus Interface Unit	0000 1780H through 0000 17FFH

Table C-5 shows all i960 Rx I/O processor integrated peripheral memory-mapped registers and their 80960 local bus addresses.

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 1 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
15.13.1	Vendor ID Register - VIDR	15-41	16	0000 1000H	00H
15.13.2	Device ID Register - DIDR	15-42	16	0000 1002H	02H
15.13.3	Primary Command Register - PCMDR	15-42	16	0000 1004H	04H
15.13.4	Primary Status Register - PSR	15-44	16	0000 1006H	06H
15.13.5	Revision ID Register - RIDR	15-46	8	0000 1008H	08H
15.13.6	Class Code Register - CCR	15-46	24	0000 1009H	09H
15.13.7	Cacheline Size Register - CLSR	15-47	8	0000 100CH	0CH
15.13.8	Primary Latency Timer Register - PLTR	15-48	8	0000 100DH	0DH
15.13.9	Header Type Register - HTR	15-49	8	0000 100EH	0EH
	Reserved			0000 100FH through 0000 1017H	0FH through 17H
15.13.10	Primary Bus Number Register - PBNR	15-50	8	0000 1018H	18H
15.13.11	Secondary Bus Number Register - SBNR	15-50	8	0000 1019H	19H
15.13.12	Subordinate Bus Number Register - SubBNR	15-51	8	0000 101AH	1AH
15.13.13	Secondary Latency Timer Register - SLTR	15-52	8	0000 101BH	1BH
15.13.14	I/O Base Register - IOBR	15-53	8	0000 101CH	1CH
15.13.15	I/O Limit Register - IOLR	15-54	8	0000 101DH	1DH
15.13.16	Secondary Status Register - SSR	15-54	16	0000 101EH	1EH
15.13.17	Memory Base Register - MBR	15-56	16	0000 1020H	20H
15.13.18	Memory Limit Register - MLR	15-57	16	0000 1022H	22H
15.13.19	Prefetchable Memory Base Register - PMBR	15-58	16	0000 1024H	24H
15.13.20	Prefetchable Memory Limit Register - PMLR	15-59	16	0000 1026H	26H
	Reserved			0000 1028H through 0000 1033H	28H through 33H
15.13.21	Bridge Subsystem Vendor ID Register - BSVIR	15-60	16	0000 1034H	34H
15.13.22	Bridge Subsystem ID Register - BSIR	15-60	16	0000 1036H	36H
	Reserved			0000 1038H through 0000 103DH	38H through 3DH
15.13.23	Bridge Control Register - BCR	15-61	16	0000 103EH	3EH
15.13.24	Extended Bridge Control Register - EBCR	15-64	16	0000 1040H	40H

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 2 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
15.13.25	Secondary IDSEL Select Register - SISR	15-66	16	0000 1042H	42H
15.13.26	Primary Bridge Interrupt Status Register - PBISR	15-68	32	0000 1044H	44H
15.13.27	Secondary Bridge Interrupt Status Register - SBISR	15-69	32	0000 1048H	48H
15.13.28	Secondary Arbitration Control Register - SACR	18-12	32	0000 104CH	4CH
15.13.29	PCI Interrupt Routing Select Register – PIRSR (80960RP 33/5.0 Volt)	8-32	32	0000 1050H	50H
15.13.29	PCI Interrupt Routing Select Register – PIRSR (80960Rx 33/3.3 Volt)	8-33	32	0000 1050H	50H
15.13.30	Secondary I/O Base Register - SIOBR	15-70	8	0000 1054H	54H
15.13.31	Secondary I/O Limit Register - SIOLR	15-71	8	0000 1055H	55H
	Reserved			0000 1056H	56H
15.13.32	Secondary Memory Base Register - SMBR	15-72	16	0000 1058H	58H
15.13.33	Secondary Memory Limit Register - SMLR	15-73	16	0000 105AH	5AH
15.13.34	Secondary Decode Enable Register - SDER	15-74	16	0000 105CH	5CH
	Reserved			0000 105EH through 0000 11FFH	5EH through FFH
16.7.1	ATU Vendor ID Register - ATUVID	16-29	16	0000 1200H	00H
16.7.2	ATU Device ID Register - ATUDID	16-29	16	0000 1202H	02H
16.7.3	Primary ATU Command Register - PATUCMD	16-30	16	0000 1204H	04H
16.7.4	Primary ATU Status Register - PATUSR	16-31	16	0000 1206H	06H
16.7.5	ATU Revision ID Register - ATURID	16-32	8	0000 1208H	08H
16.7.6	ATU Class Code Register - ATUCCR	16-32	24	0000 1209H	09H
16.7.7	ATU Cacheline Size Register - ATUCLSR	16-33	8	0000 120CH	0CH
16.7.8	ATU Latency Timer Register - ATULT	16-33	8	0000 120DH	0DH
16.7.9	ATU Header Type Register - ATUHTR	16-34	8	0000 120EH	0EH
16.7.10	ATU BIST Register - ATUBISTR	16-35	8	0000 120FH	0FH
16.7.11	Primary Inbound ATU Base Address Register - PIABAR	16-36	32	0000 1210H	10H
	Reserved			0000 1214H through 0000 122BH	14H through 2BH
16.7.13	ATU Subsystem Vendor ID Register - ASVIR	16-40	16	0000 122CH	2CH
16.7.14	ATU Subsystem ID Register - ASIR	16-40	16	0000 122EH	2EH
16.7.15	Expansion ROM Base Address Register - ERBAR	16-41	32	0000 1230H	30H
	Reserved			0000 1234H through 0000 123BH	34H through 3BH

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 3 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
16.7.16	ATU Interrupt Line Register - ATUILR	16-42	8	0000 123CH	3CH
16.7.17	ATU Interrupt Pin Register - ATUIPR	16-43	8	0000 123DH	3DH
16.7.18	ATU Minimum Grant Register - ATUMGNT	16-44	8	0000 123EH	3EH
16.7.19	ATU Maximum Latency Register - ATUMLAT	16-45	8	0000 123FH	3FH
16.7.20	Primary Inbound ATU Limit Register - PIALR	16-46	32	0000 1240H	40H
16.7.21	Primary Inbound ATU Translate Value Register - PIATVR	16-47	32	0000 1244H	44H
16.7.22	Secondary Inbound ATU Base Address Register - SIABAR	16-48	32	0000 1248H	48H
16.7.23	Secondary Inbound ATU Limit Register - SIALR	16-49	32	0000 124CH	4CH
16.7.24	Secondary Inbound ATU Translate Value Register - SIATVR	16-50	32	0000 1250H	50H
16.7.25	Primary Outbound Memory Window Value Register - POMWVR	16-51	32	0000 1254H	54H
	Reserved			0000 1258H	58H
16.7.26	Primary Outbound I/O Window Value Register - POIOWVR	16-52	32	0000 125CH	5CH
16.7.27	Primary Outbound DAC Window Value Register - PODWVR	16-53	32	0000 1260H	60H
16.7.28	Primary Outbound Upper 64-bit DAC Register - POUDR	16-54	32	0000 1264H	64H
16.7.29	Secondary Outbound Memory Window Value Register - SOMWVR	16-55	32	0000 1268H	68H
16.7.30	Secondary Outbound I/O Window Value Register - SOIOWVR	16-56	32	0000 126CH	6CH
	Reserved			0000 1270H	70H
16.7.31	Expansion ROM Limit Register - ERLR	16-57	32	0000 1274H	74H
16.7.32	Expansion ROM Translate Value Register - ERTVR	16-58	32	0000 1278H	78H
	Reserved			0000 127CH through 0000 1287H	7CH through 877H
16.7.33	ATU Configuration Register - ATUCR	16-58	32	0000 1288H	88H
	Reserved			0000 128CH	8CH
16.7.34	Primary ATU Interrupt Status Register - PATUISR	16-61	32	0000 1290H	90H
16.7.35	Secondary ATU Interrupt Status Register - SATUISR	16-62	32	0000 1294H	94H
16.7.36	Secondary ATU Command Register - SATUCMD	16-64	16	0000 1298H	98H
16.7.37	Secondary ATU Status Register - SATUSR	16-65	16	0000 129AH	9AH
16.7.38	Secondary Outbound DAC Window Value Register - SODWVR	16-66	32	0000 129CH	9CH
16.7.39	Secondary Outbound Upper 64-bit DAC Register - SOUDR	16-67	32	0000 12A0H	A0H

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 4 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
16.7.40	Primary Outbound Configuration Cycle Address Register - POCCAR	16-68	32	0000 12A4H	A4H
16.7.41	Secondary Outbound Configuration Cycle Address Register - SOCCAR	16-69	32	0000 12A8H	A8H
16.7.42	Primary Outbound Configuration Cycle Data Port - POCCDP	16-70	32	0000 12ACH	Reserved
16.7.43	Secondary Outbound Configuration Cycle Data Port - SOCCDP	16-70	32	0000 12B0H	Reserved
	Reserved			0000 12B4H through 0000 12FFH	2DH through FFH
17.7.1	APIC Register Select Register - ARSR	17-15	32	0000 1300H	Available through Primary ATU Inbound Translation Window or must translate PCI address to the 80960 Memory-Mapped Address
	Reserved			0000 1304H	
17.7.2	APIC Window Register - AWR	17-16	32	0000 1308H	
	Reserved			0000 130CH	
17.7.3	Inbound Message Register - IMRx - 0	17-16	32	0000 1310H	
17.7.3	Inbound Message Register - IMRx - 1	17-16	32	0000 1314H	
17.7.4	Outbound Message Register - OMRx - 0	17-17	32	0000 1318H	
17.7.4	Outbound Message Register - OMRx - 1	17-17	32	0000 131CH	
17.7.5	Inbound Doorbell Register - IDR	17-18	32	0000 1320H	
17.7.6	Inbound Interrupt Status Register - IISR	17-19	32	0000 1324H	
17.7.7	Inbound Interrupt Mask Register - IIMR	17-20	32	0000 1328H	
17.7.8	Outbound Doorbell Register - ODR	17-22	32	0000 132CH	
17.7.9	Outbound Interrupt Status Register - OISR	17-23	32	0000 1330H	
17.7.10	Outbound Interrupt Mask Register - OIMR	17-24	32	0000 1334H	
	Reserved			0000 1338H through 0000 134FH	Must Translate PCI address to the 80960 Memory-Mapped Address
17.7.11	Messaging Unit Configuration Register - MUCR	17-26	32	0000 1350H	
17.7.12	Queue Base Address Register - QBAR	17-27	32	0000 1354H	
	Reserved			0000 1358H through 0000 135CH	
17.7.13	Inbound Free Head Pointer Register - IFHPR	17-28	32	0000 1360H	
17.7.14	Inbound Free Tail Pointer Register - IFTPR	17-29	32	0000 1364H	
17.7.15	Inbound Post Head Pointer Register - IPHPR	17-30	32	0000 1368H	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 5 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
17.7.16	Inbound Post Tail Pointer Register - IPTPR	17-31	32	0000 136CH	Must Translate PCI address to the 80960 Memory-Mapped Address
17.7.17	Outbound Free Head Pointer Register - OFHPR	17-32	32	0000 1370H	
17.7.18	Outbound Free Tail Pointer Register - OFTPR	17-33	32	0000 1374H	
17.7.19	Outbound Post Head Pointer Register - OPHPR	17-34	32	0000 1378H	
17.7.20	Outbound Post Tail Pointer Register - OPTPR	17-35	32	0000 137CH	
17.7.21	Index Address Register - IAR	17-36	32	0000 1380H	
	Reserved			0000 1384H through 0000 13FFH	
20.7.1	Channel Control Register - CCRx - 0	20-25	32	0000 1400H	
20.7.2	Channel Status Register - CSRx - 0	20-26	32	0000 1404H	
	Reserved			0000 1408H	
20.7.3	Descriptor Address Register - DARx - 0	20-28	32	0000 140CH	
20.7.4	Next Descriptor Address Register - NDARx - 0	20-29	32	0000 1410H	
20.7.5	PCI Address Register - PADRx - 0	20-30	32	0000 1414H	
20.7.6	PCI Upper Address Register - PUADRx - 0	20-31	32	0000 1418H	
20.7.7	80960 Local Address Register - LADRx - 0	20-32	32	0000 141CH	
20.7.8	Byte Count Register - BCRx - 0	20-33	32	0000 1420H	
20.7.9	Descriptor Control Register - DCRx - 0	20-34	32	0000 1424H	
	Reserved			0000 1428H through 0000 143FH	
20.7.1	Channel Control Register - CCRx - 1	20-25	32	0000 1440H	
20.7.2	Channel Status Register - CSRx - 1	20-26	32	0000 1444H	
	Reserved			0000 1448H	
20.7.3	Descriptor Address Register - DARx - 1	20-28	32	0000 144CH	
20.7.4	Next Descriptor Address Register - NDARx - 1	20-29	32	0000 1450H	
20.7.5	PCI Address Register - PADRx - 1	20-30	32	0000 1454H	
20.7.6	PCI Upper Address Register - PUADRx - 1	20-31	32	0000 1458H	
20.7.7	80960 Local Address Register - LADRx - 1	20-32	32	0000 145CH	
20.7.8	Byte Count Register - BCRx - 1	20-33	32	0000 1460H	
20.7.9	Descriptor Control Register - DCRx - 1	20-34	32	0000 1464H	
	Reserved			0000 1468H through 0000 147FH	
20.7.1	Channel Control Register - CCRx - 2	20-25	32	0000 1480H	
20.7.2	Channel Status Register - CSRx - 2	20-26	32	0000 1484H	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 6 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
	Reserved			0000 1488H	Must Translate PCI address to the 80960 Memory-Mapped Address
20.7.3	Descriptor Address Register - DARx - 2	20-28	32	0000 148CH	
20.7.4	Next Descriptor Address Register - NDARx - 2	20-29	32	0000 1490H	
20.7.5	PCI Address Register - PADRx - 2	20-30	32	0000 1494H	
20.7.6	PCI Upper Address Register - PUADRx - 2	20-31	32	0000 1498H	
20.7.7	80960 Local Address Register - LADRx - 2	20-32	32	0000 149CH	
20.7.8	Byte Count Register - BCRx - 2	20-33	32	0000 14A0H	
20.7.9	Descriptor Control Register - DCRx - 2	20-34	32	0000 14A4H	
	Reserved	x	x	0000 14A8H through 0000 14FFH	
14.5.1	Memory Bank Control Register – MBCR	14-8	32	0000 1500H	
14.5.2	Memory Bank Base Address Registers – MBBAR0:1 - 0	14-10	32	0000 1504H	
14.5.3.1	Memory Bank Read Wait States Register – MBRWS0:1 - 0	14-12	32	0000 1508H	
14.5.3.2	Memory Bank Write Wait States Register – MBWWS0:1 - 0	14-13	32	0000 150CH	
14.5.2	Memory Bank Base Address Registers – MBBAR0:1 - 1	14-10	32	0000 1510H	
14.5.3.1	Memory Bank Read Wait States Register – MBRWS0:1 - 1	14-12	32	0000 1514H	
14.5.3.2	Memory Bank Write Wait States Register – MBWWS0:1 - 1	14-13	32	0000 1518H	
14.6.4	DRAM Bank Control Register — DBCR	14-25	32	0000 151CH	
14.6.5	DRAM Base Address Register — DBAR	14-27	32	0000 1520H	
14.6.6	DRAM Bank Read Wait State Register — DRWS	14-29	32	0000 1524H	
14.6.7	DRAM Bank Write Wait State Register — DWWS	14-31	32	0000 1528H	
14.6.8	DRAM Refresh Interval Register — DRIR	14-33	32	0000 152CH	
14.7.1	DRAM Parity Enable Register — DPER	14-35	32	0000 1530H	
14.7.2	Bus Monitor Enable Register — BMER	14-36	32	0000 1534H	
14.7.3	Memory Error Address Register — MEAR	14-37	32	0000 1538H	
14.7.4	Local Processor Interrupt Status Register — LPISR	14-38	32	0000 153CH	
	Reserved	x	x	0000 1540H through 0000 15FFH	
18.2.1	Local Bus Arbitration Control Register – LBACR	18-5	32	0000 1600H	
18.2.6	Local Bus Arbitration Latency Count Register – LBALCR	18-8	32	0000 1604H	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 7 of 7)

Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Addr	PCI Conf Address Offset
	Reserved	x	x	0000 1608H through 0000 167FH	Must Translate PCI address to the 80960 Memory-Mapped Address
21.10.1	I^{2}C Control Register – ICR	21-19	32	0000 1680H	
21.10.2	I^{2}C Status Register – ISR	21-22	32	0000 1684H	
21.10.3	I^{2}C Slave Address Register – ISAR	21-25	32	0000 1688H	
21.10.4	I^{2}C Data Buffer Register – IDBR	21-26	32	0000 168CH	
21.10.5	I^{2}C Clock Count Register – ICCR	21-27	32	0000 1690H	
	Reserved	x	x	0000 1694H through 0000 16FFH	
8.4.7	NMI Interrupt Status Register – NISR	8-43	32	0000 1700H	
8.4.5	XINT6 Interrupt Status Register – X6ISR	8-39	32	0000 1704H	
8.4.6	XINT6 Interrupt Status Register – X6ISR	8-39	32	0000 1708H	
8.4.1	PCI Interrupt Routing Select Register – PIRSR (80960RP 33/5.0 Volt)	8-32	32	See PCI-to-PCI Bridge Configuration Space (0000 1050H)	
8.4.1	PCI Interrupt Routing Select Register – PIRSR (80960Rx 33/3.3 Volt)	8-33	32		
11.5	Processor Device ID Register - PDIDR	11-23	32	0000 1710H	
	Reserved	x	x	0000 1714H through 0000 177FH	
22.5.1	APIC ID Register – APIC ID	22-6	32	0000 1780H	
22.5.2	APIC Arbitration ID Register – APIC ArbID	22-7	32	0000 1784H	
22.5.3	EOI Vector Register – EVR	22-8	32	0000 1788H	
22.5.4	Interrupt Message Register – IMR	22-9	32	0000 178CH	
22.5.5	APIC Control/Status Register – APIC CSR	22-11	32	0000 1790H	
	Reserved	x	x	0000 1794H through 0000 17FFH	

INDEX