i960® VH Processor

Developer's Manual

October 1998

Order Number: 273173-001

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

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

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

The i960® VH Processor may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

Copyright © Intel Corporation, 1998

*Third-party brands and names are the property of their respective owners.

Contents

3 Programming Environment

4 Cache and On-Chip Data RAM

18 Bus Arbitration

19 Timers

20 DMA Controller

Figures

Tables

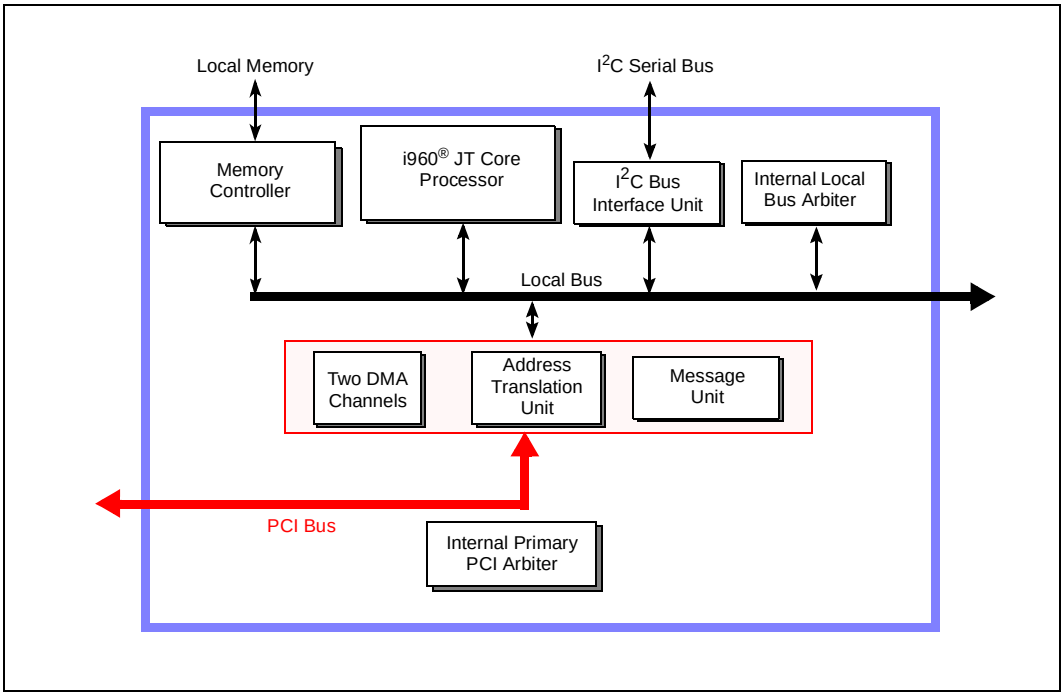
Introduction 1

1.1 Intel's i960® VH Processor

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

The PCI Bus is an industry standard, high performance, low latency system bus that operates up to 132 Mbyte/sec.

Figure 1-1. i960® VH Processor Functional Block Diagram



1.2 i960® VH Processor Features

The 80960VH combines the i960® JT processor with powerful new features to create an embedded processor. This PCI device is fully compliant with the *PCI Local Bus Specification*, revision 2.1. 80960VH-specific features include:

- DMA Controller
- Address Translation Unit
- Messaging Unit
- Memory Controller
- I2C Bus Interface Unit

Because the 80960VH's core processor is based upon the 80960JT, the two i960 family members are object code compatible and can maintain a sustained execution rate of one instruction per clock cycle. 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 80960VH 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 (80960VH)" on page 1-3 for more information.

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

1.2.1 DMA Controller

The DMA Controller allows low-latency, high-throughput data transfers between PCI bus agents and 80960 local memory. Two separate DMA channels accommodate data transfers for the primary 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.

1.2.2 Address Translation Unit

The Address Translation Unit (ATU) allows PCI transactions direct access to the 80960VH local memory. The ATU supports transactions between PCI address space and 80960VH 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.

1.2.3 Messaging Unit

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

1.2.4 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 80960VH memory. See Chapter 15, Memory Controller.

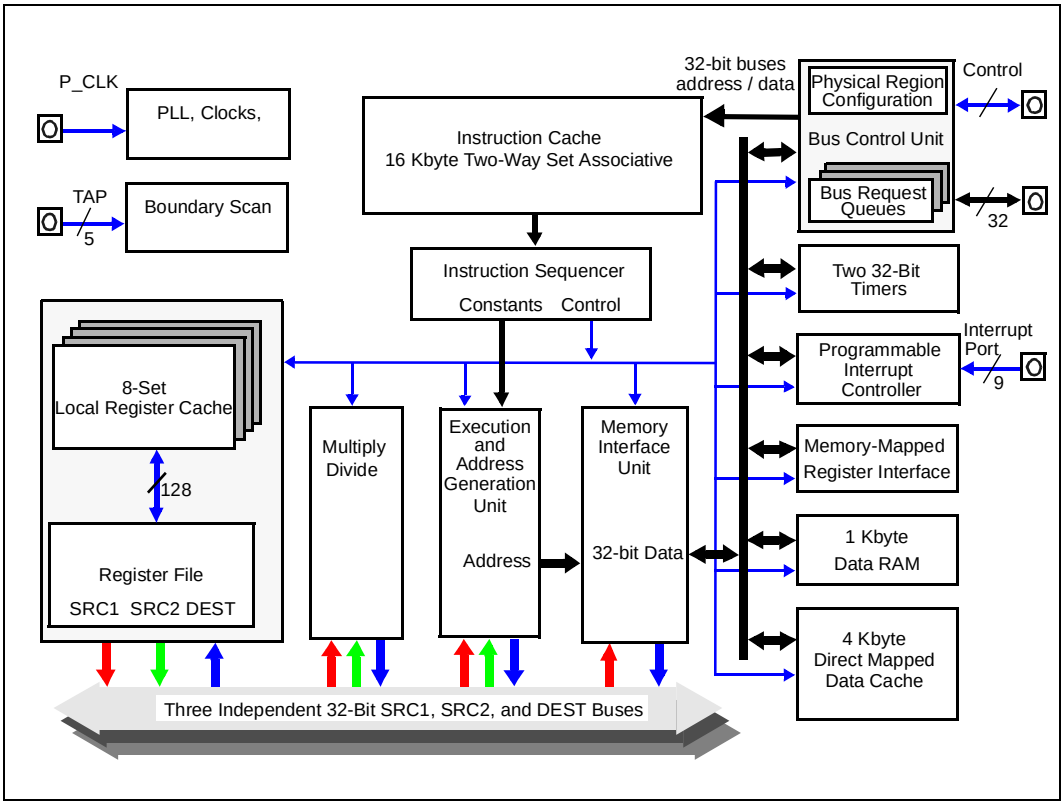
1.2.5 I²C Bus Interface Unit

The I²C (Inter-Integrated Circuit) Bus Interface Unit allows the i960 core processor to serve as a master and slave device residing on the I²C bus. The I²C unit uses a serial bus developed by Philips Semiconductor consisting of a two-pin interface. The bus allows the 80960VH to interface to other I²C 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, "I2C Bus Interface Unit". Also refer to the document *I²C Peripherals for Microcontrollers* (Philips Semiconductor).

1.3 i960® Core Processor Features (80960VH)

The processing power of the 80960VH 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 80960VH'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
- 16 Kbyte two-way set-associative, integrated instruction cache
- 4 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 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 80960VH 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 80960VH'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 80960VH 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 80960VH 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 80960VH 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 80960VH integrates considerable storage resources on-chip to decouple CPU execution from the external bus. The 80960VH includes a 16 Kbyte instruction cache, a 4 Kbyte data cache and 1 Kbyte data RAM.

1.3.6 Local Register Cache

The 80960VH 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 80960VH 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 22, “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 80960VH 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 that 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 80960VH 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 15, “Memory Controller”.

1.3.9 Instructions, Data Types and Memory Addressing Modes

As with all 80960 family processors, the 80960VH 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 80960VH instruction set, including:

- Chapter 3, Programming Environment
- Chapter 5, Instruction Set Overview
- Chapter 6, Instruction Set Reference

1.4 About This Document

The 80960VH incorporates Peripheral Component Interconnect (PCI) functionality with the 80960VH. As such, it is assumed that the reader has a working understanding of 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:

- 80960VH refers generically to the i960® VH processor.
- 80960 local bus refers to the 80960VH’s internal local bus, not the PCI local bus.
- *Primary PCI bus* is the 80960VH’s internal PCI bus that conforms to PCI SIG specifications.
- i960 core processor refers to the i960® JT processor that is integrated into the 80960VH.

- *DWORD* is a 32-bit data word.
- *80960 Local memory* is a memory subsystem on the 80960 processor local bus.
- *Downstream* — at or toward a PCI bus with a higher number (after configuration).
- *Host processor* — Processor located upstream from the i960 VH Processor.
- *Local processor* — i960 core processor within the i960 VH 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. The absence of the "#" symbol indicates that the signal's active state occurs when it is at a high voltage level.

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 Intel Customer Literature and Telephone Support

Contact Intel Corporation for literature and technical assistance for the i960® VH processor.

Country	Literature	Customer Support Number
United States	800-548-4725	800-628-8686
Canada	800-468-8118 or 303-297-7763	800-628-8686
Europe	Contact local distributor	Contact local distributor
Australia	Contact local distributor	Contact local distributor
Israel	Contact local distributor	Contact local distributor
Japan	Contact local distributor	Contact local distributor

1.4.8 Related Documents

Intel documentation is available from your Intel Sales Representative or Intel Literature Sales. See Section 1.4.7 for a complete listing of contact numbers for obtaining Intel literature.

Table 1-1. Additional Information Sources

Document Title	Order / Contact
i960® VH Processor Specification Update	Intel Order # 273174-001
i960® VH Processor at 3.3 Volts Data Sheet	Intel Order # 273179-001
i960® Jx Microprocessor Developer's Manual	Intel Order # 272483-002
MultiProcessor Specification	Intel Order # 242016
PCI Local Bus Specification, revision 2.1	PCI Special Interest Group 1-800-433-5177
PCI System Design Guide, Revision 1.0	PCI Special Interest Group 1-800-433-5177
I^2C Peripherals for Microcontrollers	Philips Semiconductor
I^2C *Bus and How to Use It* (Including Specifications)	Philips Semiconductor
I^2C *Peripherals for Microcontrollers* (Including Fast Mode)	Signetics

1.4.9 Electronic Information

Intel's documentation and other information is available from Intel's website. See Table 1-2.

Table 1-2. Electronic Information

Intel's World-Wide Web Home Page	http://www.intel.com/

Data Types and Memory Addressing Modes 2

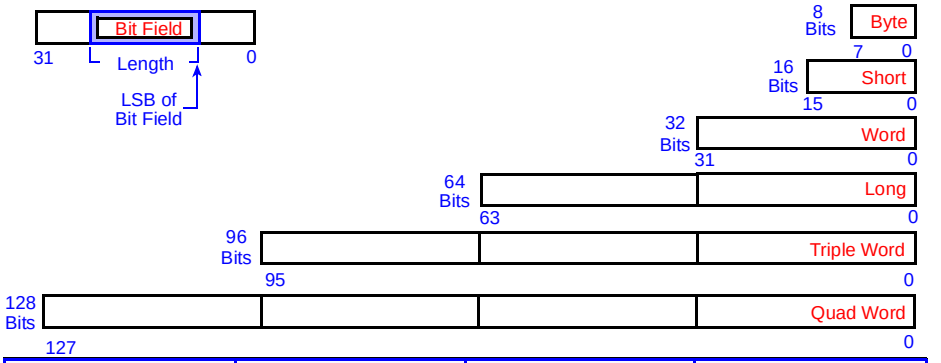
2.1 Data Types

The instruction set references or produces several data lengths and formats. The i960® VH 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.

Figure 2-1. Data Types and Ranges



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

2.1.1 Word/Dword Notation

Data lengths, as described in the *PCI Local Bus Specification* Revision 2.1, differ 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 80960VH, 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 of the load.

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

```
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
```

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 signaled. 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 (for example, g0, g2) and an integral multiple of four if three or four registers are accessed (for example, g0, g4). When a register reference for a source value is not properly aligned, the registers that the processor writes to are undefined.

The 80960VH 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, then the processor zero-extends the value to the operand size. If a literal is used in an instruction that requires integer operands, then 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" on page 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 (for example, 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".

Example 2-2. Addressing Mode Mnemonics

```
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.
stlg6,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).
ldisxyz[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 2-3. Scaled Index and Scaled Index Plus Displacement Addressing Modes

```
array_op:
      movg0,r4              # Pointer to array is copied to r4.
      subi1,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                    # Fill element at index+constant offset
      g2,0x30(r4)[r3*4]
      subi1,r3,r3           # Decrement index
.I33:
      cmpible0,r3,.I34      # Store next array elements if
      ret                   # index is not 0
```

Programming Environment 3

This chapter describes the i960® VH 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:

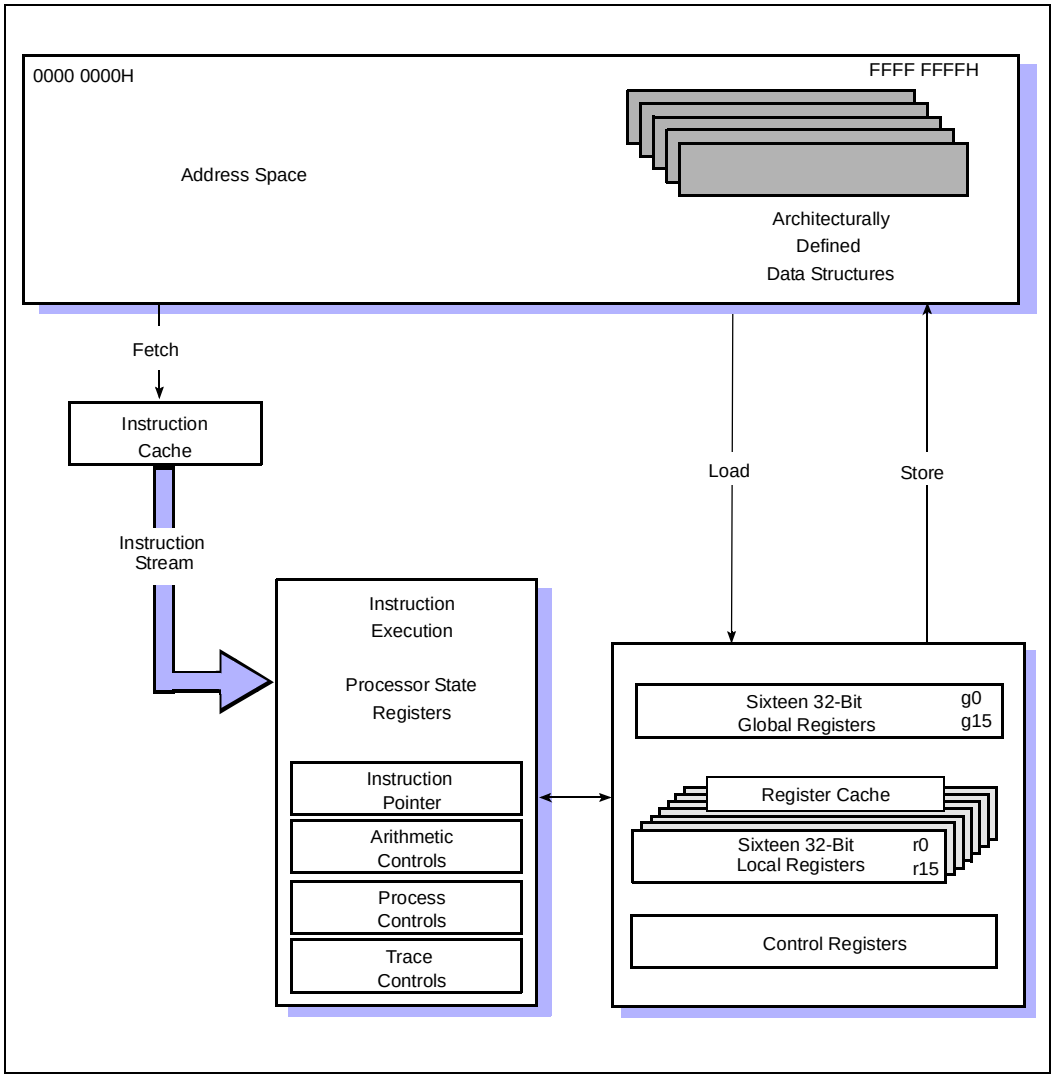
- interrupt stack
- local stack
- supervisor stack
- control table
- fault table
- interrupt table
- system procedure table
- process control block
- initialization boot record

3.2 Registers and Literals as Instruction Operands

With the exception of a few special instructions, the 80960VH 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® VH 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

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

Instruction Operand	Register Name (number)	Function	Acronym
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” on page 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 80960VH device identification and stepping information is stored in the memory-mapped register located at 0000 1710H.

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, then 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 (for example, 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
                 multiply
```

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, then the processor zero-extends the value to the operand size. If a literal is used in an instruction that requires integer operands, then 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 (for example, g0, g2) and an integral multiple of 4 if three or four registers are accessed (for example, g0, g4). If a register reference for a source value is not properly aligned, then 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, then 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

Instruction Encoding	Operand Field	Operand[1]		
		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 80960VH 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 80960VH. 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 80960VH 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 80960VH 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.

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 80960VH. 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 80960VH; 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 12, "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, then 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, then 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® VH 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 80960VH only. They support the DMA controller, memory controller, PCI and peripheral interrupt controller, messaging unit, internal arbitration unit, PCI address translation unit and I^2C 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, 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 internal bus. This interface accepts data transactions that appear on the 80960 internal bus from the Primary 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 80960 internal 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 internal 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 internal 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 internal bus (for the ATU):

- Vendor ID register
- Device ID register
- Revision ID register
- Class Code register
- Header Type 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 80960VH 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	Description
User and Supervisor Stacks Section 7.6, "User and Supervisor Stacks" on page 7-16	The processor uses these stacks when executing application code.
Interrupt Stack Section 8.1.5, "Interrupt Stack And Interrupt Record" on page 8-5	A separate interrupt stack is provided to ensure that interrupt handling does not interfere with application programs.
System Procedure Table Section 3.7, "User-Supervisor Protection Model" on page 3-17 Section 7.5, "System Calls" on page 7-13	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 Section 8.1.4, "Interrupt Table" on page 8-3	The interrupt table contains vectors (pointers) to interrupt handling procedures. When an interrupt is serviced, a particular interrupt table entry is specified.
Fault Table Section 9.3, "Fault Table" on page 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.

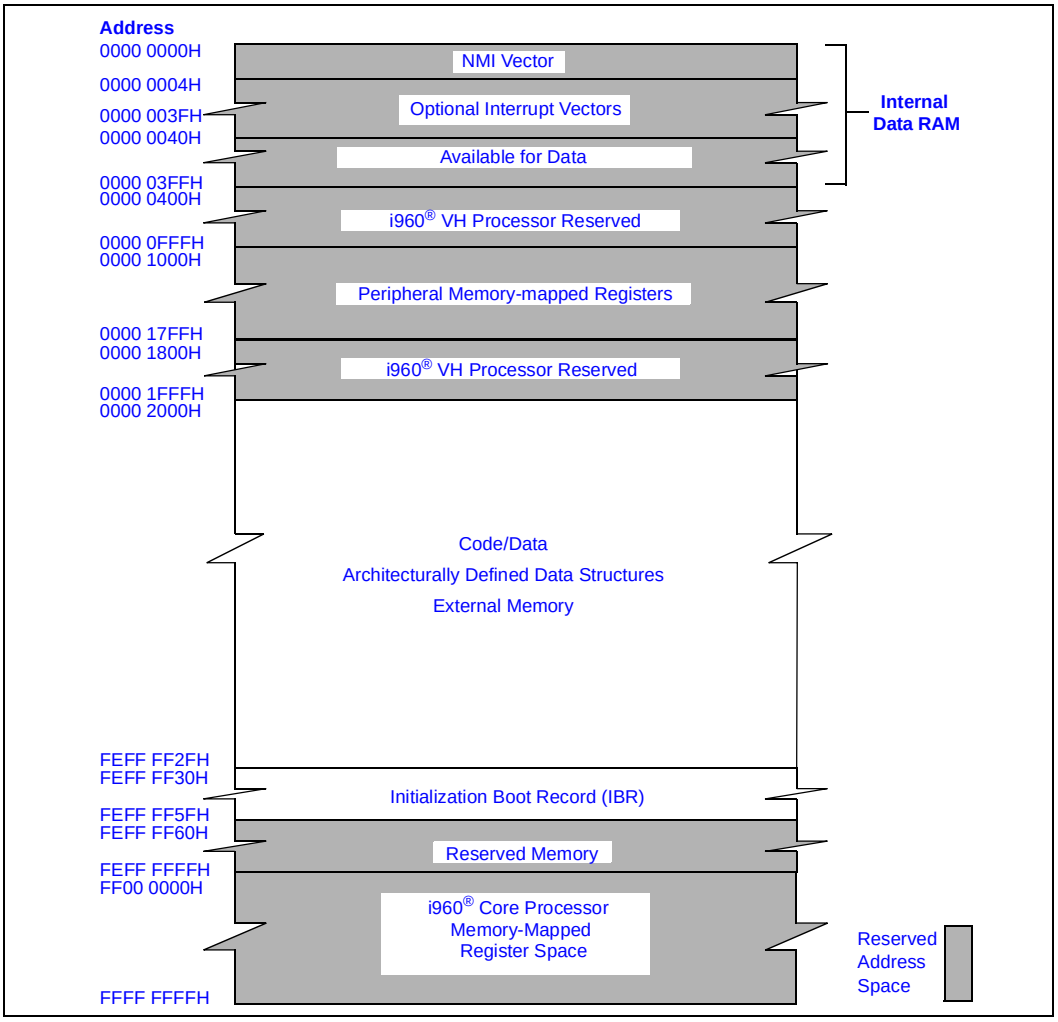
Table 3-3. Data Structure Descriptions

Structure	Description
Control Table Section 12.4.4, “Control Table” on page 12-19	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 80960VH’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 17FFH) are reserved for implementation-specific functions. 80960VH 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 80960VH's reserved memory.

The 80960VH 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" on page 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 80960VH 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 13.4.2, “Bus Transactions Across Region Boundaries” on page 13-5.
- 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 12.4.2, “Process Control Block – PRCB” on page 12-15.

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 80960VH 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 80960VH instruction cache is a 16-Kbyte, two-way set associative cache, organized in two sets of four-word lines.

3.5.6 Data Cache

The data cache on the 80960VH is a write-through 4-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.

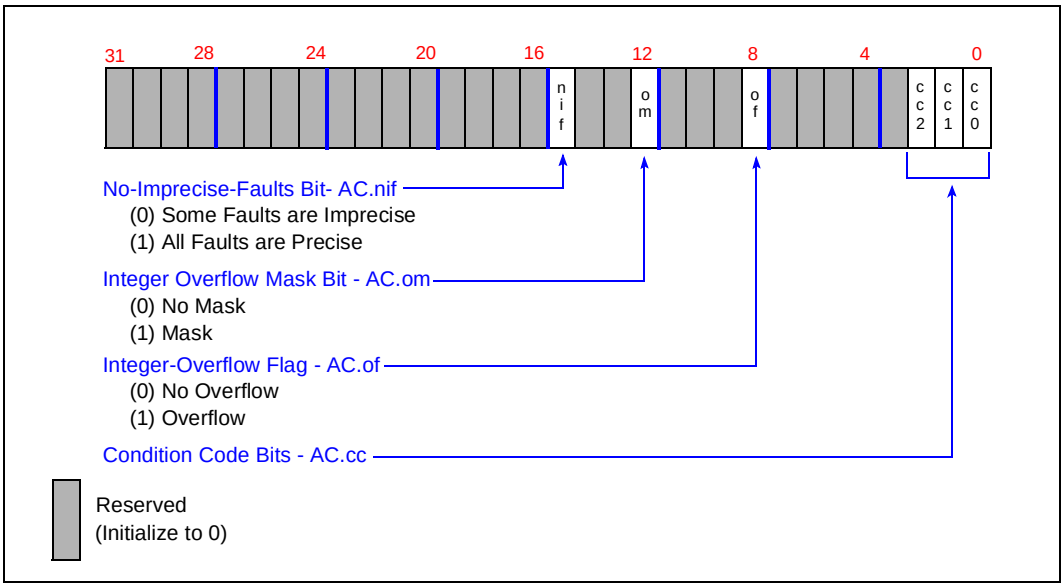
The IP register cannot be read directly. However, the IP-with-displacement addressing mode lets software use the IP as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current IP value.

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

Figure 3-3. Arithmetic Controls Register – AC



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 12, “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 3-5. To show equality and inequalities, the processor sets the condition code flags as shown in Table 3-6.

Table 3-5. Condition Codes for True or False Conditions

Condition Code	Condition
010_2	true
000_2	false

Table 3-6. 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 80960VH 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 3-7.

Table 3-7. 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, then the fault is allowed to occur and the flag is not set.

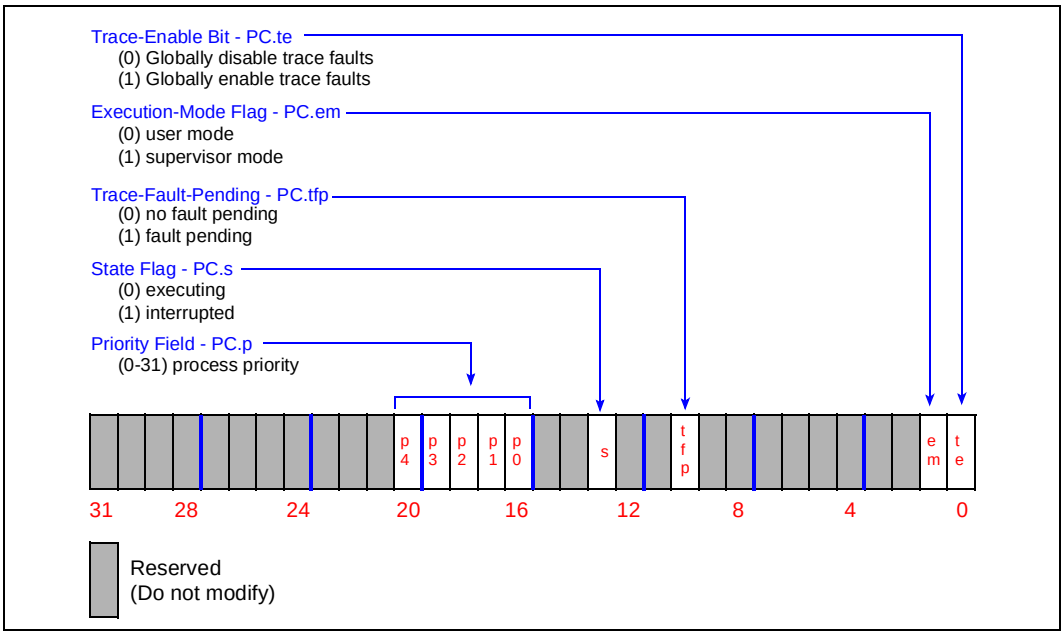
Once the processor sets this flag, the flag remains set until the application software clears it. Refer to the discussion of the ARITHMETIC.INTEGER_OVERFLOW fault in Chapter 9, “Faults” for more information about the integer overflow mask bit and flag.

The *no imprecise faults (AC.nif) bit* (bit 15) determines whether or not faults are allowed to be imprecise. If set, then all faults are required to be precise; if clear, then certain faults can be imprecise. See Section 9.9, “Precise and Imprecise Faults” on page 9-16 for more information.

3.6.3 Process Controls Register – PC

The PC register (Table 3-4) 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” on page 3-17.

Figure 3-4. 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, then 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, then 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” on page 6-72 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)
- Protected timer unit registers
- **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

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, then 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.

Cache and On-Chip Data RAM 4

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

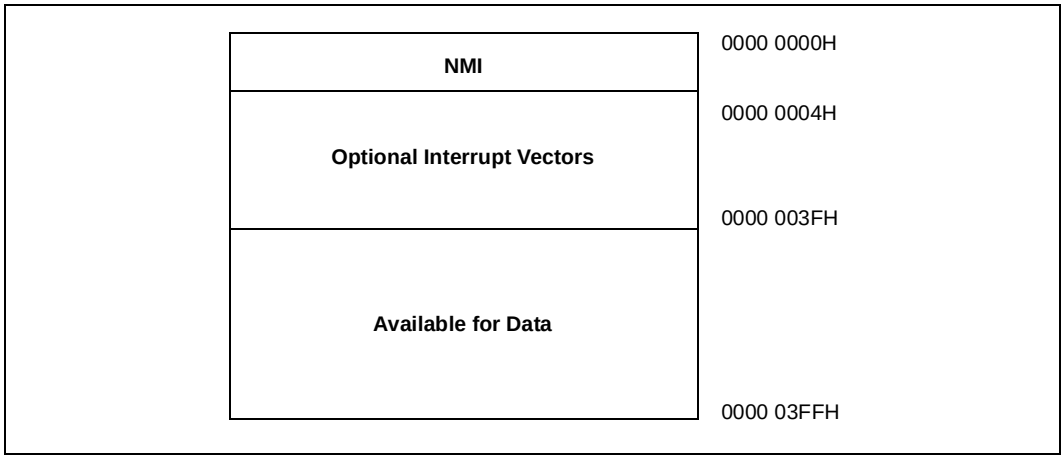
Internal data RAM is mapped to the lower 1 Kbyte (0 to 03FFH) of the address space. Loads and stores with target addresses in internal data RAM operate directly on the internal data RAM; no external bus activity is generated. Data RAM allows time-critical data storage and retrieval without dependence 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 13.3.1, “Bus Control Register – BCON” on page 13-4. 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” on page 8-35.

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 13.3.1, “Bus Control Register – BCON” on page 13-4 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® VH 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” on page 7-6 and Section 7.1.5, “Mapping Local Registers to the Procedure Stack” on page 7-10 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” on page 6-50 explains the instruction itself and Section 7.2, “Modifying the PFP Register” on page 7-10 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 that 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 results 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, then 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 80960VH features a 16-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 80960VH.

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 12-8 “Process Control Block Configuration Words” on page 12-17. 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 80960VH).
- 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 80960VH 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 80960VH 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 80960VH. Alternately, i960 processor legacy software can use **sysctl**.

4.4 Data Cache

The 80960VH features a 4-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 13.2, “Programming the Physical Memory Attributes (Pmcon Registers)” on page 13-3 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 (Table 4-1) accessed with **ldl**, **ldt**, or **ldq** miss the data cache, every word accessed by that load instruction is updated in the cache.

Table 4-1. Load Instruction Updates

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, “Core Processor Local Bus Configuration” 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 80960VH 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 80960VH 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 80960VH 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's 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 80960VH 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 80960VH 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, then 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 so that I/O regions are non-cacheable. Partitioning the system in this fashion eliminates I/O as a source of coherency problems. See Section 13.2, “Programming the Physical Memory Attributes (Pmcon Registers)” on page 13-3 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” on page 6-37 for more information.

Instruction Set Overview 5

This chapter provides an overview of the i960® microprocessor family's instruction set and i960® VH 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

80960VH 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" on page 2-6.

```
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

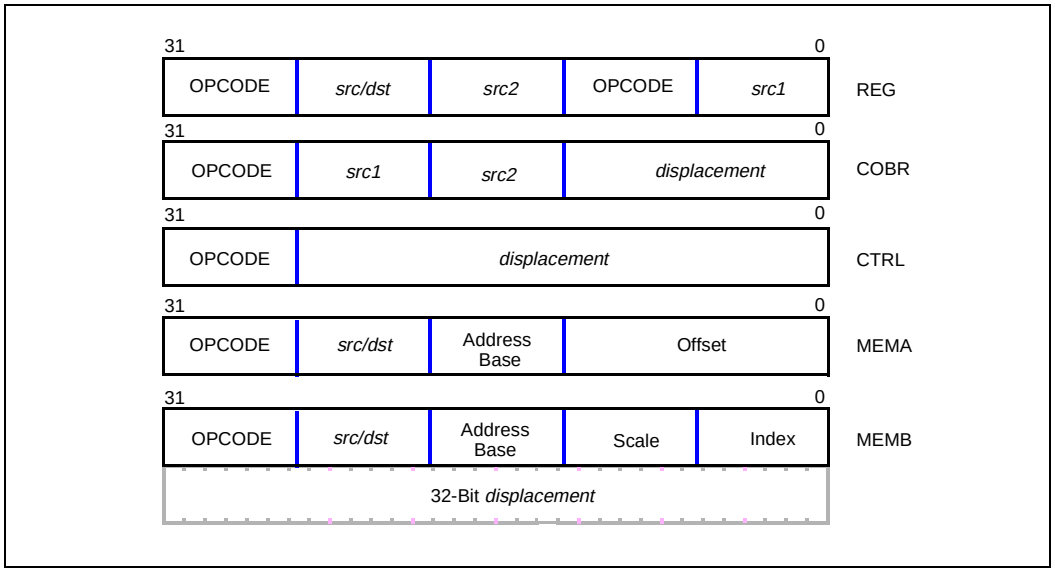
All instructions are encoded in one 32-bit machine language instruction — an *opword* — which must be word aligned in memory. An opword's most significant eight bits contain the opcode field. The opcode field determines the instruction to be performed and how the remainder of the machine

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.

Format	Operand(s)	Description
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" on page 2-4. 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 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. i960® VH Processor Instruction Set (Sheet 1 of 2)

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

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

Table 5-2. i960® VH Processor Instruction Set (Sheet 2 of 2)

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” on page 3-9 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 results 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 80960VH, **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 80960VH 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” on page 5-15 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” on page 5-10, 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 “endianness.”

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” on page 3-13 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 ($B \leq A \leq C$). Here, a compare instruction (**cmpi** or **cmpo**) checks one side of the range ($A \geq B$) and a conditional compare instruction (**concmpi** or **concmpo**) checks the other side ($A \leq C$) 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 when the bit is not set.

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" on page 7-18, 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)” on page 3-14 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” on page 5-10. 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 80960VH offers an on-chip call/return mechanism for making procedure calls. Refer to Section 7.1, "Call and Return Mechanism" on page 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 80960VH’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 80960VH 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 80960VH 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 80960VH, **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 80960VH. 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 80960VH. 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 80960VH 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.

Instruction Set Reference 6

This chapter provides detailed information about each instruction available to the i960® VH processor. Instructions are listed alphabetically by assembly language mnemonic. Format and notation used in this chapter are defined in Section 6.1, “Notation” on page 6-1.

Information in this chapter is oriented toward programmers who write assembly language code for the 80960VH. 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 A, “Machine-level Instruction Formats” - Describes instruction set opword encodings.
- Appendix B, “Opcodes and Execution Times” - A quick-reference listing of instruction encodings assists debugging with a logic analyzer.

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 (for example, **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 (for example, **BRANCH<cc>** or **FAULT<cc>**).

The 80960VH-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 (for example, **addio**, **addig**, **addoo**, **addog**).

Operand names are designed to describe operand function (for example, *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 & 010₂)==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

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

Table 6-2. Faults Applicable to All Instructions (Sheet 1 of 2)

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.

Table 6-2. Faults Applicable to All Instructions (Sheet 2 of 2)

Fault Type	Subtype	Description
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_OPERAND	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		
addole	110_2	Less or equal
addile		

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

Instruction	Mask	Condition
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” on page 6-4.
ARITHMETIC.OVERFLOW	Occurs only with **addi<cc>**.

Example:

```
# Assume (AC.cc AND 001₂) ≠ 0.
addig r4, r8, r10     # r10 = r8 + r4

# Assume (AC.cc AND 101₂) = 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, then condition code bit 1 is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, then 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] = 000₂;
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” on page 6-4.

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” on page 6-4.
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 $X1X_2$, then 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 & 010₂)==0)
    dst = src & ~(2**(bitpos%32));
else
    dst = src | 2**(bitpos%32);
```

Faults: STANDARD — Refer to Section 6.1.6, “Faults” on page 6-4.

Example:

```
# Assume AC.cc = 010₂.
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:

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” on page 6-4.

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 Section 3.5.1, "Memory Requirements" on page 3-10 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" on page 6-4.

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,*	*mask,*	*src/dst*
		reg	reg/lit	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 Section 3.5.1, "Memory Requirements" on page 3-10 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" on page 6-4.

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 Section 2.3, "Memory Addressing Modes" on page 2-4 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" on page 6-4.

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*,	*dst*
	mem	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 Section 2.3, "Memory Addressing Modes" on page 2-4 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" on page 6-4.

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*,	*src*,	*targ*
	reg/lit	reg	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, then 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" on page 6-4.

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” on page 6-4.

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,*	*src2:dst*
	reg/lit	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” on page 6-4.

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 Section 7.1.3.1, "Call Operation" on page 7-5. 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 80960VH.
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" on page 6-4.

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 Section 7.5, “System Calls” on page 7-13. *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, then 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, then 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 80960VH.
            temp.RRR = 000₂;
      }
      else            # Supervisor call from user mode.
      {     tempa = SSP;                    # Get Supervisor Stack pointer.
            temp.RRR = 010₂ | PC.te;
            PC.em = supervisor;
            PC.te = temp.te;
      }
PFP = FP;
PFP.rrr = temp.RRR;
```

```
FP = tempa;
SP = tempa + 64;
```

Faults:	STANDARD PROTECTION.LENGTH	Refer to Section 6.1.6, “Faults” on page 6-4. 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 Section 7.1.3.1, "Call Operation" on page 7-5. 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 *targ* 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 80960VH.
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" on page 6-4.

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*,	*src2*
		reg/lit	reg/lit

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

Action:

```
if (((src2 & 2**(bitpos % 32)) == 0)
    AC.cc = 000₂;
else
    AC.cc = 010₂;
```

Faults: STANDARD — Refer to Section 6.1.6, “Faults” on page 6-4.

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*,	*src*,	*dst*
		reg/lit	reg/lit	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” on page 6-4.

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 = 100₂;
else if(src1 == src2)
    AC.cc = 010₂;
else
    AC.cc = 001₂;
dst = src2 -1;      # Overflow suppressed for cmpdeci.
```

Faults: STANDARD — Refer to Section 6.1.6, “Faults” on page 6-4.

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 Compare and Increment Integer
cmpinco Compare and Increment Ordinal

Format:

cmpinc*	*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 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” on page 6-4.

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 = 100₂;
else if (src1[N:0] == src2[N:0])
    AC.cc = 010₂;
else if (src1[N:0] > src2[N:0])
    AC.cc = 001₂;
```

Faults: STANDARD Refer to Section 6.1.6, “Faults” on page 6-4.

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, then 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" on page 6-31.

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*

Instruction	Mask	Branch Condition
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 = 100₂;
else if(src1 == src2)
    AC.cc = 010₂;
else
    AC.cc = 001₂;
if((mask && AC.cc) != 000₂)
    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” on page 6-4.

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*,	*src2*
	reg/lit	reg/lit

Description: Compares *src2* and *src1* values if condition code bit 2 is not set. If comparison is performed, then 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 (for example, 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), then 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), then condition code is set to 010_2; otherwise, it is set to 001_2.

Action:

```
if (AC.cc != 1XX2)
{    if(src1 <= src2)
          AC.cc = 0102;
     else
          AC.cc  = 0012;
}
```

Faults: STANDARD Refer to Section 6.1.6, “Faults” on page 6-4.

Example:

```
cmpo g6, g3          # Compares g6 and g3
                     # and sets AC.cc.
concmpo g5, g3       # If AC.cc < 1002 (g6 Š 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*,	*src2*,	*src/dst*
	reg/lit	reg/lit	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 (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 (Figure 6-1).
Reserved	7	NA	NA
Quick invalidate	8	1	NA
Reserved	9	NA	NA

1. Invalidates data cache on 80960VH.

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

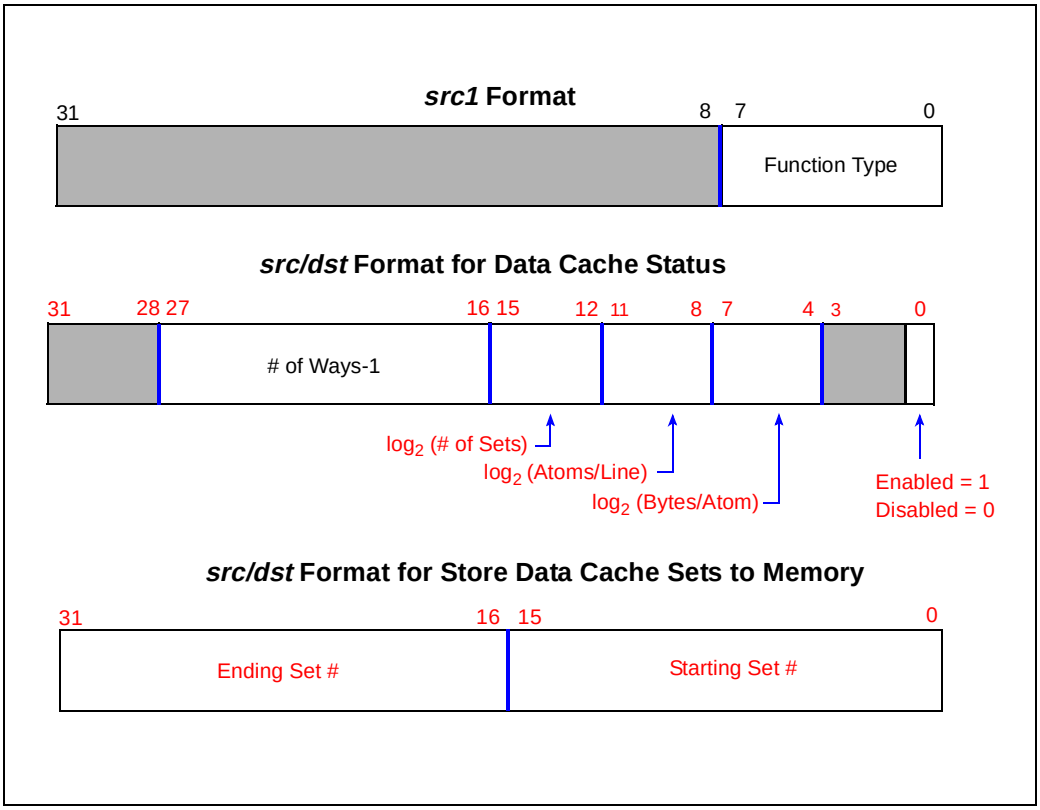


Table 6-7. dcctl Status Values and D-Cache Parameters

Value	Value on 80960VH
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

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

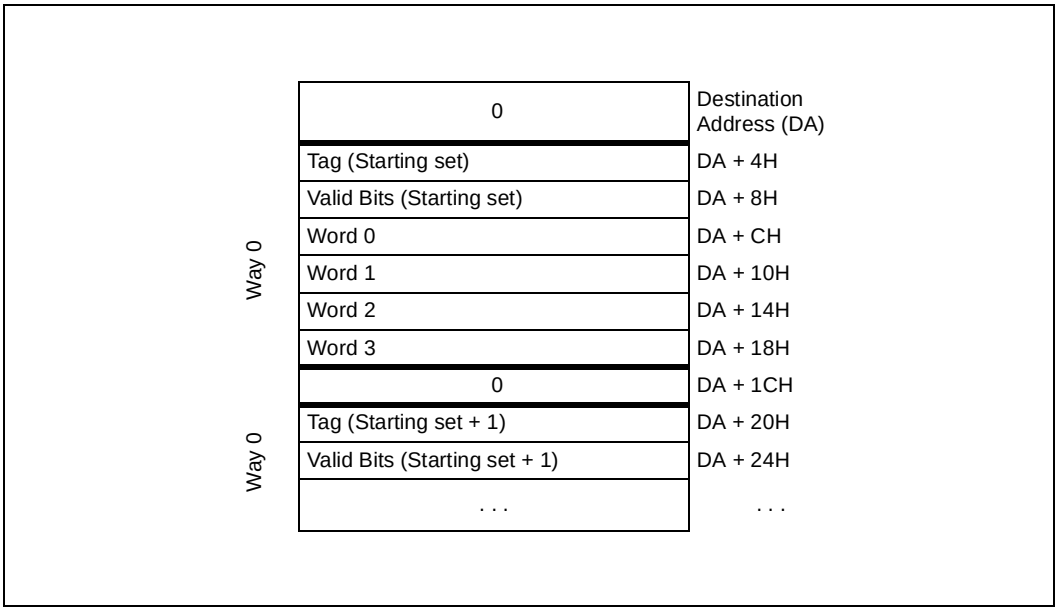
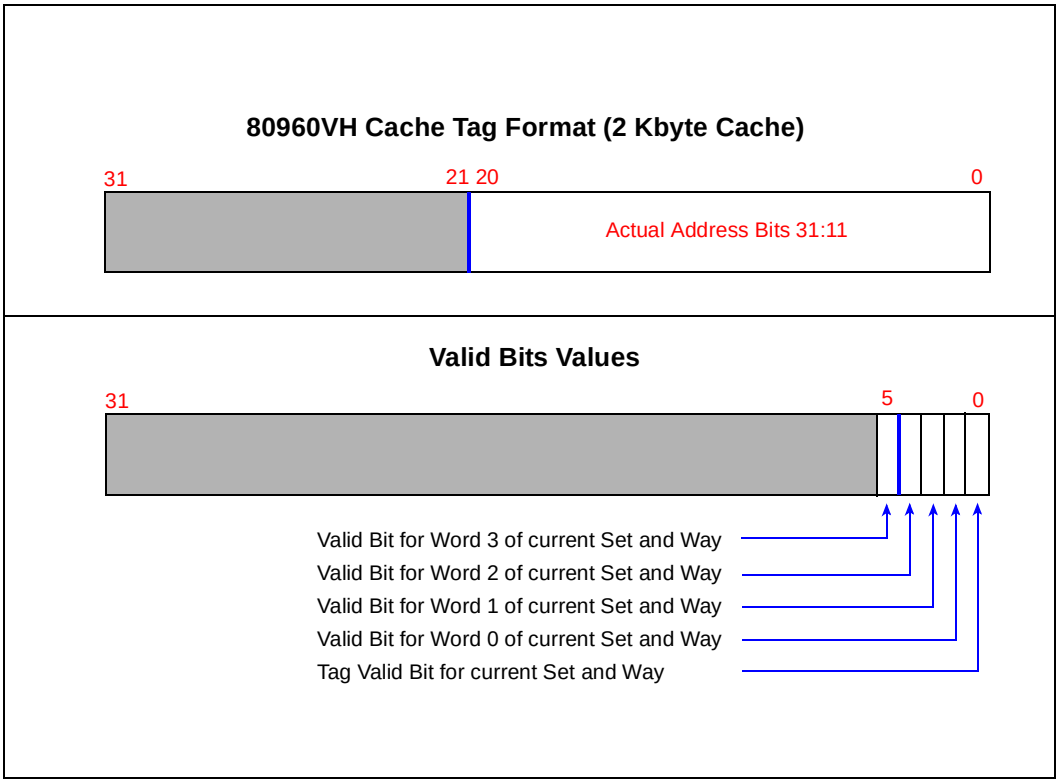


Figure 6-3. D-Cache Tag and Valid Bit Formats



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” on page 6-4.
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 is 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 80960VH, 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*,	*src2*,	*dst*
	reg/lit	reg/lit	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, “Faults” on page 6-4.
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, then 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*,	*src2*,	*dst*
	reg/lit	reg/lit	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), then 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" on page 6-4.
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*,	*src2*,	*dst*
		reg/lit	reg/lit	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" on page 6-4.

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*,	*src2*,	*dst*
	reg/lit	reg/lit	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" on page 6-4.

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” on page 6-4.

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 != 000_2)
    generate_fault(CONSTRAINT.RANGE);

faultno:

if(AC.cc == 000_2)
    generate_fault(CONSTRAINT.RANGE);

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-4.
CONSTRAINT.RANGE	If condition being tested is true.

Example:

```
# Assume (AC.cc AND 110₂)≠000₂
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, then **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" on page 6-4.

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" on page 6-4.
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*.

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.

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

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” on page 6-4.
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 80960VH 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*,	*src2*,	*src/dst*
	reg/lit	reg/lit	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 (Figure 6-4).
Get I-cache locking status	5	NA	*dst:* Receives status (Figure 6-4)
Store I-cache sets to memory	6	Destination address for cache sets	*src*: I-cache set #'s to be stored (Figure 6-4).

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

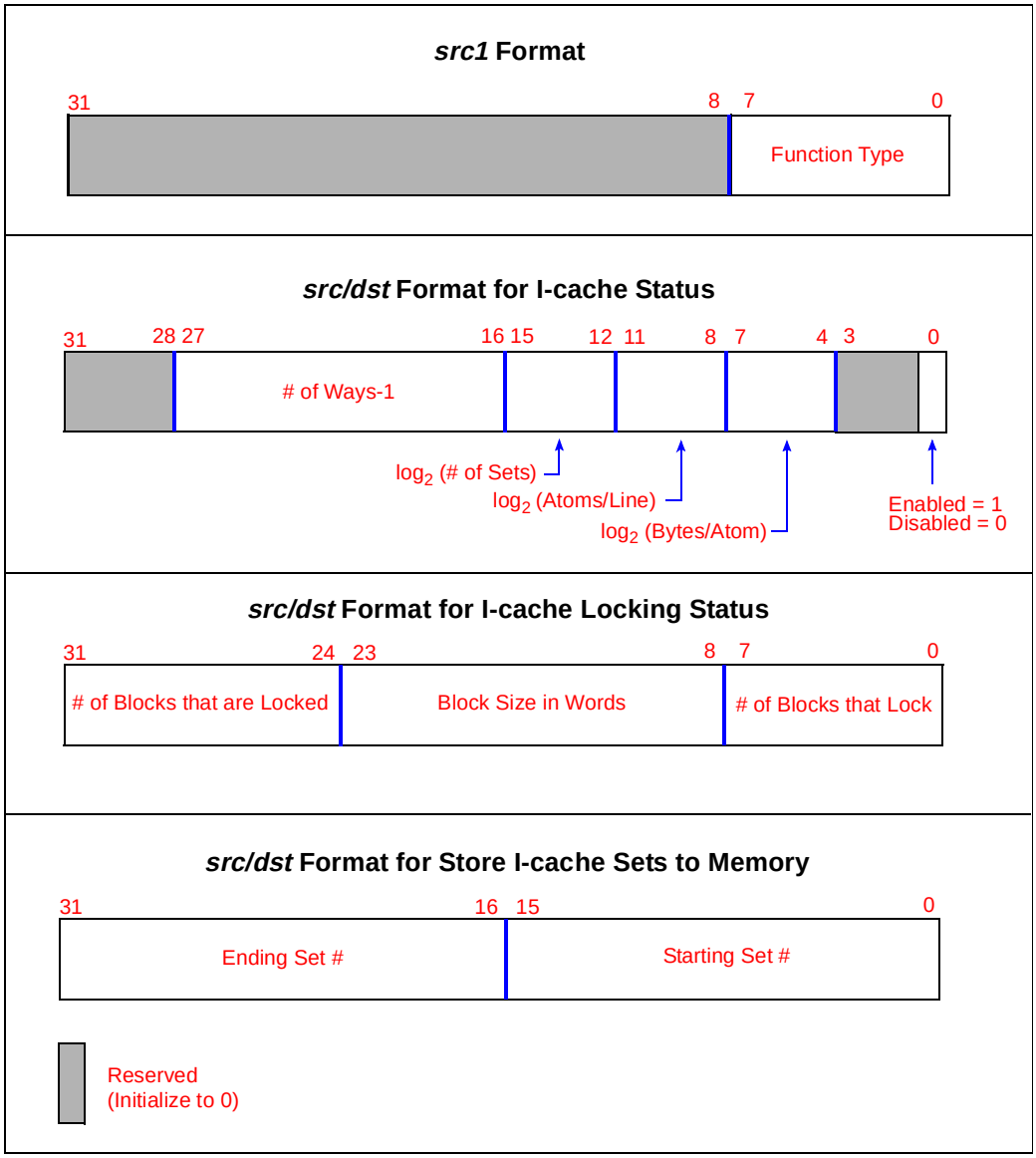


Table 6-9. icctl Status Values and I-Cache Parameters

Value	Value on i960VH 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

Figure 6-5. Store Instruction Cache to Memory Output Format

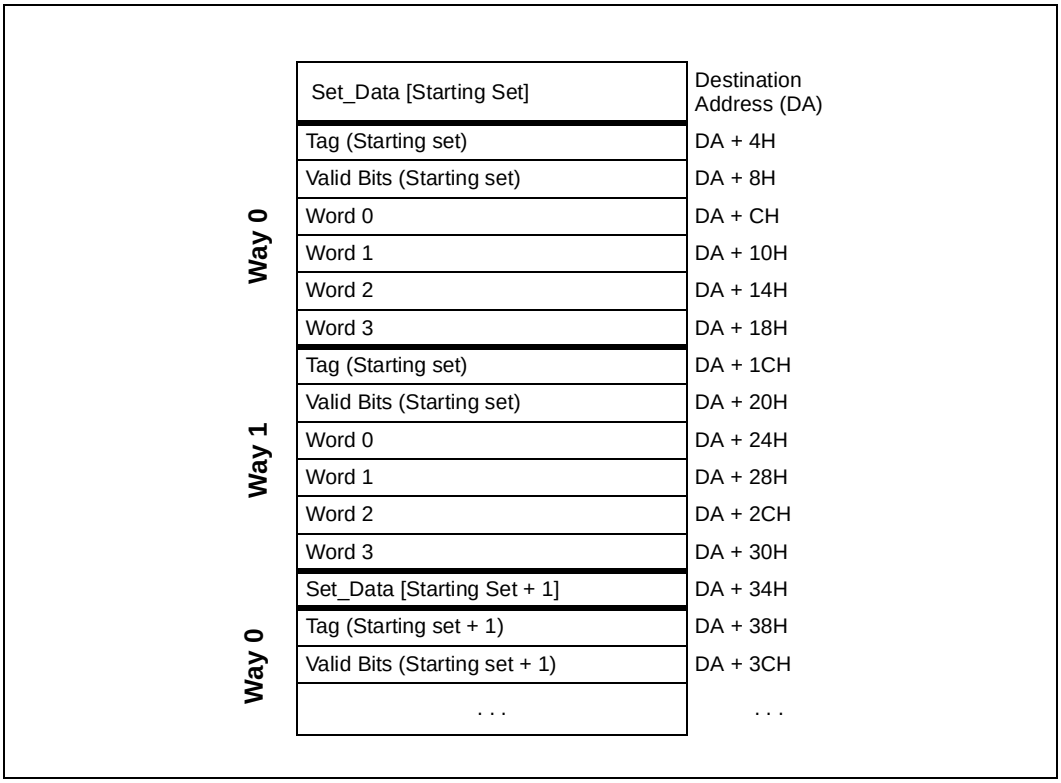
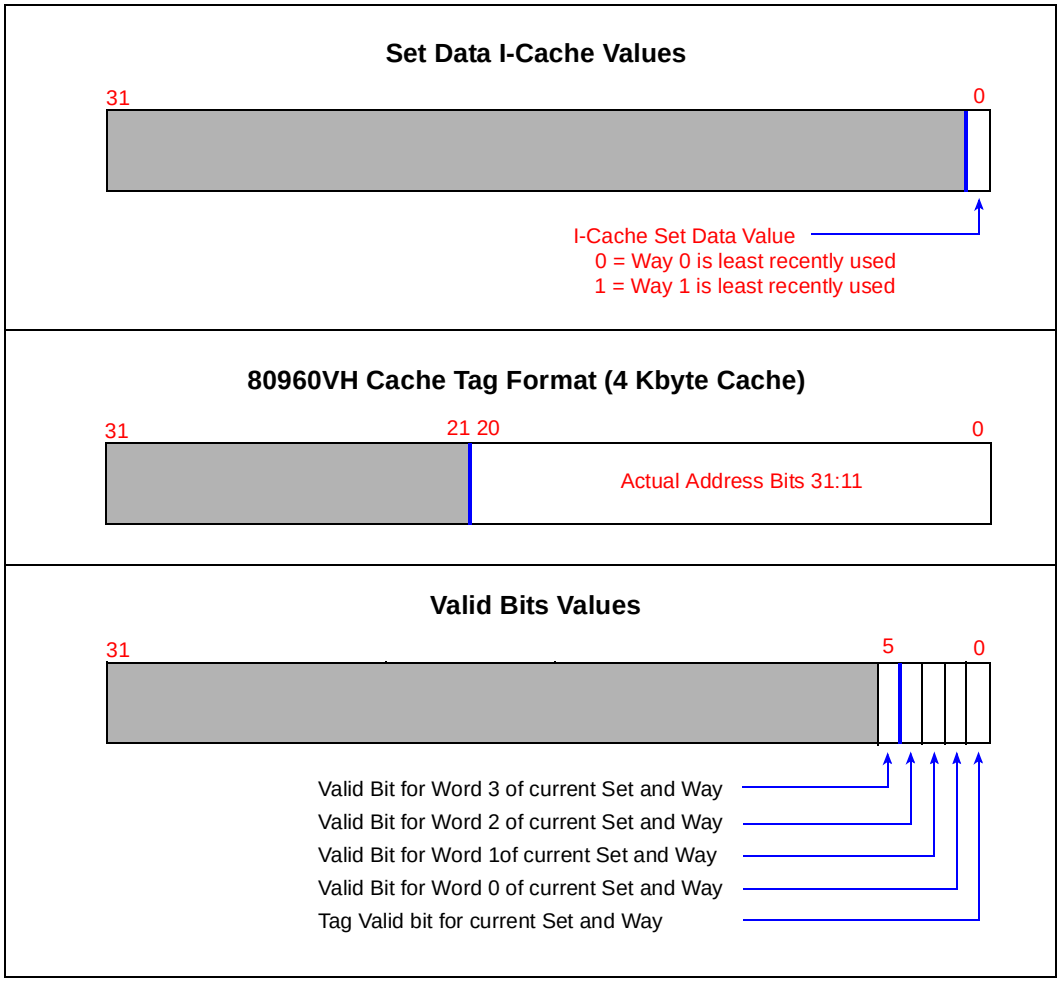


Figure 6-6. I-Cache Set Data, Tag and Valid Bit Formats



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 VH, 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” on page 6-4.
TYPE.MISMATCH		Attempt to execute instruction while not in supervisor mode.

Example:

```
icctl g0,g1,g2      # g0 = 3, g1=0x10000000, g2=1
                    # 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 80960VH, 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” on page 6-4.
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 80960VH, 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” on page 6-4.
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 80960VH, 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” on page 6-4.
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 80960VH, 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] != 00₂ ) && 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” on page 6-4.
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" on page 6-4.

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, then **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" on page 6-4.
	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*,	*src*,	*dst*
		reg/lit	reg/lit	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” on page 6-4.

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*,	*src2*,	*dst*
		reg/lit	reg/lit	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, then *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	Refer to Section 6.1.6, “Faults” on page 6-4.
	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” on page 6-4.

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*,	*mask*,	*src/dst*
	reg/lit	reg/lit	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, then 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, then 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, then the processor may not recognize the change before the next four non-branch instructions are executed. For more information see Section 3.6.3, “Process Controls Register – PC” on page 3-15.

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” on page 6-4.
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*,	*src2*,	*dst*
	reg/lit	reg/lit	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, then 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" on page 6-4.

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 (for example, g0, g2, ... or r4, r6, ...) for **movl** and an integral multiple of four (for example, 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” on page 6-4.

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” on page 6-4.
ARITHMETIC.OVERFLOW	Result is too large for destination register (**muli** only). If a condition of overflow occurs, then 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” on page 6-4.

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	*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” on page 6-4.

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” on page 6-4.

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” on page 6-4.

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” on page 6-4.

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*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
	ornot	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	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" on page 6-4.

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 Remainder Integer
remo Remainder Ordinal

Format:

rem*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	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” on page 6-4.
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 000₂:          #local return
          get_FP_and_IP();
          break;
     case 001₂:          #fault return
          tempa = memory(FP-16);
          tempb = memory(FP-12);
          get_FP_and_IP();
          AC = tempb;
          if(execution_mode == supervisor)
               PC = tempa;
          break;
     case 010₂:          #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” on page 6-4.

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*,	*src2*,	*dst*
		reg/lit	reg/lit	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” on page 6-4.

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*,	*dst*
	reg/lit	reg

Description: Searches *src1* for a set bit (1 bit). If a set bit is found, then 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, then 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" on page 6-4.

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, then 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 = 010₂;
else
    AC.cc = 000₂;
```

Faults: STANDARD Refer to Section 6.1.6, “Faults” on page 6-4.

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,*	*src2,*	*dst*
	reg/lit	reg/lit	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” on page 6-4.

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” on page 6-4.

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, then *dst* equals *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” on page 6-4.
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*,	*dst*
	reg/lit	reg

Description: Searches *src* value for the most significant clear bit (0 bit). If a most significant 0 bit is found, then its bit number is stored in *dst* and condition code is set to 010_2. If *src* value is all 1's, then 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" on page 6-4.

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 Section 2.3, “Memory Addressing Modes” on page 2-4 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, then 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 (for example, g0, g2, ... or r0, r2, ...). For **stt** and **stq**, *src* must specify a register number that is a multiple of four (for example, 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] != 00₂) && 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] != 0000₂) && 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] != 0000₂) && 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” on page 6-4.
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*,	*src2*,	*dst*
	reg/lit	reg/lit	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, then 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, then 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] = 000₂;
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” on page 6-4.

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 ARITHMETIC.OVERFLOW	Refer to Section 6.1.6, “Faults” on page 6-4. 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” on page 6-4.
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” on page 6-4.

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*,	*src2*,	*src/dst*
	reg/lit	reg/lit	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

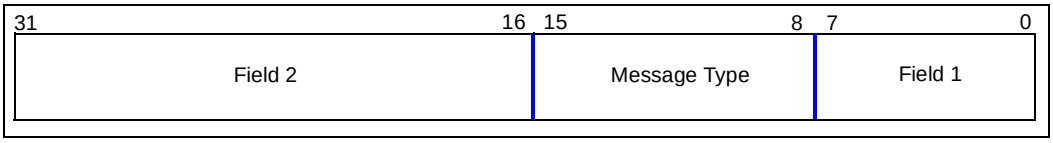


Table 6-10. sysctl Field Definitions

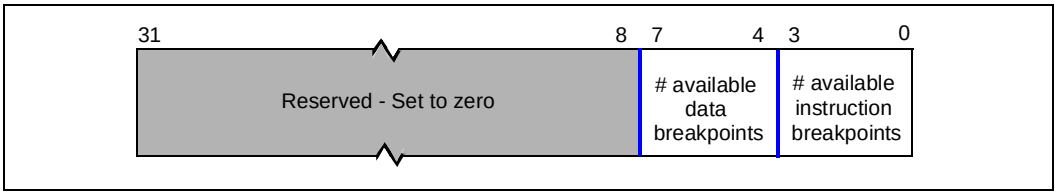
Message	*src1*			*src2*	*src/dst*
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	0x0	Vector Number	N/U	N/U	N/U
Invalidate Cache	0x1	N/U	N/U	N/U	N/U
Configure Instruction Cache	0x2	Cache Mode Configuration (Table 6-11)	N/U	Cache load address	N/U
Reinitialize	0x3	N/U	N/U	Starting IP	PRCB Pointer
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 (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	80960VH
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

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 80960VH.
                                        # 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” on page 6-4.

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 configu-
ration 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 80960VH, 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” on page 6-4.

Example:

```
# Assume AC.cc = 100₂
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” on page 6-4.

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**

Procedure Calls 7

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

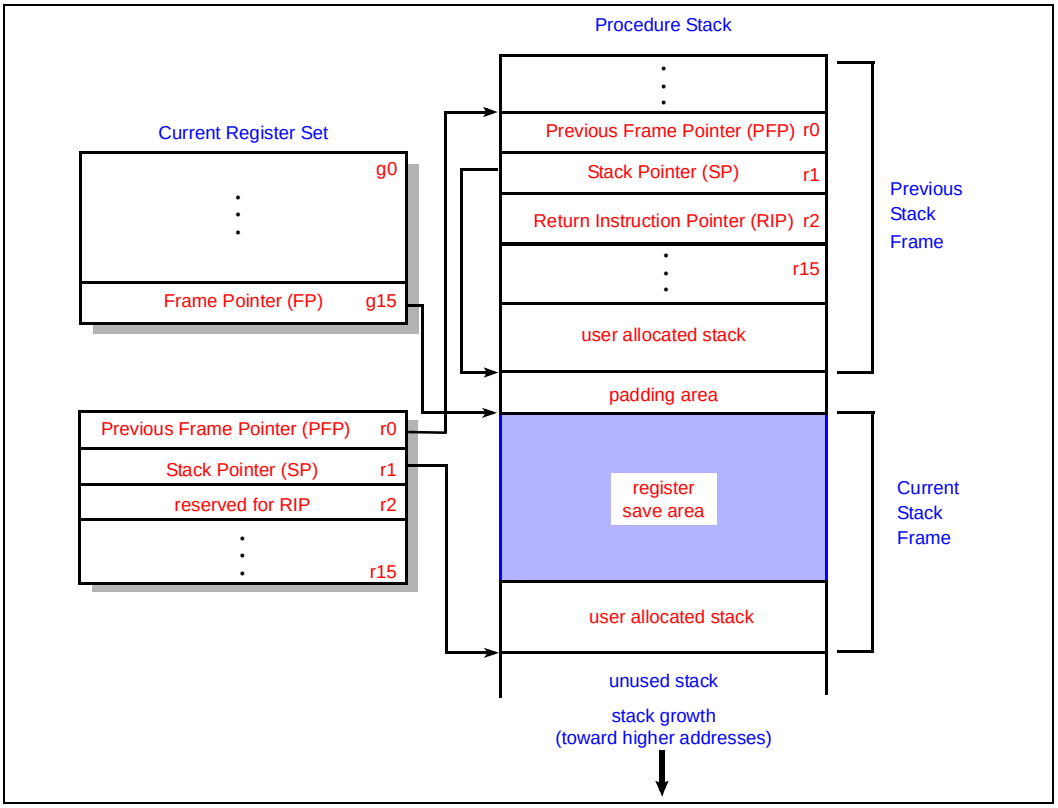
The processor automatically allocates a set of 16 local registers for each procedure. Since local registers are on-chip, they provide fast access storage for local variables. Of the 16 local registers, 13 are available for general use; r0, r1 and r2 are reserved for linkage information to tie procedures 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" on page 7-6 and Section 7.1.4.1, "Reserving Local Register Sets for High Priority Interrupts" on page 7-7 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® processor, stacks are aligned on 16-byte boundaries (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:

```
movsp,r4
addo24,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:

```
subo24,sp,r4
ld 20(r4),rn
   ...
ld (r4),rn
movr4,sp
```

7.1.2.5 Previous Frame Pointer

The previous frame pointer is the previous stack frame's first byte address. This address' 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 Figure 7-5 and Table 7-2 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" on page 7-17.

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 80960VH 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 performed. See Section 7.8, "Returns" on page 7-17.
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 is 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 Section 4.2, “Local Register Cache” on page 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 Section 6.2.30, “flushreg” on page 6-50, 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 is 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 (Table 12-8 “Process Control Block Configuration Words” on page 12-17). 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 12.4.2, “Process Control Block – PRCB” on page 12-15.

Figure 7-2. Frame Spill

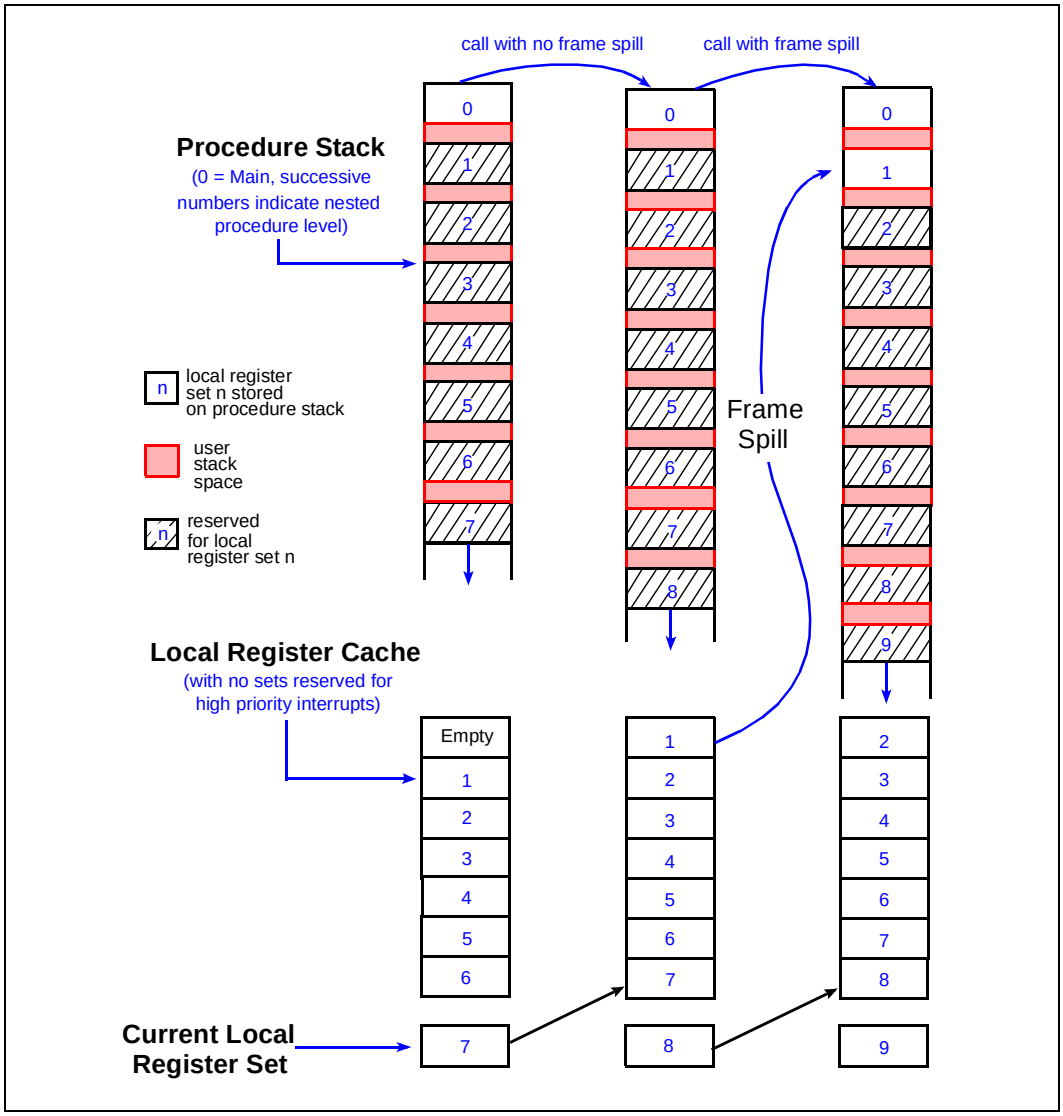
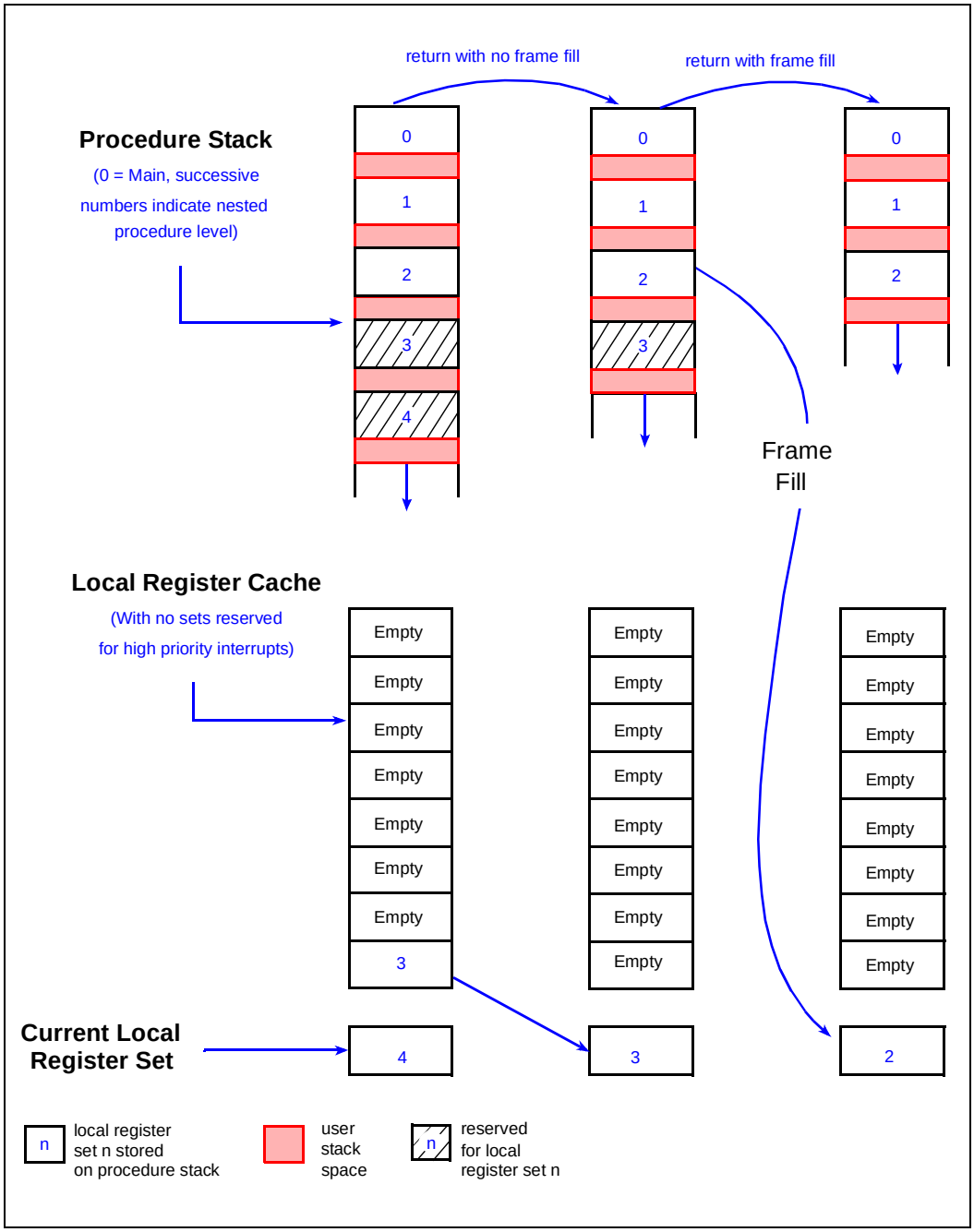


Figure 7-3. Frame Fill



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, then there is no guarantee that the modification is 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 because this may corrupt 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 returns 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 (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 were omitted, then 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 processor 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 executes very quickly because the local register cache has already been flushed by the previous **flushreg**; only synchronization of the PFP is performed. i960 processor implementations may provide other mechanisms to ensure PFP synchronization in addition to **flushreg**, but a **flushreg** after a PFP modification will 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 fits 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, then 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, then 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, then the calling procedure passes a pointer to an argument list on its stack where the remaining return values are 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 in two ways:

- with the **call** and **callx** instructions; or
- with a system-local call as described in Section 7.5, "System Calls" on page 7-13.

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” on page 7-5. 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.)

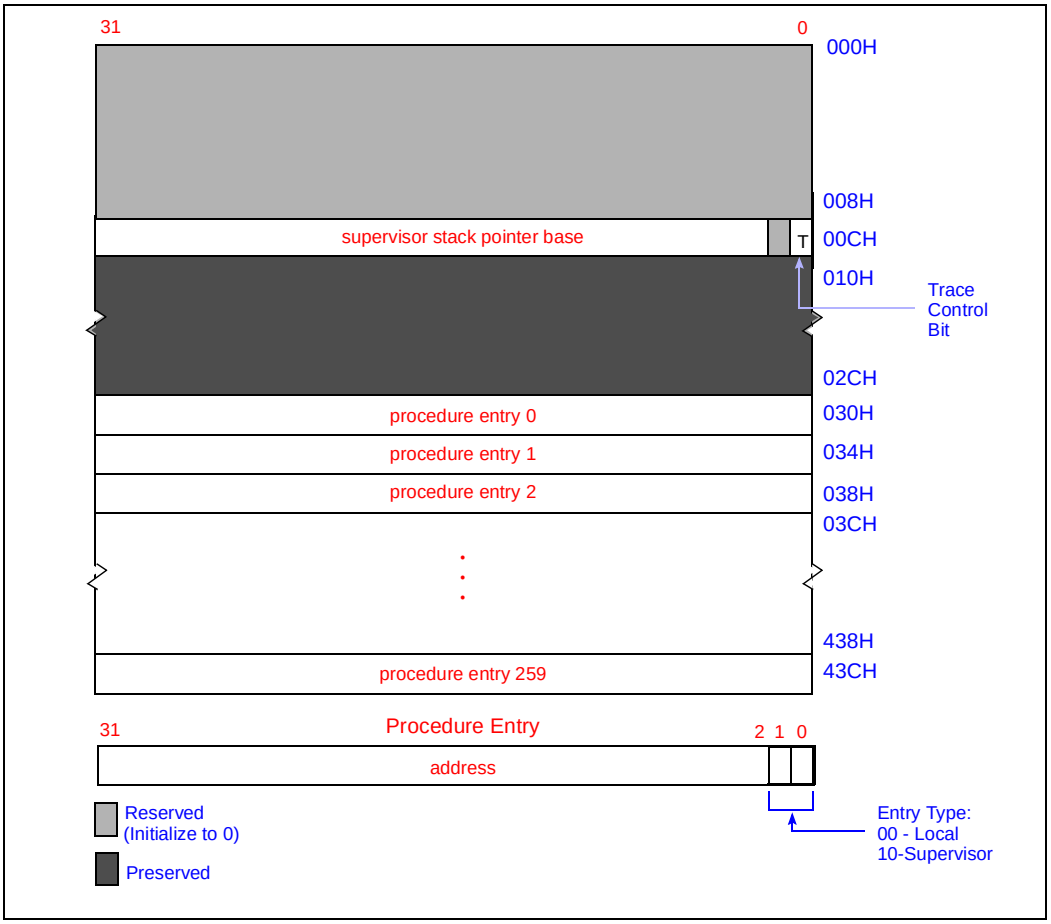
The system call mechanism offers two benefits. First, it supports application software portability. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, application software does not need to be changed each time the implementation 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 Section 3.7, “User-Supervisor Protection Model” on page 3-17.

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 Section 9.1, “Fault Handling Overview” on page 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 it is 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” on page 10-2.

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” on page 7-5. 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 Section 6.2.14, “calls” on page 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” on page 7-5, with the following exceptions:

- If the processor is in user mode, then it switches to supervisor mode.
- If a mode switch occurs, then 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, then the new frame is allocated on the current stack.

- If a mode switch occurs, then 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, then 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" on page 3-17.

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 12.2, "i960® VH Processor Initialization" on page 12-2. 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.

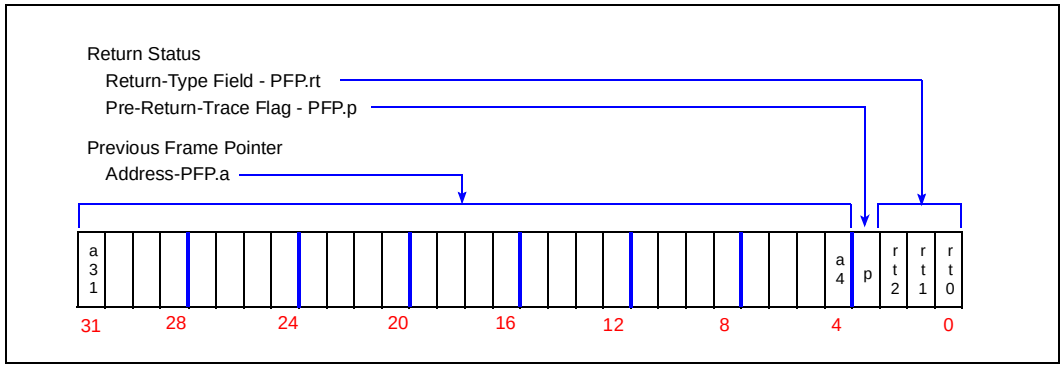
A call to a fault procedure is similar to a system call. Fault procedure calls can be local calls or supervisor calls. The processor obtains pointers to fault procedures through the fault table and (optionally) through the system procedure table.

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 (Figure 7-5) to determine the type of return action to take.

Figure 7-5. Previous Frame Pointer Register – PFP



return-type field indicates the type of call which was made. Table 7-2 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" on page 10-11.

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, then the processor sets the prereturn-trace flag; otherwise it clears the flag. If this flag is set and prereturn-trace mode is enabled, then 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" on page 10-3 for a discussion of interaction between call-trace and prereturn-trace modes with the prereturn-trace flag.

Table 7-2. Encoding of Return Status Field (Sheet 1 of 2)

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]	

Table 7-2. Encoding of Return Status Field (Sheet 2 of 2)

Return Status Field	Call Type	Return Action
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.

Interrupts 8

This chapter describes the i960® core processor architecture interrupt mechanism, the i960® VH processor interrupt controller, peripheral interrupts and secondary PCI interrupt routing. Key topics include the 80960VH'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 80960VH 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.

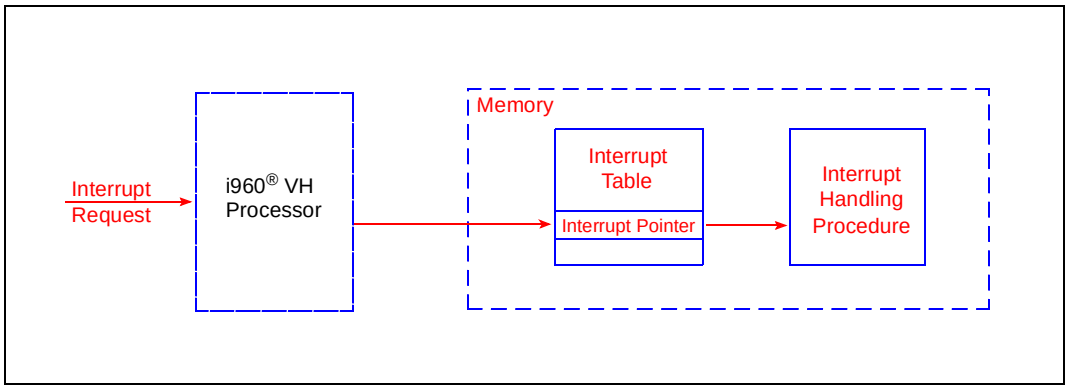
When the interrupt call is made, the processor uses a dedicated interrupt stack. The processor creates a new frame for the interrupt on this stack and a new set of local registers is allocated to the interrupt procedure. The interrupted program's current state is also saved.

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" on page 8-6.

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 (for example, fast SRAM).

Figure 8-1. Interrupt Handling Data Structures



Requests for interrupt service come from many sources and are prioritized so that instruction execution is redirected only when an interrupt request is of higher priority than that of the executing task. On the 80960VH, interrupt requests may originate from external hardware sources, internal peripherals or software. The 80960VH contains a number of integrated peripherals which may generate interrupts, including:

- DMA Channel 0
- DMA Channel 1
- Timers 0 & 1
- Primary ATU
- I^2C Bus Interface Unit
- Messaging Unit
- Memory Controller

The interrupt controller can also intercept external 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® VH Processor Core Interrupt Architecture

The 80960VH 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 (for example, 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" on page 8-5. When multiple interrupt requests are pending at the same priority level, the request with the highest vector number is serviced first.

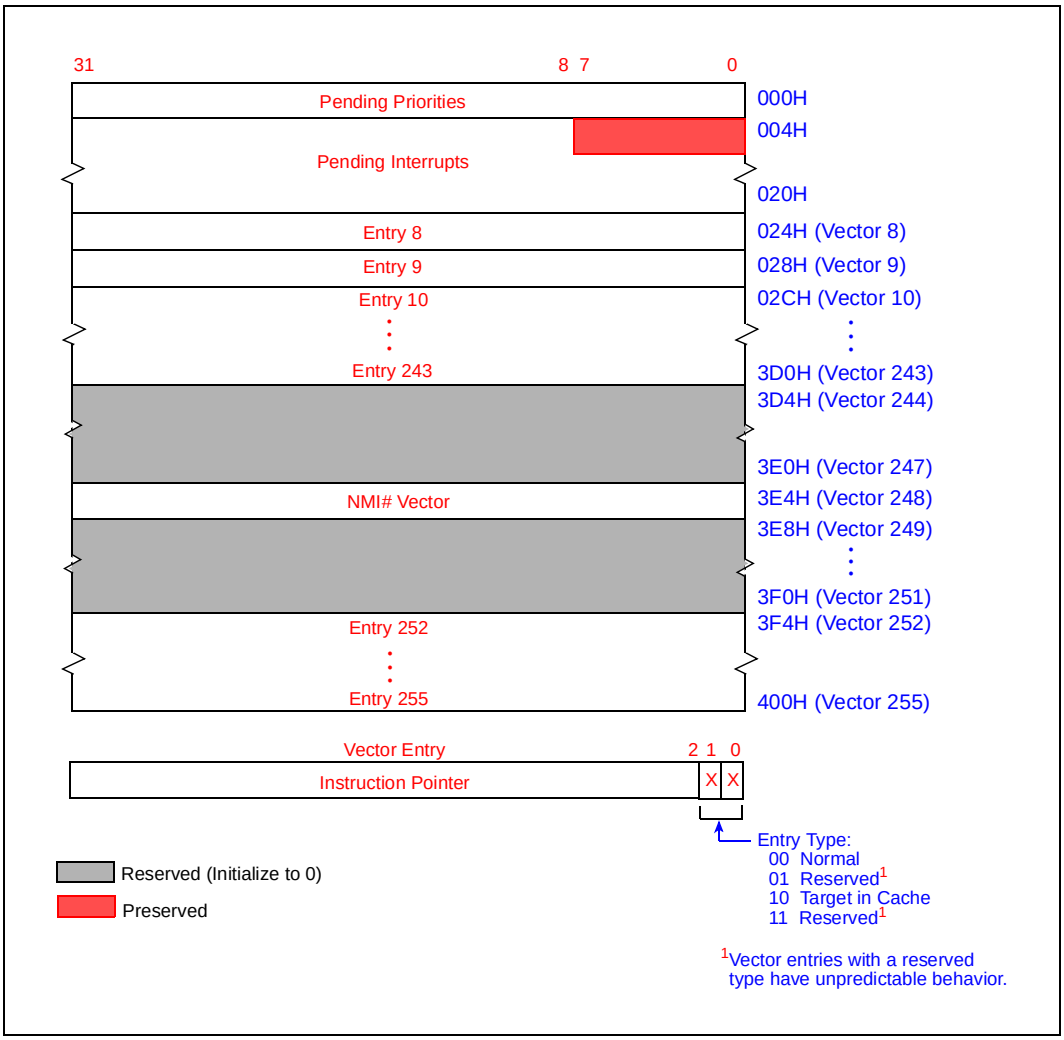
A priority-31 interrupt is handled as a special case. Even when the processor is executing at priority level 31, a priority-31 interrupt will interrupt the processor. On the 80960VH, the non-maskable interrupt (NMI#) interrupts priority-31 execution; no interrupt can interrupt an NMI# handler.

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" on page 8-36. 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; for example, 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 80960VH 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.

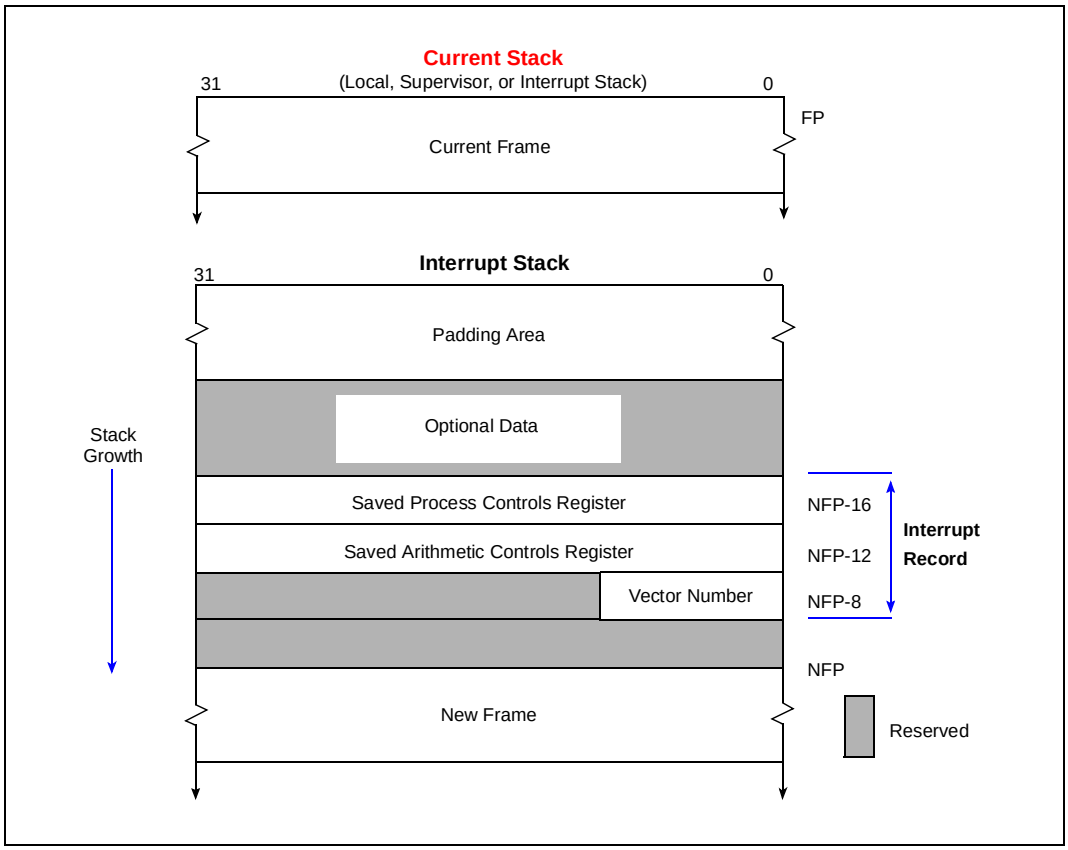
This caching mechanism is non-transparent; the processor may modify fields in a cached interrupt table without modifying the same fields in the interrupt table itself. Vector caching is described in Section 8.5.2.1, "Vector Caching Option" on page 8-35.

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" on page 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 80960VH, 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 80960VH.
- External Interrupts: interrupts posted through the interrupt table, by an external agent to the 80960VH.
- Hardware interrupts: interrupts posted directly to the 80960VH through an implementation-dependent mechanism that may avoid using the interrupt table.

8.1.6.1 Posting Software Interrupts via sysctl

In the 80960VH, **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" on page 6-104 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, and 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 80960VH 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 (Sheet 1 of 2)

```
/* 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)
```

Example 8-2. Interrupt Resolution (Sheet 2 of 2)

```
        { 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 80960VH, 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” on page 8-24.

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, then 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

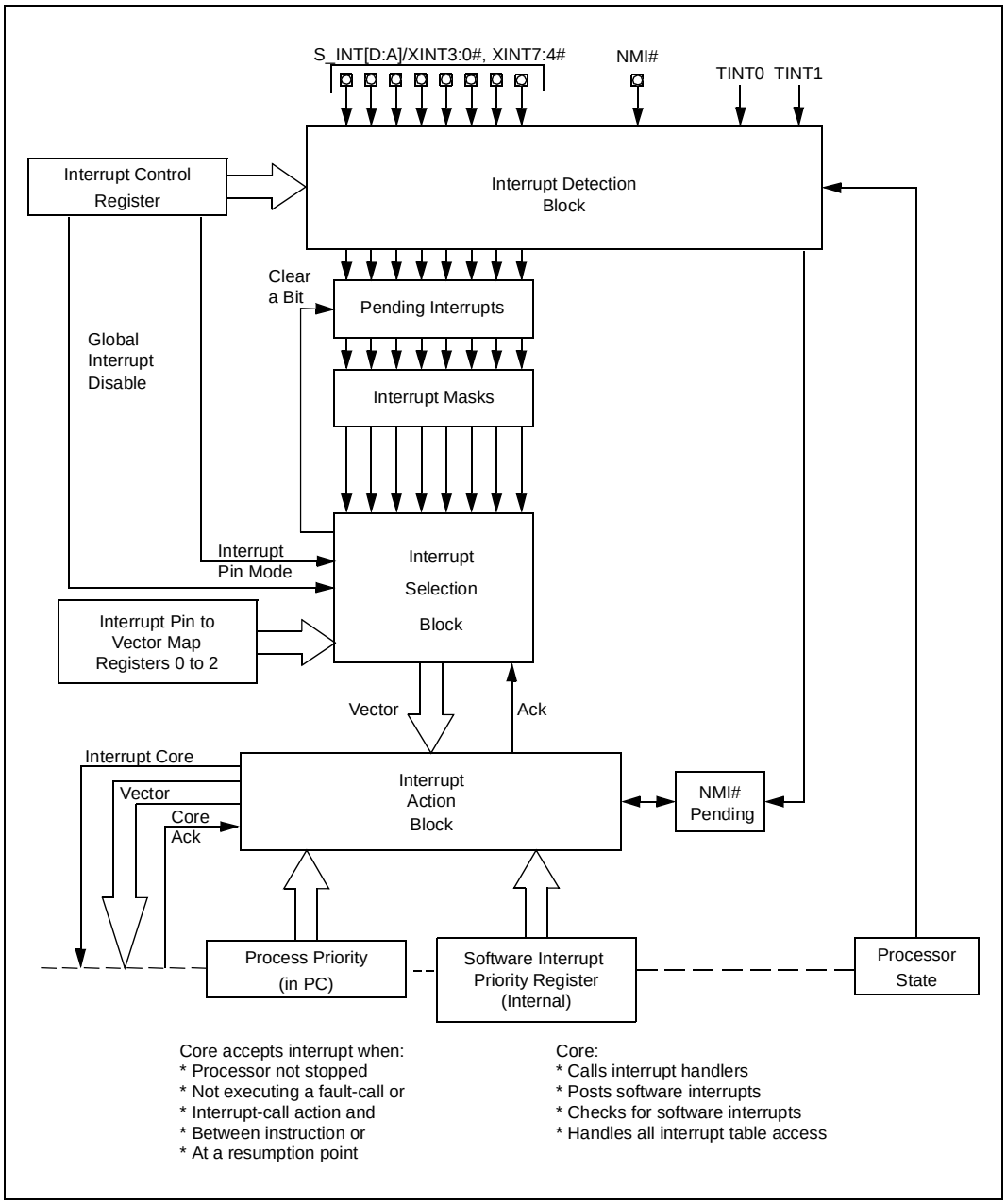
The 80960VH Interrupt Controller Unit (ICU) provides a flexible, low-latency means for requesting and posting interrupts and minimizing the core’s interrupt handling burden. Acting independently from the core, the interrupt controller posts interrupts requested by hardware and software sources and compares 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.

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.

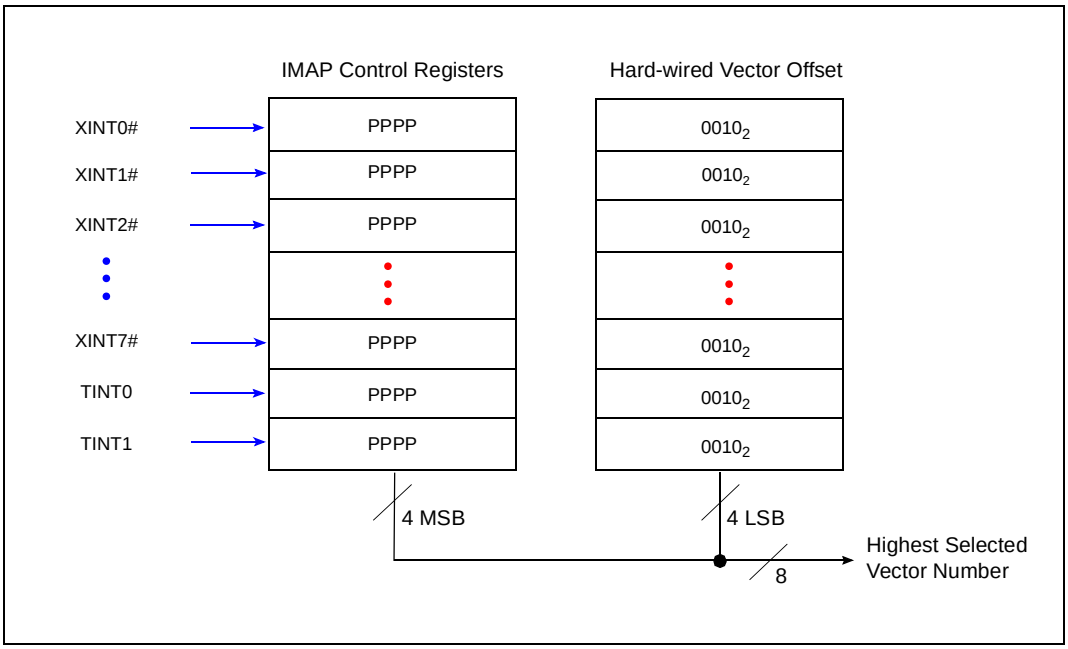
8.2.1 Interrupt Controller Dedicated Mode

The 80960VH 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" on page 8-22 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, then 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, then the interrupt is re-entered after the return is executed.

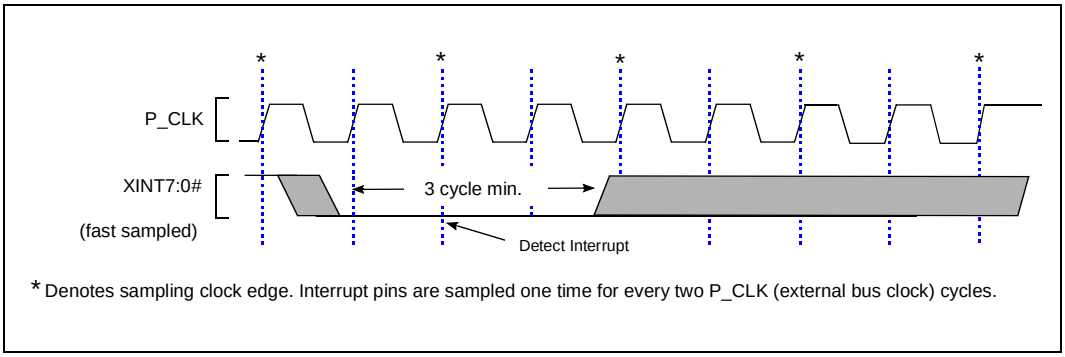
Example 8-3 demonstrates how a level detect interrupt is typically handled. The example assumes that the **ld** from address "timer_0," deactivates the interrupt input.

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 P_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 P_CLK cycles. For practical purposes, this means that asynchronous interrupting devices must generate an interrupt signal that is asserted for at least three P_CLK cycles. See your 80960VH Data Sheet for setup and hold specifications that guarantee detection of the interrupt on particular edges of P_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.

Figure 8-6. Interrupt Fast Sampling



* Denotes sampling clock edge. Interrupt pins are sampled one time for every two P_CLK (external bus clock) cycles.

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 80960VH 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 80960VH, 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 (XINT7:0#), 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 80960VH 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, then 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" on page 8-36 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, on page 8-16 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

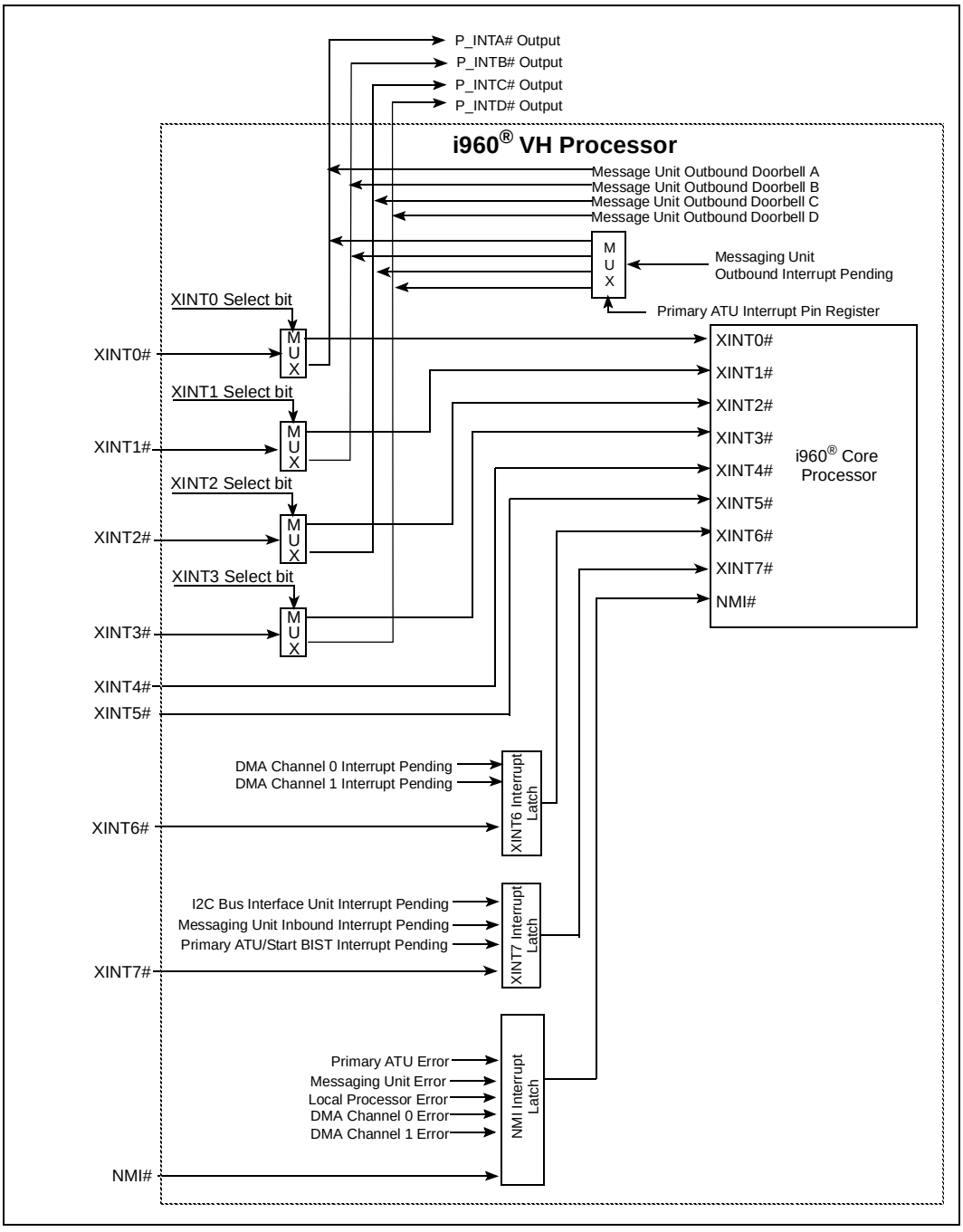
The PCI and peripheral portion of the interrupt controller has two functions:

- 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 some 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 ATU configuration space or the 80960VH local bus.

Figure 8-7. Interrupt Controller Connections for 80960VH



8.3.1 Pin Descriptions

The 80960VH 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 80960VH consists of the following pins:

Table 8-1. Interrupt Input Pin Descriptions

Signal	Description
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.
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.
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.
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 two internal interrupts. They are the interrupts from each of the two 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 three internal interrupts. They are the interrupts from the I^2C 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 five internal interrupts. They include error interrupts from the local processor, primary ATU, Messaging Unit and the two 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” on page 8-12.

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 8-2.

Table 8-2. PCI Interrupt Routing Summary for 80960VH

PIRSR Select Bit	Bit Value	Description
bit 0	1	XINT0# Input Pin routed to i960 core processor XINT0# Input Pin
	0	XINT0# Input Pin routed to P_INTA# Output Pin
bit 1	1	XINT1# Input Pin routed to i960 core processor XINT1# Input Pin
	0	XINT1# Input Pin routed to P_INTB# Output Pin
bit 2	1	XINT2# Input Pin routed to i960 core processor XINT2# Input Pin
	0	XINT2# Input Pin routed to P_INTC# Output Pin
bit 3	1	XINT3# Input Pin routed to i960 core processor XINT3# Input Pin
	0	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 two 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” on page 20-3 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 8-3

Table 8-3. 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		
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 I^2C 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 8-4

Table 8-4. XINT7 Interrupt Sources

Unit	Interrupt Condition	Interrupt Status		Interrupt MASK	
		Register	Bit	Register	Bit
I^2C 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
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 ATU and the i960 core and each of the two DMA channels. Each of the interrupts represents an error condition in the peripheral unit. 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 8-5.

Table 8-5. NMI Interrupt Sources

Unit	Error Condition	Interrupt Status		Interrupt MASK	
		Register	Bit	Register	Bit
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
Messaging Unit	NMI Doorbell	IISR	03	IIMR	03
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
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
NMI# Pin	External Source	N/A	N/A	N/A	N/A

8.3.4 PCI Outbound Doorbell Interrupts

The 80960VH 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 ten memory-mapped control registers. Table 8-6 describes these registers.

Table 8-6. Interrupt Control Registers Memory-Mapped Addresses (Sheet 1 of 2)

Register Name	Description	Address
PIRSR	PCI Interrupt Routing Select Register	0000 12C8H
NISR	NMI Interrupt Status Register	0000 1700H

Table 8-6. Interrupt Control Registers Memory-Mapped Addresses (Sheet 2 of 2)

Register Name	Description	Address
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

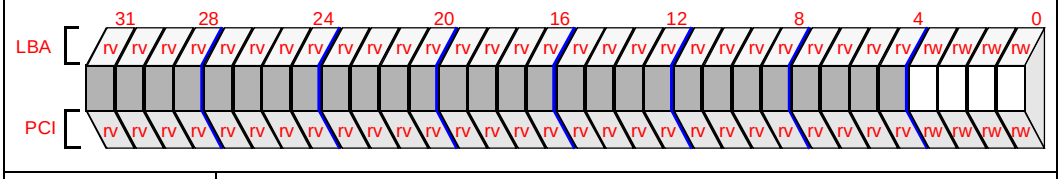
All registers are visible to software as 80960VH memory-mapped registers and can be accessed through the internal memory bus. The PCI Interrupt Routing Select Register is accessible from the internal memory bus and through the PCI configuration register space of the ATU (function #0). See Chapter 11, "Core and Peripheral Control Unit" for additional information regarding the PCI configuration cycles that can access this register.

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 external interrupt inputs which are routed to either the primary PCI interrupts or the i960 core interrupts. If the external interrupt inputs are routed to the primary PCI interrupt pins, then the i960 core XINT3:0# inputs must be set inactive by setting bits 3-0 in the IMSK register to zero.

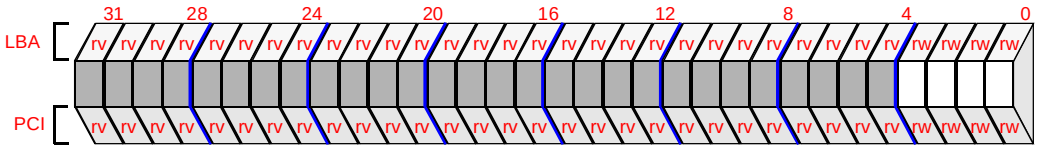
Table 8-7 shows the bit definitions for programming the PCI Interrupt Routing Select Register. The XINT Select bit defaults to a 0.

Table 8-7. PCI Interrupt Routing Select Register – PIRSR (Sheet 1 of 2)



LBA: 12C8H **PCI:** C8H		**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 - (0) Interrupts Routed To P_INTx# Pins (1) Interrupts Routed To i960 core Interrupt Controller Input
02	0_2	XINT2 Select Bit - (0) Interrupts Routed To P_INTx# Pins (1) Interrupts Routed To i960 core Interrupt Controller Input

Table 8-7. PCI Interrupt Routing Select Register – PIRSR (Sheet 2 of 2)



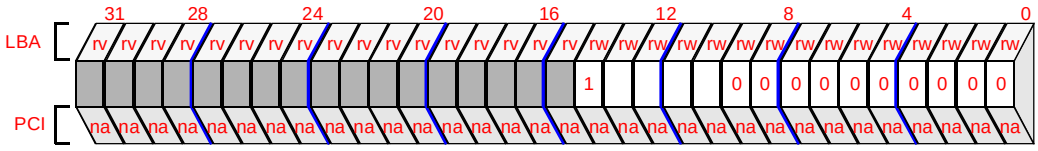
LBA: 12C8H **PCI:** C8H	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
01	0_2	XINT1 Select Bit - (0) Interrupts Routed To P_INTx# Pins (1) Interrupts Routed To i960 core Interrupt Controller Input
00	0_2	XINT0 Select Bit - (0) Interrupts Routed To P_INTx# Pins (1) 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 8-8 describes the layout of the ICON register.

Table 8-8. Interrupt Control Register – ICON



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).

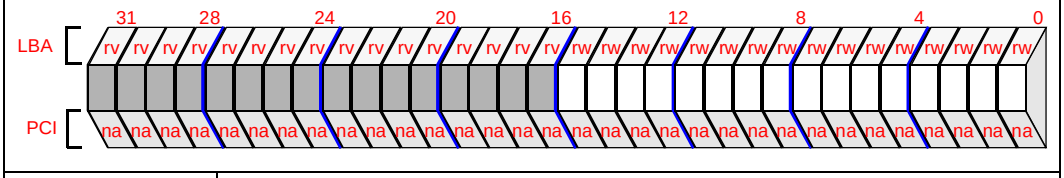
8.4.3 Interrupt Mapping Registers – IMAP0-IMAP2

The IMAP registers (Table 8-9 through Table 8-11) 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.

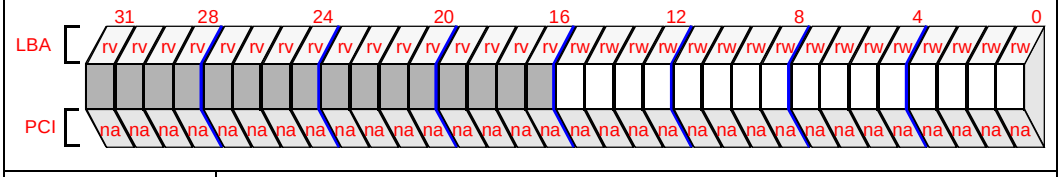
Table 8-9. Interrupt Map Register 0 – IMAP0



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.

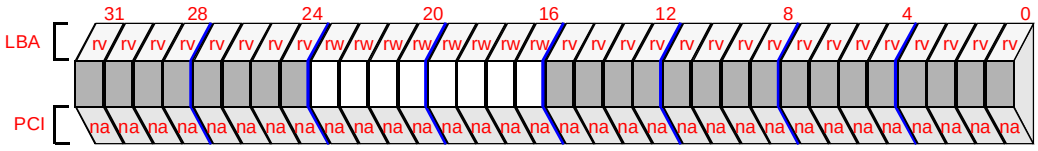
Table 8-10. Interrupt Map Register 1 – IMAP1



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 8-11. Interrupt Map Register 2 – IMAP2



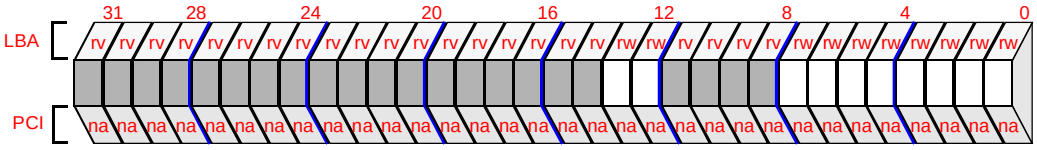
LBA: 8528H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

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.

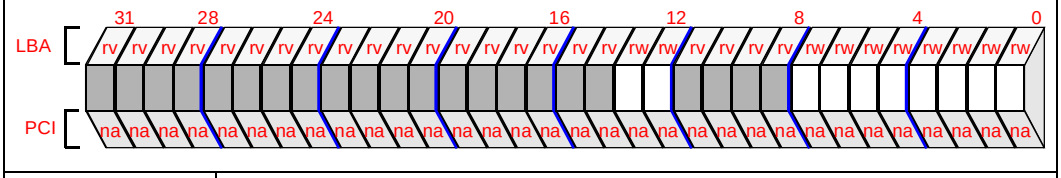
Table 8-12. Interrupt Pending Register – IPND



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 8-13. Interrupt Mask Register – IMSK



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, then 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

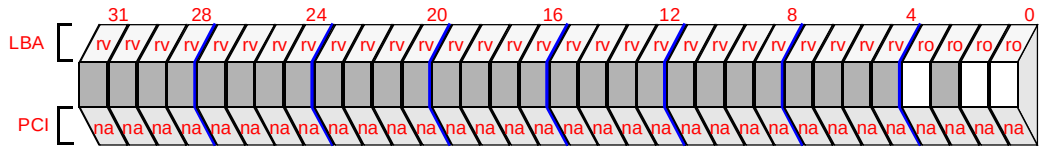
The XINT6 Interrupt Status Register (X6ISR) shows the pending XINT6 interrupts. The source of the XINT6 interrupt can be the internal peripheral devices connected through the XINT6 interrupt latch or the external XINT6# interrupt pin. The interrupts which are connected to the XINT6 input are detailed in Section 8.3.3, “Internal Peripheral Interrupt Routing” on page 8-20.

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 8-3) are cleared. X6ISR reflects the current state of the input to the XINT6 interrupt latch.

Due to the asynchronous nature of the 80960VH 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 8-14 details the X6ISR register.

Table 8-14. XINT6 Interrupt Status Register – X6ISR



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.
02	0_2	Reserved.
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

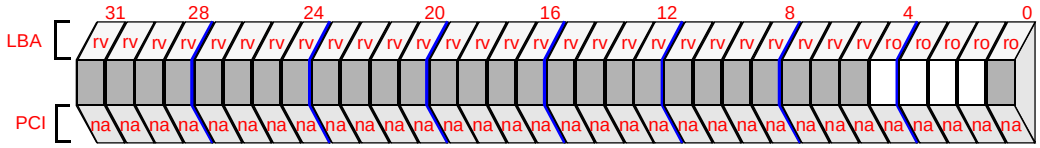
The XINT7 Interrupt Status Register (X7ISR) shows the pending XINT7 interrupts. The source of the XINT7 interrupt can be the internal peripheral devices connected through the XINT7 interrupt latch or the external XINT7# interrupt pin. The interrupts which are connected to the XINT7# input are detailed 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 8-4) are cleared. X7ISR reflects the current state of the input to the XINT7 interrupt latch.

Due to the asynchronous nature of the 80960VH 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 8-14 details the definition of the X7ISR.

Table 8-15. XINT7 Interrupt Status Register – X7ISR



LBA: 1704H **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	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	Reserved.

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 8-5) 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 80960VH requires a level input and must be latched external to the 80960VH.

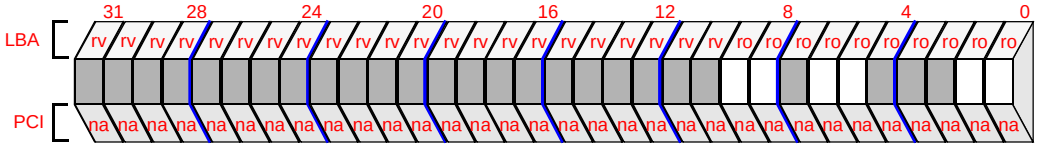
Due to the asynchronous nature of the 80960VH internal peripheral units, multiple interrupts can be active when the application software reads the NISR register. It is up to the application software to handle the occurrence of multiple interrupts. In addition, software must check the contents of the 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 & 32)
      dma_channel_0_error();
if (NISR & 64)
      dma_channel_1_error();
if (NISR & 256)
      messaging_unit_interrupt();
if (NISR & 512)
      external_nmi_interrupt();  }
while( !NISR );
return;
```

Table 8-16 shows the bit definitions for reading the NMI interrupt status register.

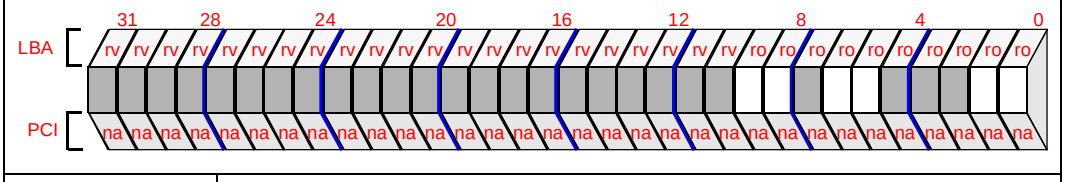
Table 8-16. NMI Interrupt Status Register – NISR (Sheet 1 of 2)



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	Reserved.
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.

Table 8-16. NMI Interrupt Status Register – NISR (Sheet 2 of 2)



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
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	Reserved.
03	0_2	Reserved.
02	0_2	Reserved.
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 8-17 shows the power-up and reset values. Refer to Section 12.4, “Initial Memory Image (IMI)” on page 12-10 for more information on register values after reset.

Table 8-17. Default Interrupt Routing and Status Values Summary

Register	Default Value	Description
PCI Interrupt Routing Select Register	0000 0000H	XINT0# routed to the P_INTA# XINT1# routed to the P_INTB# XINT2# routed to the P_INTC# XINT3# routed to the P_INTD#
NMI Interrupt Status Register	0000 0000H	No interrupts set
XINT7 Interrupt Status Register	0000 0000H	No interrupts set

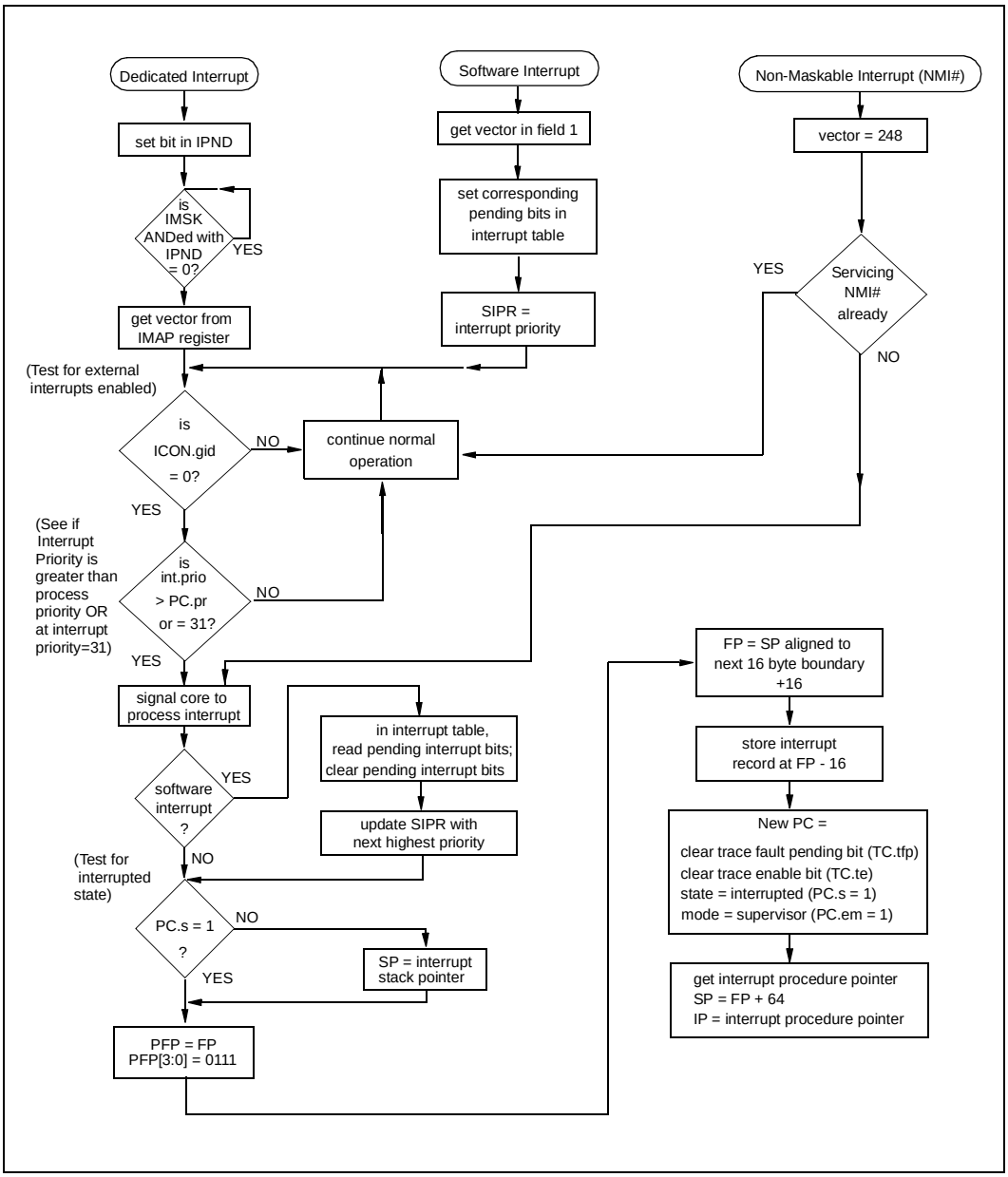
Table 8-17. Default Interrupt Routing and Status Values Summary

Register	Default Value	Description
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 8-8 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.

Figure 8-8. 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 80960VH 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 80960VH loads 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 8-18 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 8-18. 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 8-18. Location of Cached Vectors in Internal RAM (Sheet 1 of 2)

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

Table 8-18. Location of Cached Vectors in Internal RAM (Sheet 2 of 2)

Vector Number (Binary)	Vector Number (Decimal)	Internal RAM Address
$0101\ 0010_2$	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 80960VH'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" on page 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 13.2, "Programming the Physical Memory Attributes (Pmcon Registers)" on page 13-3 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 8-19 shows the base latencies for all interrupt types, with varying vector caching options.

Table 8-19. Base Interrupt Latency

Interrupt Type	Vector Caching Enabled	Typical 80960VH Latency (Bus Clocks)[1]
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

NOTE:
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 8-19, except for instruction execution.

Table 8-20. Worst-Case Interrupt Latency Controlled by divo to Destination r15

Interrupt Type	Vector Caching Enabled	Worst 80960VH 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

NOTE:
1. a = MAX (0,N - 11) where "N" is the number of bus cycles needed to perform a word load.

Table 8-21. Worst-Case Interrupt Latency Controlled by divo to Destination r3 (Sheet 1 of 2)

Interrupt Type	Vector Caching Enabled	Worst 80960VH Latency (Bus Clocks)[1]
NMI#	Yes	60

NOTE:
1. a = MAX (0,N - 7) where "N" is the number of bus cycles needed to perform a word load.

Table 8-21. Worst-Case Interrupt Latency Controlled by divo to Destination r3 (Sheet 2 of 2)

Interrupt Type	Vector Caching Enabled	Worst 80960VH Latency (Bus Clocks)[1]
XINT5:4#, TINT1:0	Yes	65
	No	72+a
XINT7:6#, XINT3:0#	Yes	66
	No	73+a

NOTE:
1. a = MAX (0,N - 7) where “N” is the number of bus cycles needed to perform a word load.

Table 8-22. Worst-Case Interrupt Latency Controlled by calls

Interrupt Type	Vector Caching Enabled	Worst 80960VH 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

NOTE:
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 8-23. Worst-Case Interrupt Latency When Delivering a Software Interrupt

Interrupt Type	Vector Caching Enabled	Worst 80960VH Latency (Bus Clocks)[1]
NMI#	Yes	97
XINT5:4#, TINT1:0	Yes	99
	No	107+a
XINT7:6#, XINT3:0#	Yes	100
	No	108+a

NOTE:
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 flushreg of One Stack Frame

Interrupt Type	Vector Caching Enabled	Worst 80960VH 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

NOTE:
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 XINT3:0# 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.

Faults 9

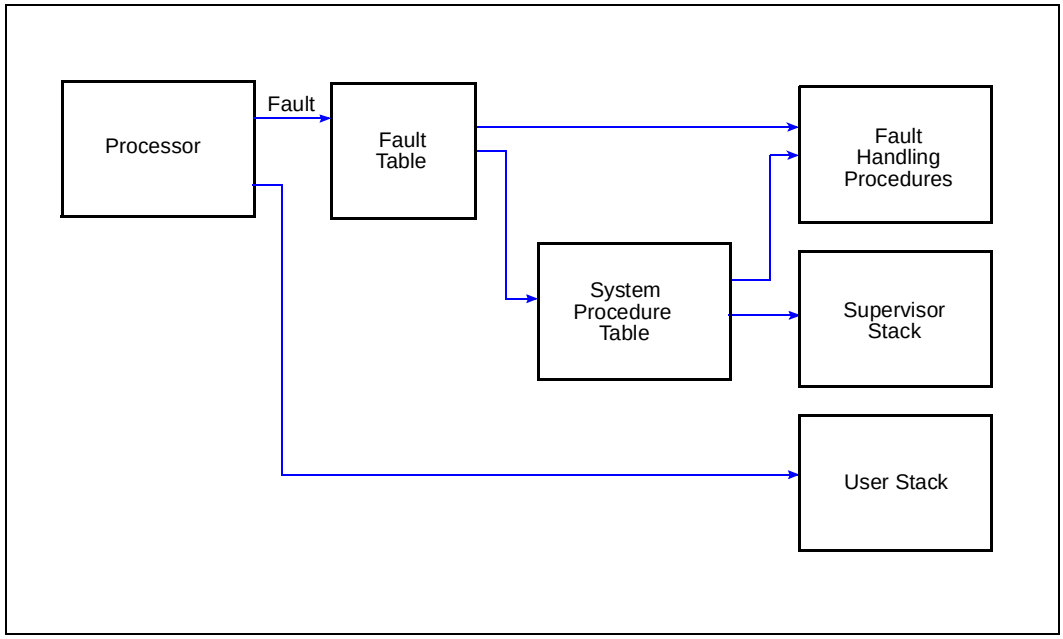
This chapter describes the i960® VH processor's fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanisms. See Section 9.10, "Fault Reference" on page 9-18 for detailed information on each fault type.

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” on page 7-17 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 realigned 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 (Section 9.10, “Fault Reference” on page 9-18) 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 80960VH recognizes i960 architecture-defined faults and a new fault subtype for detecting unaligned memory accesses. Table 9-1 lists all faults that the 80960VH detects, arranged by type and subtype. Text that follows Table 9-1 gives column definitions.

Table 9-1. i960® VH 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" on page 9-9
1H	TRACE	Bit 1	INSTRUCTION	0001 0002H
		Bit 2	BRANCH	0001 0004H
		Bit 3	CALL	0001 0008H
		Bit 4	RETURN	0001 0010H
		Bit 5	PRERETURN	0001 0020H
		Bit 6	SUPERVISOR	0001 0040H
		Bit 7	MARK/BREAKPOINT	0001 0080H
2H	OPERATION	1H	INVALID_OPCODE	0002 0001H
		2H	UNIMPLEMENTED	0002 0002H
		3H	UNALIGNED	0002 0003H
		4H	INVALID_OPERAND	0002 0004H
3H	ARITHMETIC	1H	INTEGER_OVERFLOW	0003 0001H
		2H	ZERO-DIVIDE	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.

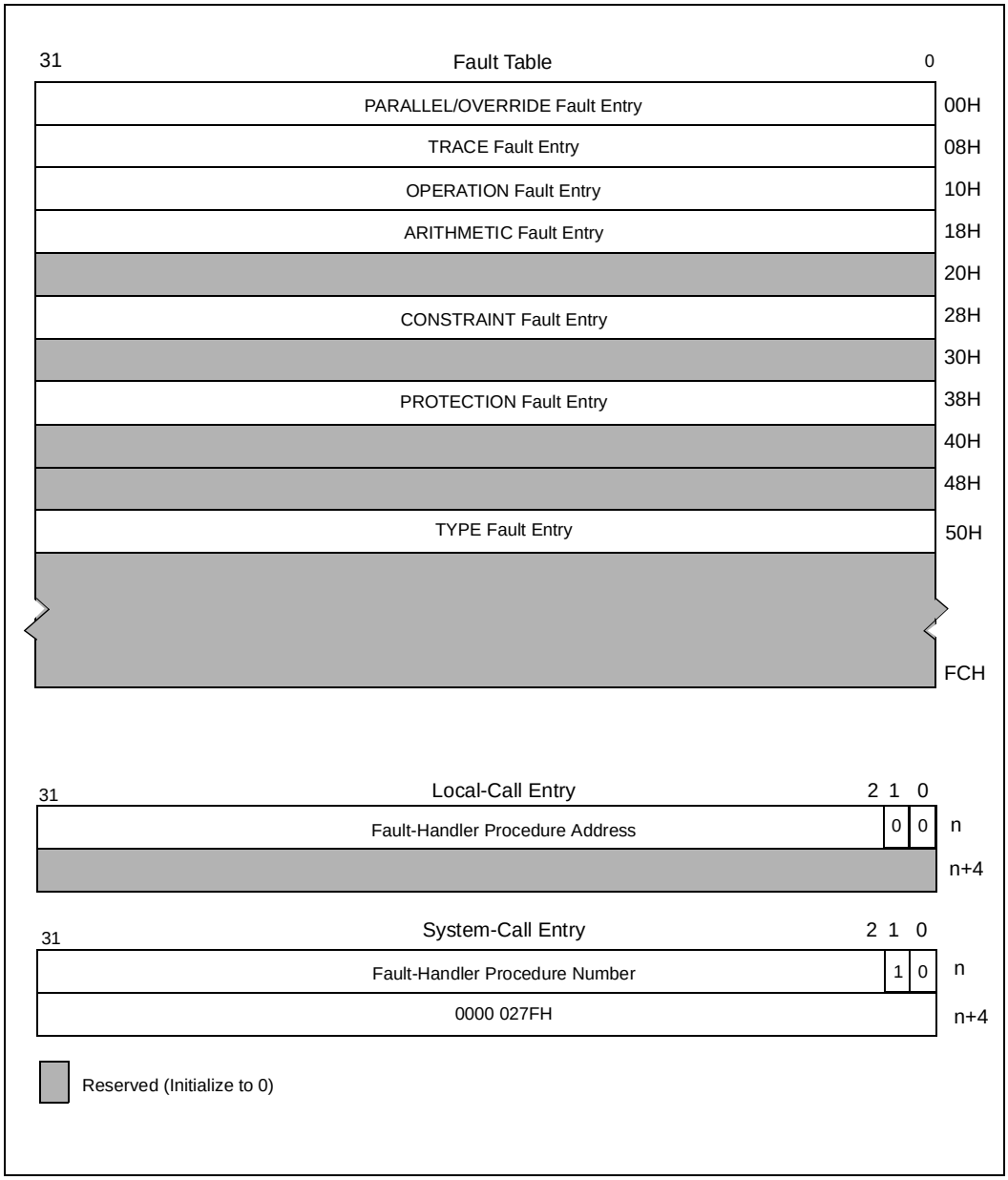
The fault types are used consistently for all family members. For example, Fault Type 4H is reserved for floating point faults. Any i960 processor with floating point operations uses Entry 4H to store the pointer to the floating point fault handling procedure.

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, then 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 uses to handle the fault.

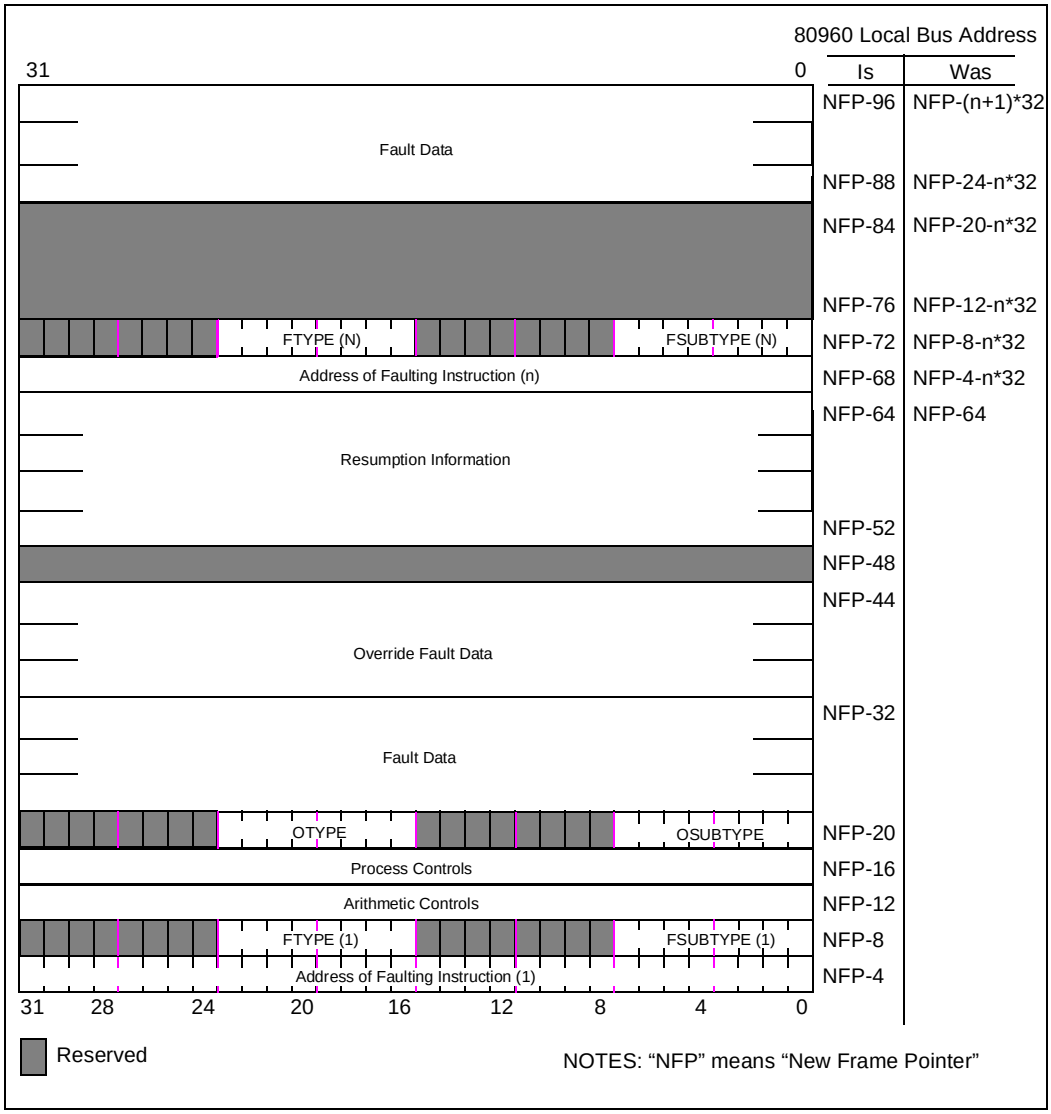
9.5.1 Fault Record Description

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.

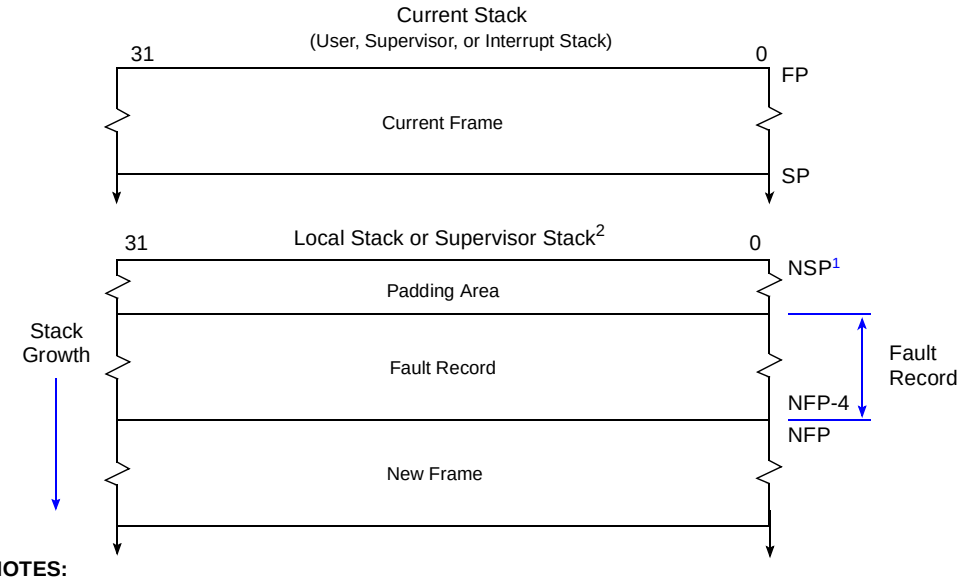
Figure 9-3. Fault Record



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.

Figure 9-4. Storage of the Fault Record on the Stack



NOTES:

1. If the call to the fault handler procedure does not require a stack switch, then 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, then the processor switches to the supervisor stack.

9.6 Multiple and Parallel Faults

Multiple fault conditions can occur during a single instruction execution and during a 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 reports 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 (Section 9.9, “Precise and Imprecise Faults” on page 9-16). 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 (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" on page 9-18.
- 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 80960VH 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 (Section 9.9.4, "No Imprecise Faults (AC.nif) Bit" on page 9-17). This mechanism allows instructions that can fault to be executed in parallel with other instructions or to be executed 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 points 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.

Example 9-1. Imprecise Fault Generations

```
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 (Section 9.9.5, "Controlling Fault Precision" on page 9-18), 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 then returns to the next instruction not yet executed in the program flow.

On the 80960VH, 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” on page 9-6. 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 80960VH, 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 80960VH 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” on page 7-17 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 realigned 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, then 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 12.3.1.5, “FAIL# Code” on page 12-9.

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 (for example, fetch from on-chip RAM)
- During instruction execution (for example, integer overflow)
- Immediately following execution (for example, 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 so 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 register. 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" on page 9-18 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" on page 9-18.

9.7.4 Returning to Point in Program Where Fault Occurred

As described in Section 9.7.2, "Program Resumption Following a Fault" on page 9-12, most faults can be handled so 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" on page 6-84.

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" on page 9-18.

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" on page 9-16.

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" on page 9-18. 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 13.4.2, "Bus Transactions Across Region Boundaries" on page 13-5.

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” on page 7-5, 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 (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” on page 9-12.

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” on page 7-6, 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” on page 7-5, 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 (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” on page 7-6 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 is in user mode when the fault occurs, 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 faults, 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” on page 9-22, 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” on page 9-18 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 (for example, 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 80960VH.

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 reduces 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 (for example, 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.

Notes: 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 exhibit 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” on page 9-9 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 exhibit 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:

	NumberName
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, then 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, then the interrupt is serviced before the TRACE fault is handled. The TRACE.PRERETURN fault is different. Since the fault 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 80960VH 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 occurs before execution of the instruction. Machine state is not changed.

Trace Reporting: The trace event is lost.

Tracing and Debugging 10

This chapter describes the i960® VH 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 80960VH 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.

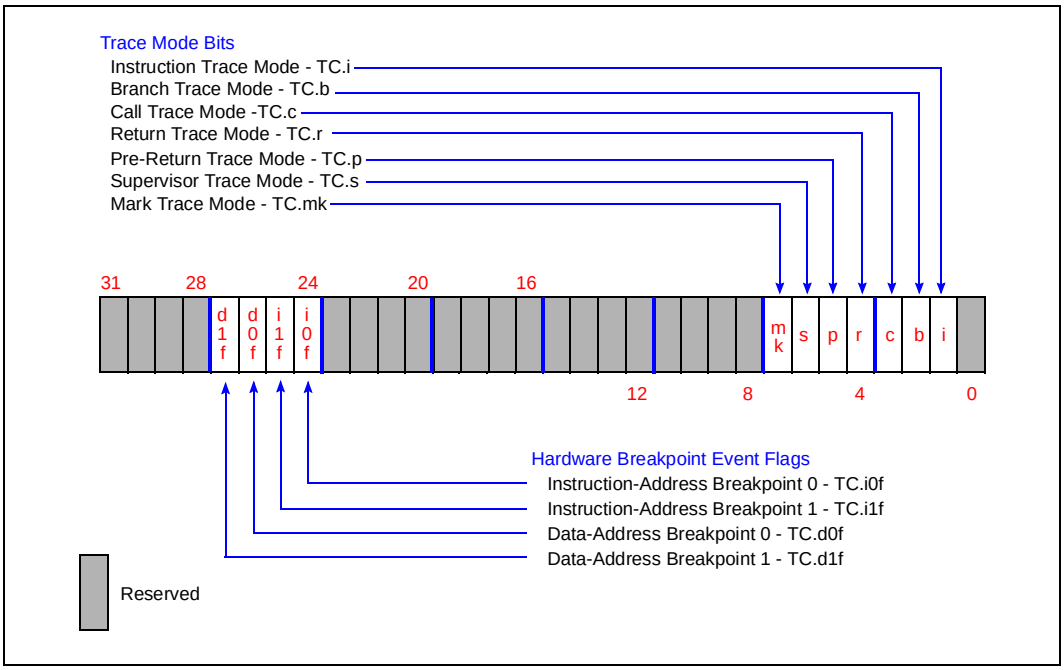
- TC register mode bits
- DAB0-DAB1 registers' address field and enable bit (in the control table)
- System procedure table supervisor-stack-pointer field trace control bit
- IPB0-IPB1 registers' address field (in the control table)
- PC register trace enable bit
- PFP register return status field prereturn 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 (Figure 10-1) allows software to define conditions that generate trace events.

Figure 10-1. i960® VH processor Trace Controls Register – TC



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, “Trace Modes” on page 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 80960VH, 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 (Section 3.6.3, “Process Controls Register – PC” on page 3-15). 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” on page 10-11.

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 (for example, 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 (Section 10.5.2.4, “Tracing on Return from Implicit Call: Fault Case” on page 10-13).

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
- Branch trace
- Mark trace
- Prereturn trace
- Call trace
- Return trace
- Supervisor trace

See Section 10.4, “Handling Multiple Trace Events” on page 10-10 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” on page 10-11 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” on page 10-11 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” on page 10-12.

A return from fault may be traced whereas a return from interrupt cannot be traced. See Section 10.5.2.4, “Tracing on Return from Implicit Call: Fault Case” on page 10-13 and Section 10.5.2.5, “Tracing on Return from Implicit Call: Interrupt Case” on page 10-13 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, then 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 used to generate 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, then **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 80960VH 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)” on page 3-5 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, then none are available for application code. If an emulation tool is not being used in conjunction with the device, then modification rights to breakpoint resources are 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, then an OPERATION.UNIMPLEMENTED fault is generated. In this case, the breakpoint resource are not 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 are 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 10-1.

Table 10-1. *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 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.
```

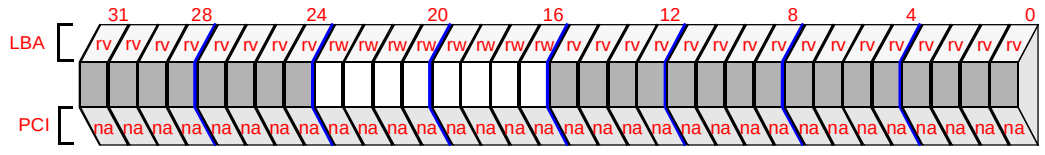
Assume in this example that after execution of the **sysctl** instruction, the value of r4 is 0000 0022H. This indicates that the application has gained modification rights to both instruction and data address breakpoint registers. If the value returned is zero, then the application has not gained the rights to the breakpoint resources.

Because the 80960VH 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 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 10-2 and Table 10-5. 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 generates a break event.

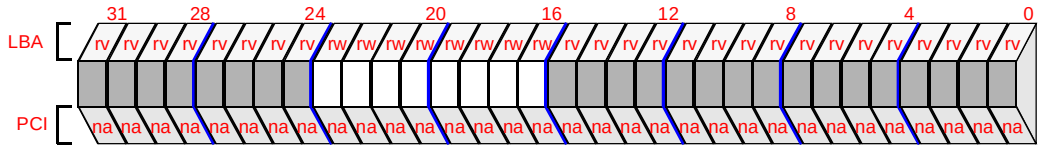
Table 10-2. Breakpoint Control Register – BPCON (Sheet 1 of 2)



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

Table 10-2. Breakpoint Control Register – BPCON (Sheet 2 of 2)



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
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 10-3 and Table 10-4.

Table 10-3. 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 10-4.

Table 10-4. 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 10-5. 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 (Section 10.2.7.4, "Breakpoint Control Register – BPCON" on page 10-6), qualify the access types that DAB matches. 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 causes 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:

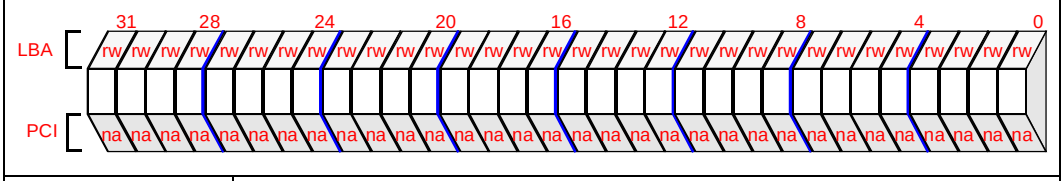
```
ldt 0x1000,r8
```

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, one cannot predict whether or not breakpoint traces are generated when an OPERATION fault or TYPE.MISMATCH fault occurs. The OPERATION or TYPE.MISMATCH fault is always reported in this case.

Table 10-5. Data Address Breakpoint Register – DABx



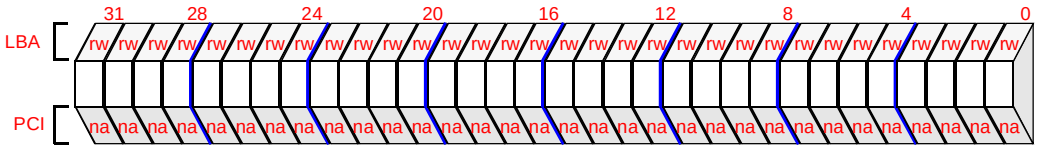
LBA: Ch 0-8420H Ch 1-8424H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	Data Address.

10.2.7.6 Instruction Breakpoint Registers – IPBx

The format for the instruction breakpoint registers is given in Table 10-6. The upper 30 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 10-6. Instruction Breakpoint Register – IPBx



LBA: Ch 0-8400H Ch 1-8404H **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	Instruction Address.
01	0_2	IPBX Mode: IPB1
00	0_2	IPBX Mode: IPB0

Programming the instruction breakpoint register modes is shown in Table 10-7.

On the 80960VH, 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 is always correct.

If it is necessary to avoid this condition, then use the modify memory-mapped control register operation of the **sysctl** instruction to update the IPB registers.

Table 10-7. 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 always reports a supervisor event and it always signals at least one event.

If the processor reports prereturn trace and other trace types at the same time, then 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” on page 9-11 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” on page 9-11 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” on page 10-11 and Section 10.5.2.5, “Tracing on Return from Implicit Call: Interrupt Case” on page 10-13 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 (Section 10.5.2.3, “Tracing on Return from Explicit Call” on page 10-12). The **calls** instruction and all instructions of the procedure called are traced according to the new PC.te.

Table 10-8. 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

NOTE: Refer to Table 7-2 “Encoding of Return Status Field” on page 7-17.

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 10-9 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 10-9 summarizes all cases.

Table 10-9. 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[1]	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

NOTES:
1. On the 80960VH 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 80960VH, 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 10-10 shows all cases.

Table 10-10. Tracing on Return from Explicit Call (Sheet 1 of 2)

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

Refer to Table 7-2 “Encoding of Return Status Field” on page 7-17.

Table 10-10. Tracing on Return from Explicit Call (Sheet 2 of 2)

PFP.rt2:0	Execution Mode PC.em	Trace Enable Used for Trace on Return
$01a_2$	supervisor	t_2 (from PFP.r2:0)

Refer to Table 7-2 “Encoding of Return Status Field” on page 7-17.

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, then 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 80960VH, the interrupt handler returns directly to the trace fault handler.

If the interrupt return is executed from user mode, then the PC register is not restored and tracing of the return occurs according to the PC.te and TC.modes bit fields.

Core and Peripheral Control Unit 11

11.1 Overview

This chapter describes how to select the operating speed of the i960® VH processor. It also describes how the 80960 processor core and the local bus on the 80960VH can be reset. These registers are extended registers of the Address Translation Unit (ATU), therefore they can be accessed through either the primary PCI bus or the 80960 processor local bus.

11.2 Register Definitions

Refer to Chapter 16, "Address Translation Unit" for all of the ATU extended configuration registers. The ATU extended registers shown in Table 11-1 are described in the following sections.

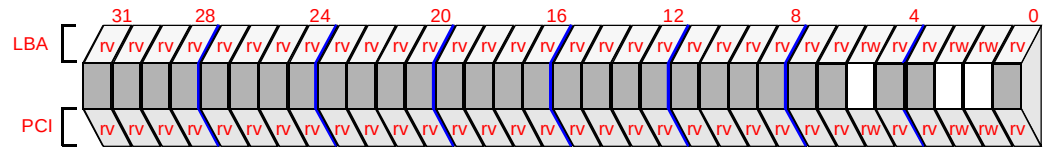
Table 11-1. ATU Extended Configuration Register Addresses

Register Name	Register Size in Bits	PCI Configuration Cycle Register Number	Internal Bus Address
Reset/Retry Control Register - RRCR	32	49	0000.12C4H
PCI Interrupt Routing Select Register - PIRSR	32	50	0000.12C8H
Core Select Register - CSR	32	51	0000.12CCH

11.2.1 Reset/Retry Control Register - RRCR

The Reset/Retry Control Register is used to control how the 80960 processor core and the local bus on the 80960VH can be reset. This register also provides a mechanism to allow PCI configuration cycles to be retried.

Table 11-2. Reset/Retry Control Register - RRCR (Sheet 1 of 2)

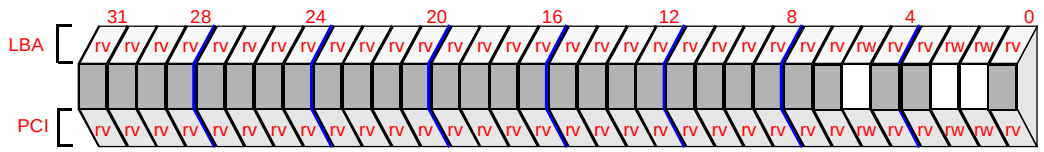


LBA: 12C4H
PCI: C4H

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
31:06	0000 000H	Reserved.
05	0_2	Reset Local Bus - When set, the 80960 processor core and all units on the local bus shall be reset, except for the Core and Peripheral Unit. The i960 VH processor hardware will clear this bit after the reset operation completes. Note that the i960 core processor will be held in reset if the default value of the Core Processor Reset bit in the RRCR is set.

Table 11-2. Reset/Retry Control Register - RRCR (Sheet 2 of 2)



LBA: 12C4H
PCI: C4H

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	Reserved.
03	0_2	Reserved.
02	Varies with external state of **RETRY** pin at primary PCI bus reset	Configuration Cycle Retry - When this bit is set, the primary PCI interface of the i960 VH processor will respond to all configuration cycles with a Retry condition. When clear, the i960 VH processor will respond to the appropriate configuration cycles. The default condition for this bit is based on the external state of the **RETRY** pin at the rising edge of **P_RST#**. If the external state of the pin is high, then the bit is set. If the external state of the pin is low, then the bit is cleared. When the **RST_MODE#** pin is high, this bit will be forced to a zero regardless of the state of the external **RETRY** pin. Refer to Chapter 12, "Initialization and System Requirements" for more details on the i960 VH processor initialization modes.
01	Varies with external state of **RST_MODE#** pin at primary PCI bus reset	Core Processor Reset - This bit is set to its default value by the hardware when either P_RST# is asserted or the Reset Local Bus bit in the RRCR is set. When this bit is set, the i960 core processor is being held in reset. Software cannot set this bit. Software will be required to clear this bit to deassert 80960 processor reset. The default condition for this bit is based on the external state of the **RST_MODE#** pin at the rising edge of **P_RST#**. If the external state of the pin is low, then the default value of this bit is set. If the external state of the pin is high, then the default value of this bit is clear.
00	0_2	Reserved.

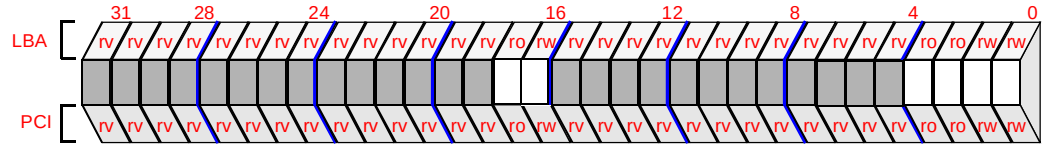
11.2.2 PCI Interrupt Routing Select Register - PIRSR

Refer to Section 8.4.1, "PCI Interrupt Routing Select Register (PIRSR)" on page 8-23 for a description of this register.

11.2.3 Core Select Register - CSR

The Core Select Register is used to select the operating speed of the 80960VH. The PCI bus and the 80960 local bus operate at the same frequency as the system clock, P_CLK. The 80960 processor core can operate in DX, DX2, or DX4 modes. The 80960 processor core speed can be selected by either controlling bits in the CSR register or by controlling the external CLKMODE1:0# pins. When both CLKMODE1:0# pins are high, the 80960 processor speed can be selected by controlling the content of the CSR register, otherwise the speed is selected based on the states of the CLKMODE1:0# pins. Refer to Table 11-4.

Table 11-3. Core Select Register - CSR



LBA: 12CCH
PCI: CCH

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.
16	0_2	Primary Discard Timer Value - This bit controls the timeout value for the primary delayed read and delayed write discard timers. A value of 0 indicates the timeout value is 2^{15} clocks. A value of 1 indicates the timeout value is 2^{10} clocks.
15:04	000H	Reserved.
03	Varies with the external state of the CLKMODE1# pin at reset	CLKMODE1# Pin Status - This bit indicates the external state of the external CLKMODE1# pin. This bit's default condition is based on the external state of the CLKMODE1# 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 bit is cleared.
02	Varies with the external state of the CLKMODE0# pin at reset	CLKMODE0# Pin Status - This bit indicates the external state of the external CLKMODE0# pin. This bit's default condition is based on the external state of the CLKMODE0# 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 bit is cleared.
01:00	01_2	Clock Mode Bits - These bits are used to select the operating speed of the 80960 processor core. The 80960 processor core can operate in DX, DX2, and DX4 modes. These bits are only operational when the external CLKMODE1:0# pins are both high. When CLKMODE1:0# pins are both high, the default value of these bits are 01_2 after reset, for example, DX mode. Software must alter these bits to select the operating speed of the 80960 processor core. (00) - DX Mode (01) - DX Mode (10) - DX2 Mode (11) - DX4 Mode

Table 11-4 shows how to control the 80960 processor core speed using the external CLKMODE1:0# pins.

Table 11-4. Selecting the 80960 Processor Speed

CLKMODE1:0# Pins	Description
11_2	The 80960 processor core operating speed is selected using the Clock Mode bits in the CSR register. The CSR register can be accessed via the PCI bus. The default speed is DX mode after reset.
10_2	The i960 VH processor operates in DX mode. The Clock Mode bits in the CSR register are not operational.
01_2	The i960 VH processor operates in DX2 mode. The Clock Mode bits in the CSR register are not operational.
00_2	The i960 VH processor operates in DX4 mode. The Clock Mode bits in the CSR register are not operational.

Initialization and System Requirements 12

This chapter describes the steps that the i960® VH 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.

12.1 Overview

The 80960VH 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 80960VH'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 80960VH. 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.

12.1.1 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.

12.1.2 General Initialization

The 80960VH 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 80960VH also supports JTAG boundary scan (see Chapter 22, "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 (P_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.

12.2 i960® VH Processor Initialization

Several functional units within the 80960VH must be initialized before system operation. These are the Core and Peripheral Control Unit, Address Translation Unit (ATU), i960 core processor and Memory Controller. The order in which they are initialized is dependent on how the 80960VH is used in the system. The initialization process begins when the Primary PCI Bus Reset signal (P_RST#) is asserted.

12.2.1 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 12-1 describes the relationship between the RST_MODE# and RETRY signal values and the initialization mode.

Table 12-1. Initialization Modes

RST_MODE#	RETRY	Initialization Mode	Primary PCI Interface	i960 Core Processor
0	X	Mode 0	Accepts Transactions	Held in Reset
1	0	Mode 1	Accepts Transactions	Initializes
1	1	Mode 2 (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 Reset/Retry Control Register (RRCR). See Chapter 11, "Core and Peripheral Control 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 RRCR 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 RRCR. When RETRY is active and P_RST# is de-asserted, the 80960VH 33/3.3 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 80960VH accepts PCI configuration cycles on the primary PCI bus.

Figure 12-1 shows a flow chart of the initialization process.

12.2.2 Mode 0 Initialization

Mode 0 allows a host processor to configure the 80960VH peripherals while the i960 core processor is held in reset. The host processor configures the Core and Peripheral Control Unit. 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 RRCR. This deasserts the internal reset signal on the i960 core processor and the processor begins its initialization process.

12.2.3 Mode 1 Initialization

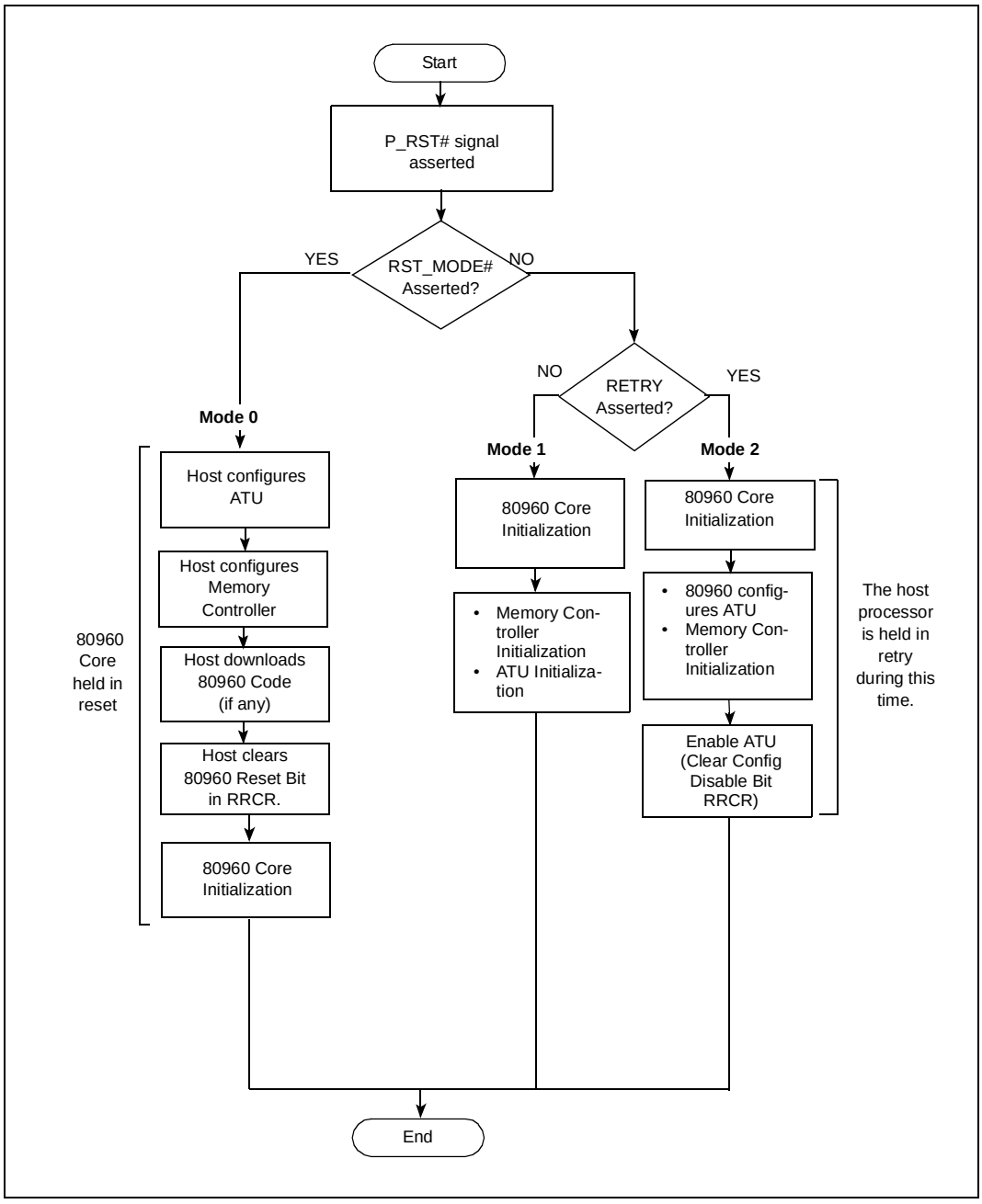
Mode 1 allows configuration cycles on the ATU at any time and allows the i960 core processor to initialize after reset. Mode 1 allows each unit of the 80960VH to be initialized in its own manner. All units are reset when the P_RST# signal is asserted. Each unit returns to its default state. Be aware that race conditions may exist between 80960 operation after reset and PCI configuration.

12.2.4 Mode 2 (Default Mode)

Mode 2 allows the i960 core processor to initialize and control the initialization process before the host processor is allowed to configure the 80960VH 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 RRCR. 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 80960VH appears to the PCI configuration software.

Figure 12-1. Initialization Examples Flow Chart



12.2.5 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 12-2. All of the bus masters are initialized to the highest priority. None of the devices are disabled at powerup.

Table 12-2. 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

12.2.6 Reset State Operation

The 80960VH has two reset conditions:

- P_RST#
- L_RST#

Each is described in detail in the following sections.

12.2.6.1 i960® VH Processor Reset State Operation

The P_RST# signal, when asserted, causes the 80960VH 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 (P_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 80960VH Datasheet.

While the processor's P_RST# signal is asserted, output signals are driven to the states as indicated in Table 12-2. User software cannot reset the entire 80960VH; however, the **sysctl** instruction can reset the i960 core processor. The P_RST# signal must be asserted to enter the reset state. See Section 12.6, "Reinitializing and Relocating Data Structures" on page 12-22.

12.2.6.2 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 RRCR when the ATU and DMA have indicated that they are off the PCI bus. L_RST# also asserts when P_RST# asserts.

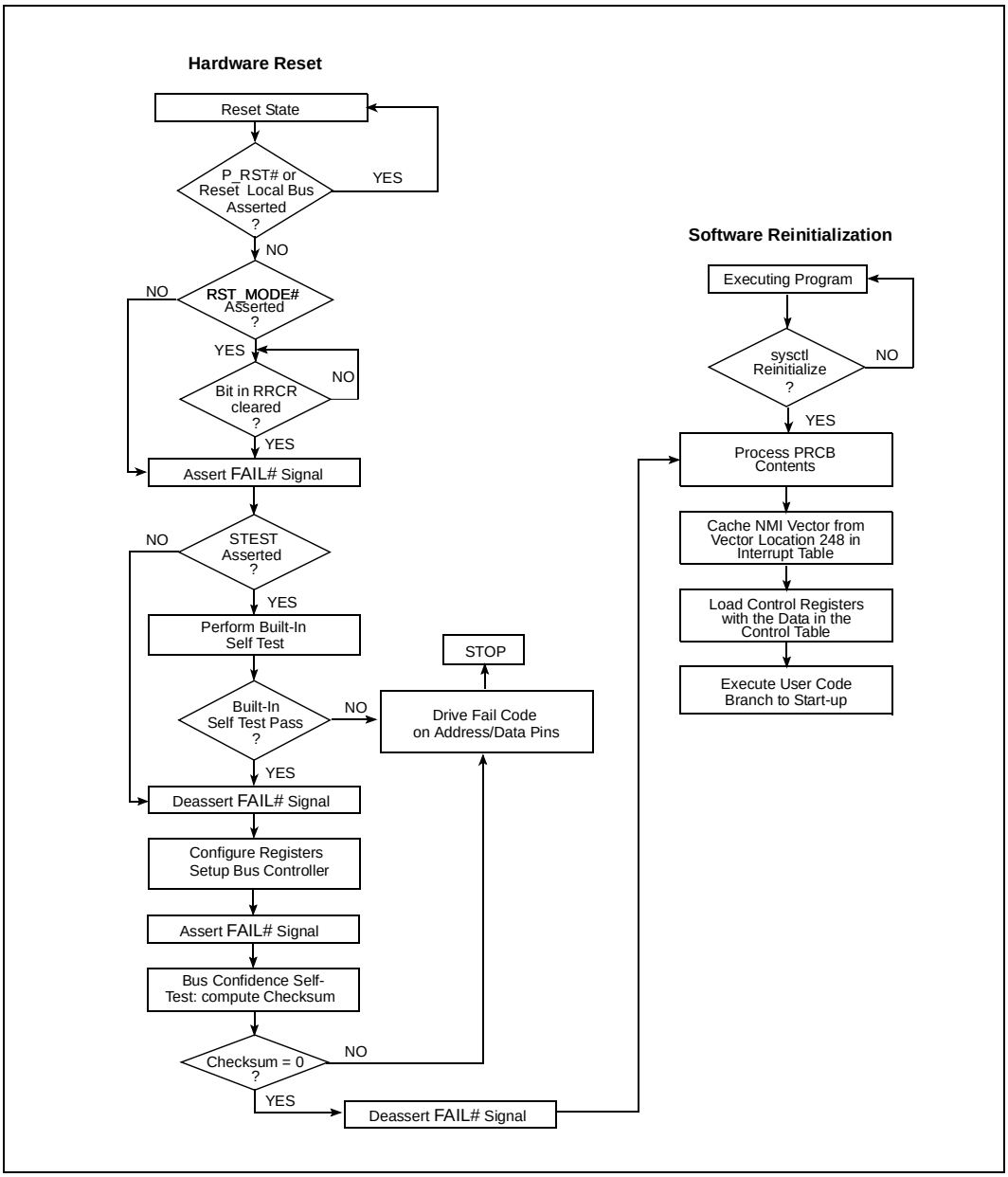
L_RST# asserts after P_RST# is asserted. L_RST# deasserts after P_RST# deasserts.

12.3 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 12-2 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.

Figure 12-2. Processor Initialization Flow



12.3.1 Self Test Function (STEST, FAIL#)

As part of initialization, the 80960VH 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.

12.3.1.1 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#

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.

12.3.1.2 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 FFFF 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.

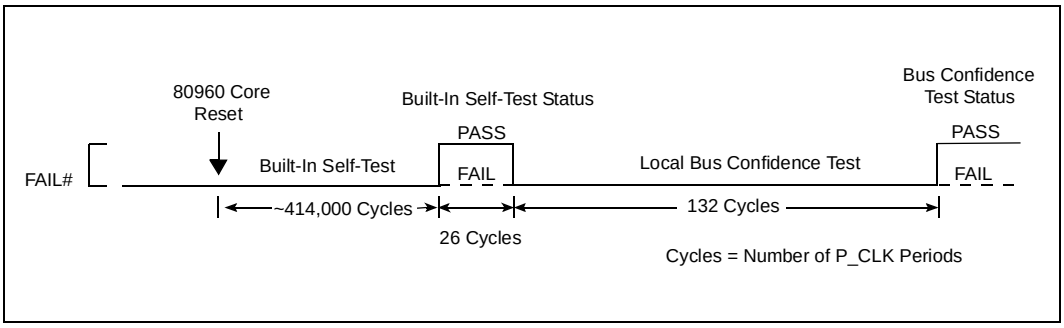
12.3.1.3 The Fail Signal (FAIL#)

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 12-3):

- 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 12.3.1.4, "IMI Alignment Check and Core Processor Error" on page 12-9 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 12-3, all transitions on the FAIL# signal are relative to P_CLK as described in the 80960VH Datasheet.

Figure 12-3. FAIL# Timing



12.3.1.4 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.

12.3.1.5 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 12-3 and Table 12-4.

Table 12-3. 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 12-4. 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

12.4 Initial Memory Image (IMI)

The IMI comprises the minimum set of data structures that the processor needs to initialize. As shown in Figure 12-4, 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 12.3.1.3, "The Fail Signal (FAIL#)" on page 12-8.

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)" on page 3-5. Recall that the supervisor stack pointer is located in the preamble of the system procedure table at byte offset

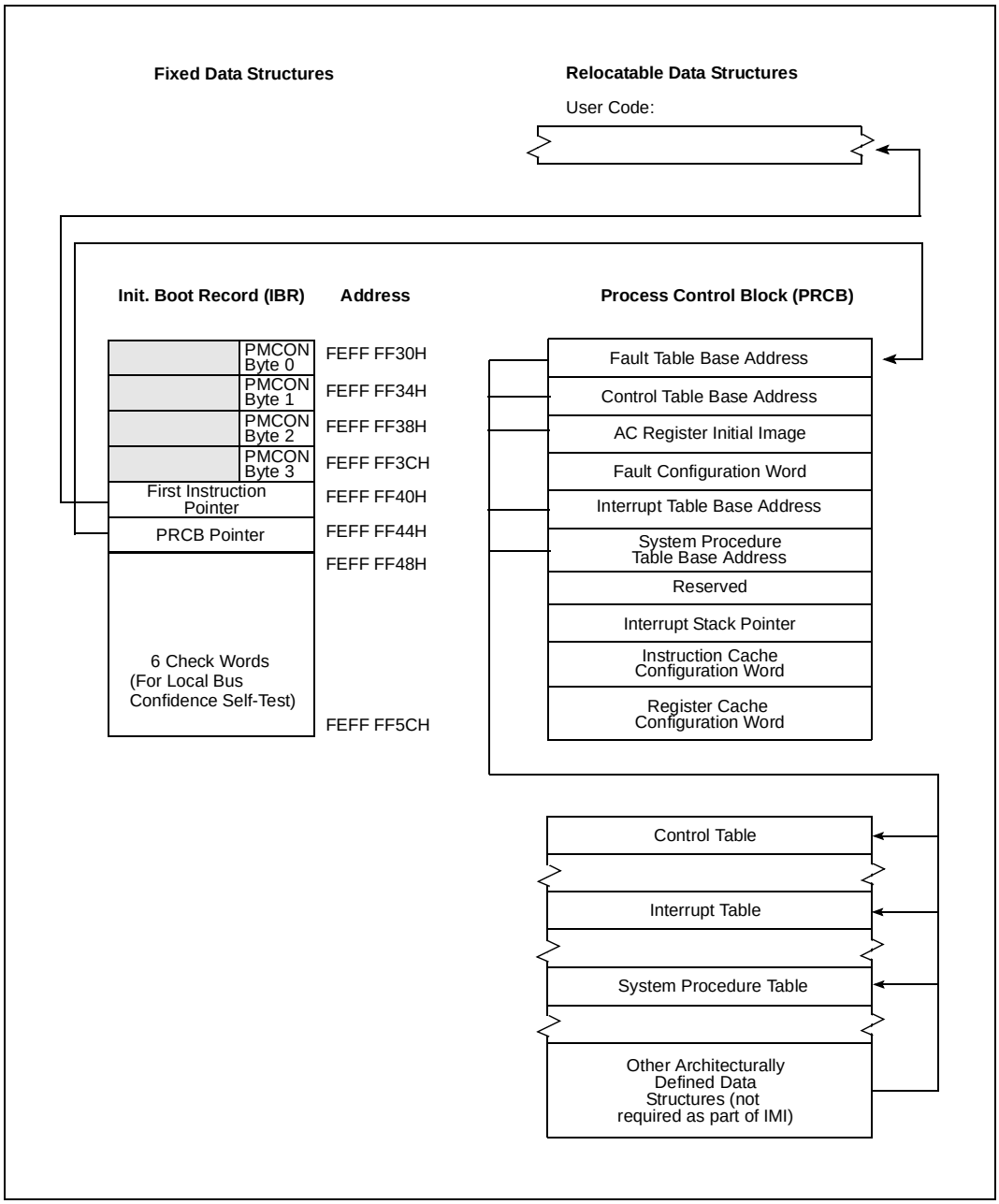
12 from the base address. The system procedure table base address is programmed in the PRCB. Consult Section 7.5.1, “System Procedure Table” on page 7-13 for the format of the system procedure table.

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 80960VH’s local bus RAM.

Figure 12-4. Initial Memory Image (IMI) and Process Control Block (PRCB)



12.4.1 Initialization Boot Record (IBR)

The initialization boot record (IBR) is the primary data structure required to initialize the 80960VH processor. The IBR is a 12-word structure which must be located at address FEFF FF30H (see Table 12-5). 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 12-5. 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 80960VH, 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 12-1.

Example 12-1. 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 */

}
```

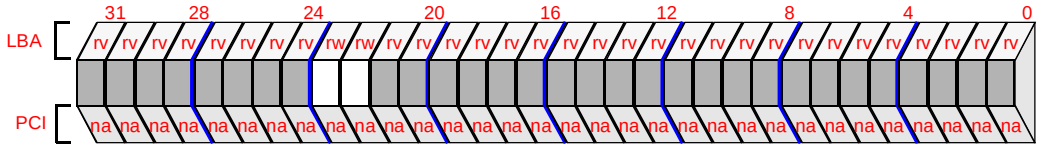
The processor initializes the DLMCON.dcen bit to 0 to disable data caching. The remainder of the assembled word is used to initialize PMCON14_15. In conjunction with this step, the processor clears the bus control table valid bit (BCON.ctv), to ensure for the remainder of initialization that every bus 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 14, “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 12-1. When the checksum calculation equals zero, then the confidence test of the local bus passes.

Table 12-6 further describes the IBR organization.

Table 12-6. PMCON14_15 Register Bit Description in IBR



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.

12.4.2 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 12-7.

Table 12-7. PRCB Configuration (Sheet 1 of 2)

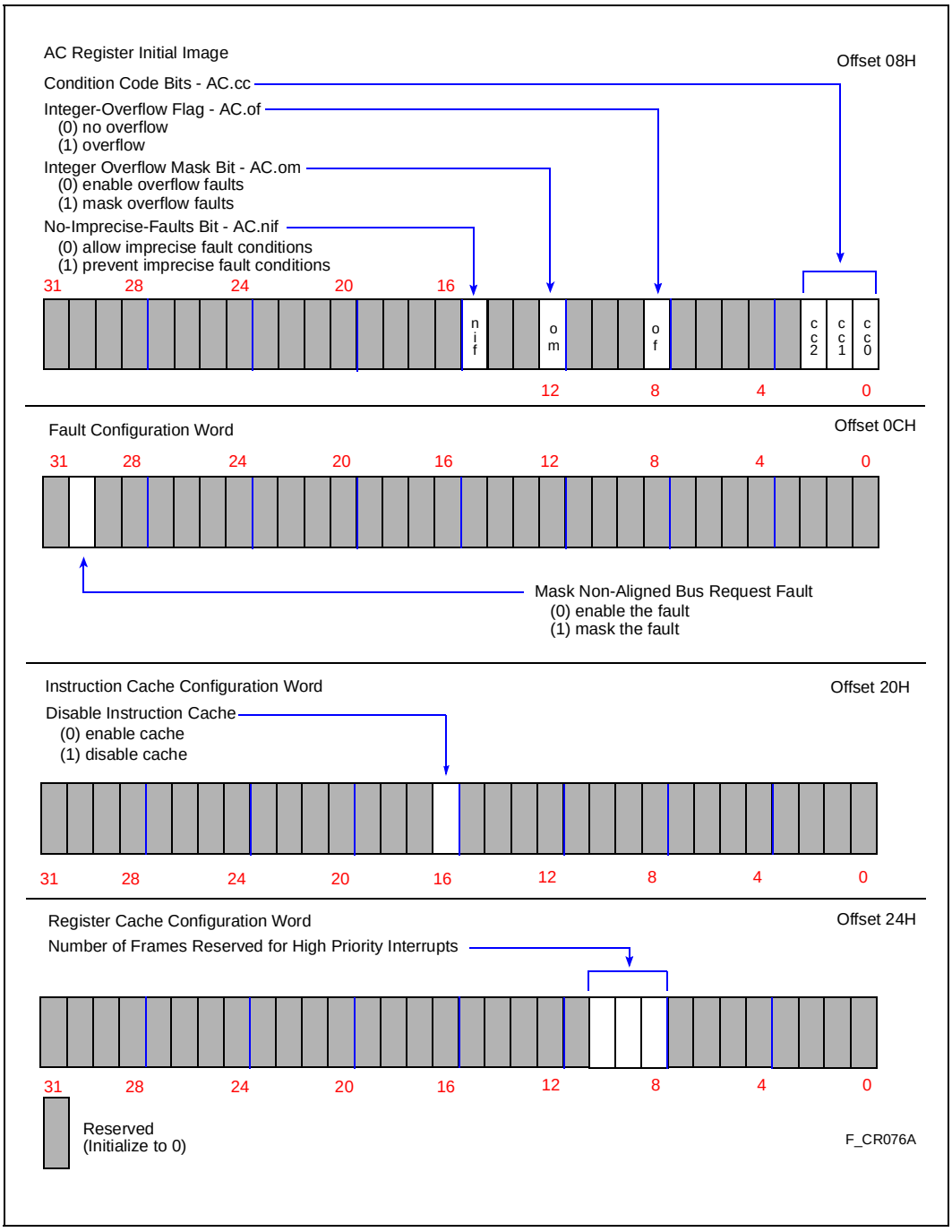
Physical Address	Description
PRCB POINTER + 00H	Fault Table Base Address
PRCB POINTER + 04H	Control Table Base Address
PRCB POINTER + 08H	AC Register Initial Image
PRCB POINTER + 0CH	Fault Configuration Word
PRCB POINTER + 10H	Interrupt Table Base Address
PRCB POINTER + 14H	System Procedure Table Base Address

Table 12-7. PRCB Configuration (Sheet 2 of 2)

Physical Address	Description
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 12-8 show these configuration words.

Table 12-8. Process Control Block Configuration Words



12.4.3 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 12-2. 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;
```

12.4.3.1 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.

12.4.3.2 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.

12.4.3.3 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.

12.4.3.4 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” on page 4-2.

12.4.4 Control Table

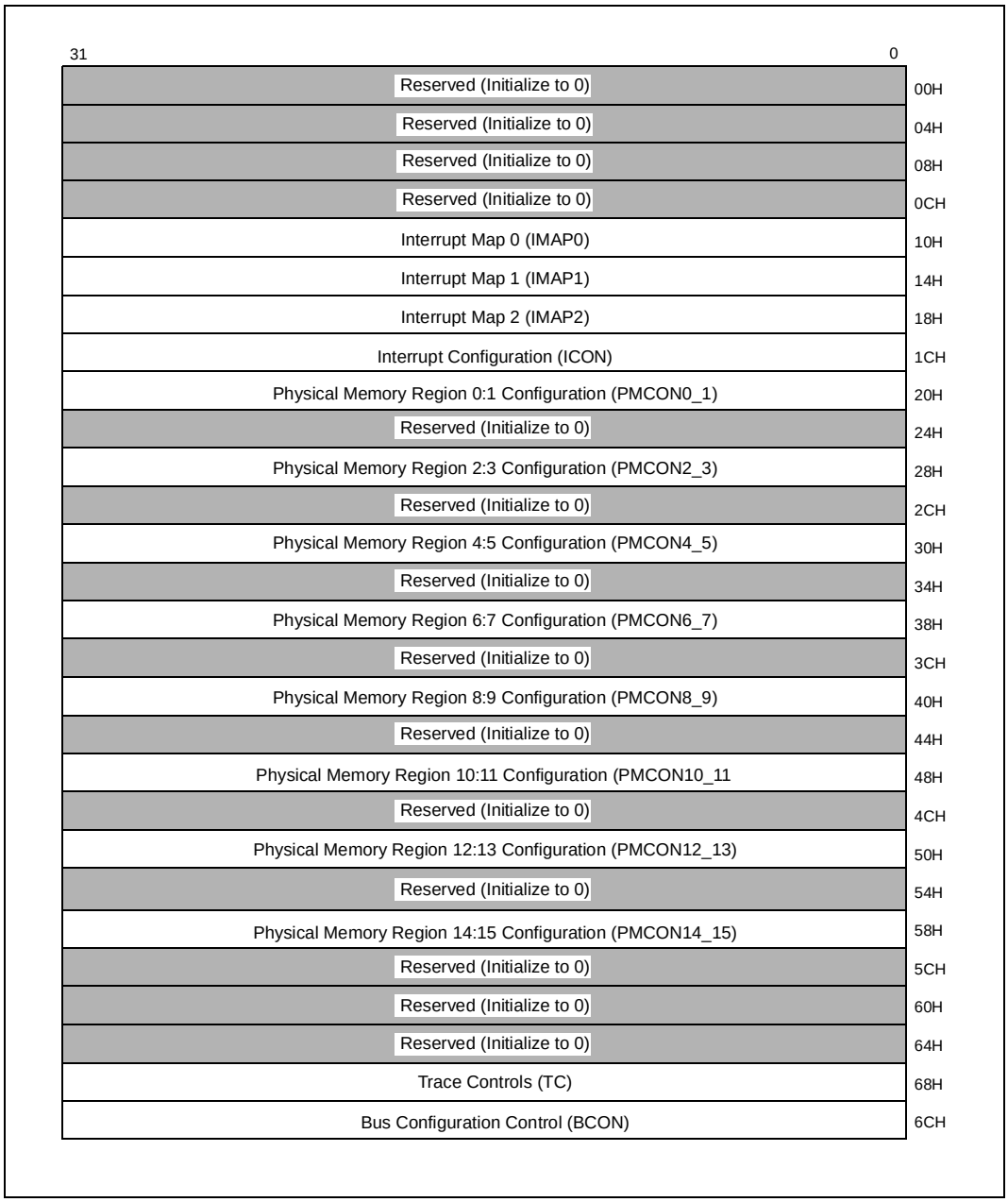
The control table is the data structure that contains the on-chip control registers values. It is automatically loaded during initialization and must be completely constructed in the IMI. Figure 12-5 shows the Control Table format.

For register bit definitions of the on-chip control table registers, see the following:

- IMAP — Table 8-9 through Table 8-11, Interrupt Map Register 2 – IMAP2 (page 8-27)
- ICON — Table 8-8. Interrupt Control Register – ICON (pg. 8-25)
- PMCON — Table 13-2. Physical Memory Control Registers – PMCON0:15 (pg. 13-4)
- TC — Figure 10-1. i960® VH processor Trace Controls Register – TC (pg. 10-2)

- BCON — Table 13-3. Bus Control Register – BCON (pg. 13-5)

Figure 12-5. Control Table

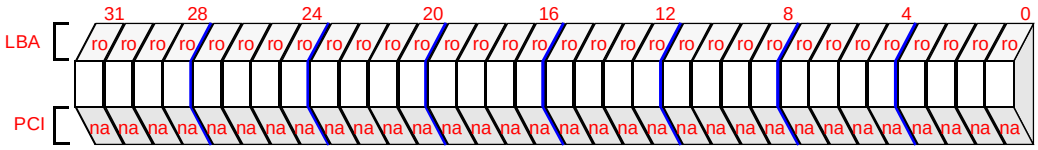


12.5 Device Identification on Reset

During the manufacturing process, values characterizing the 80960VH type and stepping are programmed into the memory-mapped registers. The 80960VH 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 12-9 and Table 12-10 describe the fields of the two Device IDs. During initialization, the PDIDR is placed in g0.

Table 12-9. Processor Device ID Register - PDIDR

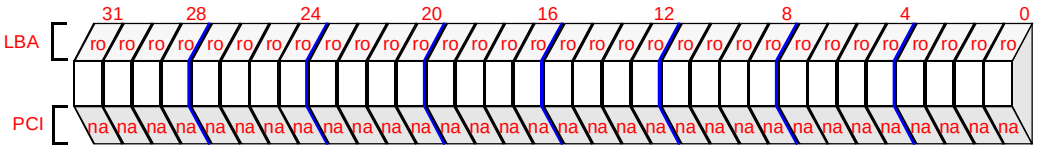


LBA: 1710H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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 register varies with stepping. Refer to the *i960® VH Processor Specification Update* (Intel Order # 273174-001) for the correct value.

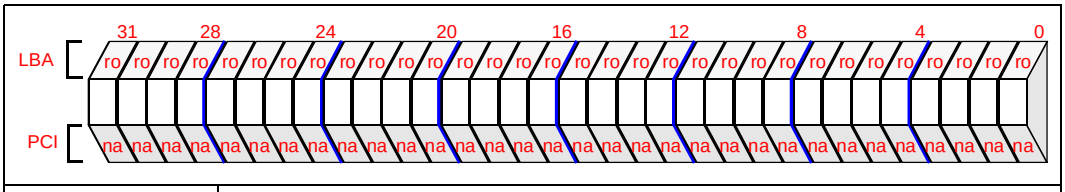
Table 12-10. i960® Core Processor Device ID Register - DEVICEID (Sheet 1 of 2)



LBA: FF00 8710 H **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.

Table 12-10. i960® Core Processor Device ID Register - DEVICEID (Sheet 2 of 2)



LBA: FF00 8710 H **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
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 register varies with stepping. Refer to the *i960® VH Processor Specification Update* (Intel Order # 273174-001) for the correct value.

12.6 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 Section 6.2.67, “sysctl” on page 6-104 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 12.4.2, “Process Control Block – PRCB” on page 12-15.

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

12.7 System Requirements

The following sections discuss generic hardware requirements for a system built around the 80960VH. This section describes electrical characteristics of the processor's interface to the external circuit, including the P_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.

12.7.1 Clocking

The 80960VH has a single clock input (P_CLK) for control. All input/output timings are relative to P_CLK.

The range of operation for all PCI clocks is 0 to 33 MHz. The 80960VH 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 80960VH Datasheet. P_CLK input capacitance is minimal; for this reason, it may be necessary to terminate the P_CLK circuit board traces at the processor to reduce overshoot and undershoot.

12.7.2 Output Clocks

The 80960VH 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, "I2C Clock Count Register – ICCR" on page 21-21.

12.7.3 Reset

There are multiple ways to reset the 80960VH. Reset is controlled either through external signals or control registers.

When the primary PCI bus reset signal P_RST# is asserted, the 80960VH:

- Resets the i960 core processor and the local bus.
- Resets all internal units, including the Core and Peripheral Control Unit.
- Asserts local bus reset.

Reset is also available through the Reset/Retry Control Register in the Core and Peripheral Control Unit:

- The Reset Local Bus bit in the Reset/Retry Control Register (RRCR) resets the i960 core processor and all units on the local bus. Before reset, the DMA channels and the ATU halt all PCI bus transactions. Software must ensure that the I^2C bus is 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 RRCR. The local bus reset does not reset the Core and Peripheral Control Unit or its configuration registers. All other configuration registers are reset.

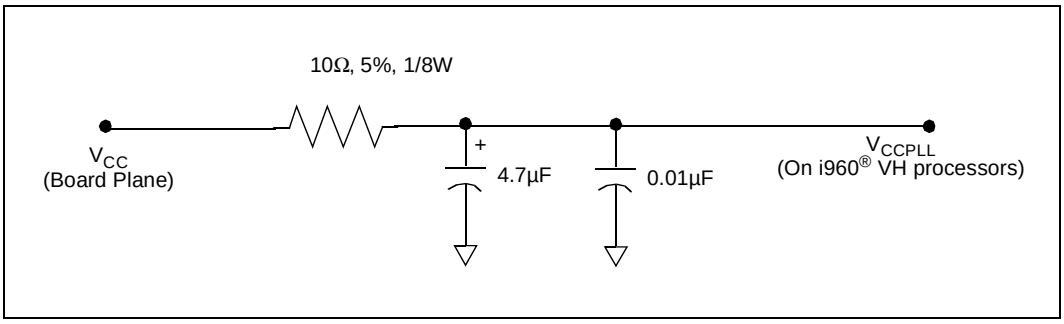
See Chapter 11, “Core and Peripheral Control Unit” for a full description of the Reset/Retry Control Register.

12.7.4 Power and Ground Requirements (V_{cc}, V_{ss})

The large number of V_{SS} and V_{CC} signals effectively reduce the impedance of power and ground connections to the chip and reduces transient noise induced by current surges. The 80960VH 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 80960VH Datasheet).

To reduce clock skew internal to the 80960VH, the V_{CCPLL} pins for the Phase Lock Loop (PLL) circuits are isolated on the pinout. The lowpass filter, as shown in Figure 12-6, 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 12-6. V_{CCPLL} Lowpass Filter



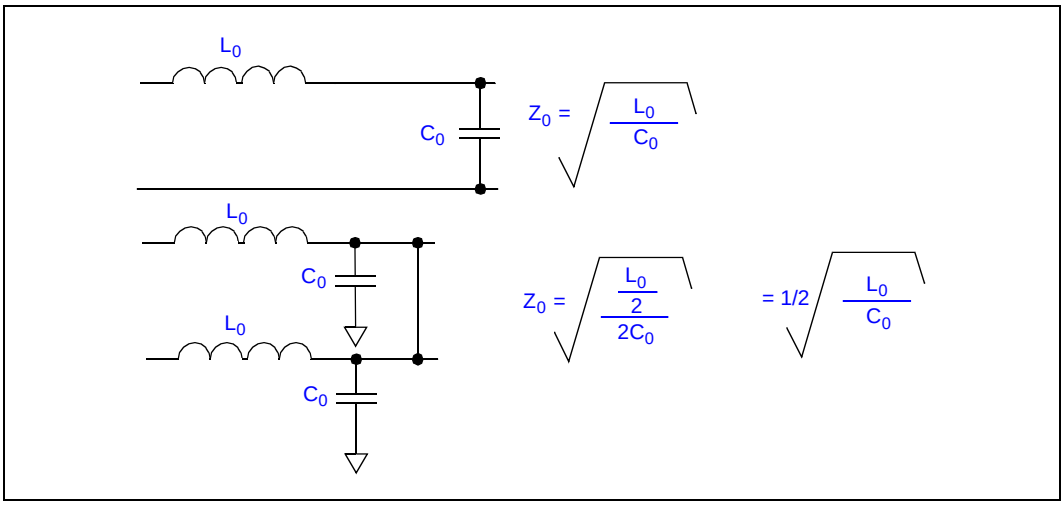
12.7.5 Power and Ground Planes

Power and ground planes must be used in 80960VH 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 12-7, 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 80960VH should be located at the center of the board to take full advantage of these planes, simplify layout and reduce noise.

Figure 12-7. Reducing Characteristic Impedance



12.7.6 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.

12.7.7 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.

12.7.8 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 80960VH, 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 12-8. 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 12-9). 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 12-8. Series Termination

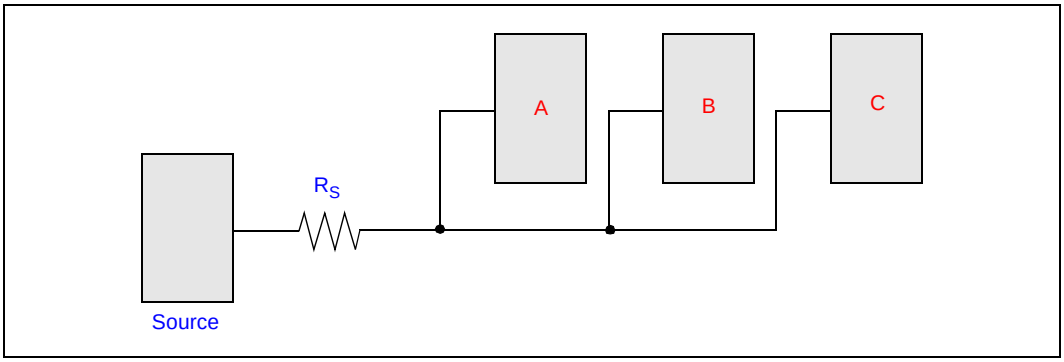
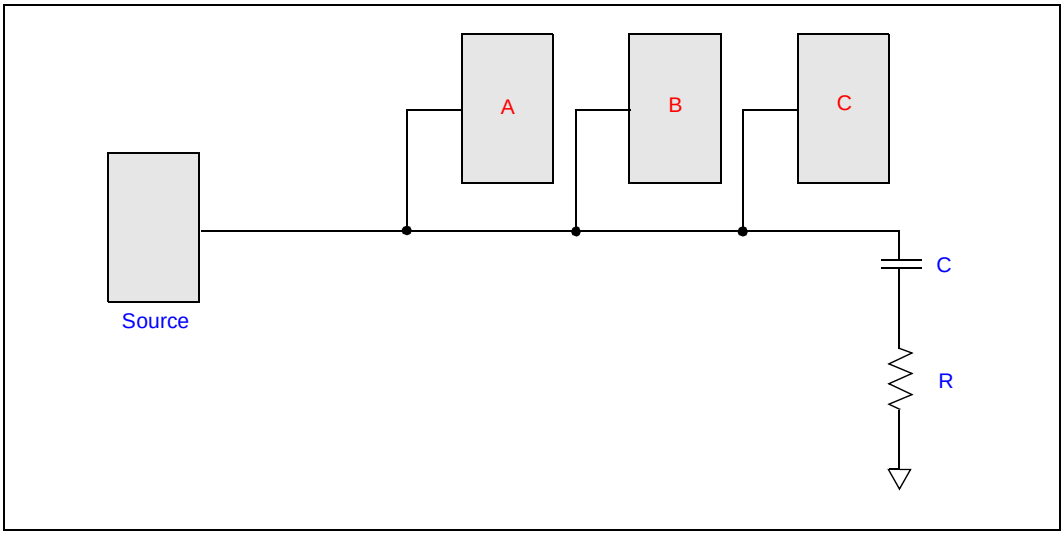


Figure 12-9. AC Termination



12.7.9 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 80960VH signal or a device connected to an 80960VH signal before applying power to the 80960VH itself.
- Prevent overshoot and undershoot on I/O pins by adding line termination and by designing to reduce noise and reflection on signal lines.

12.7.10 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:

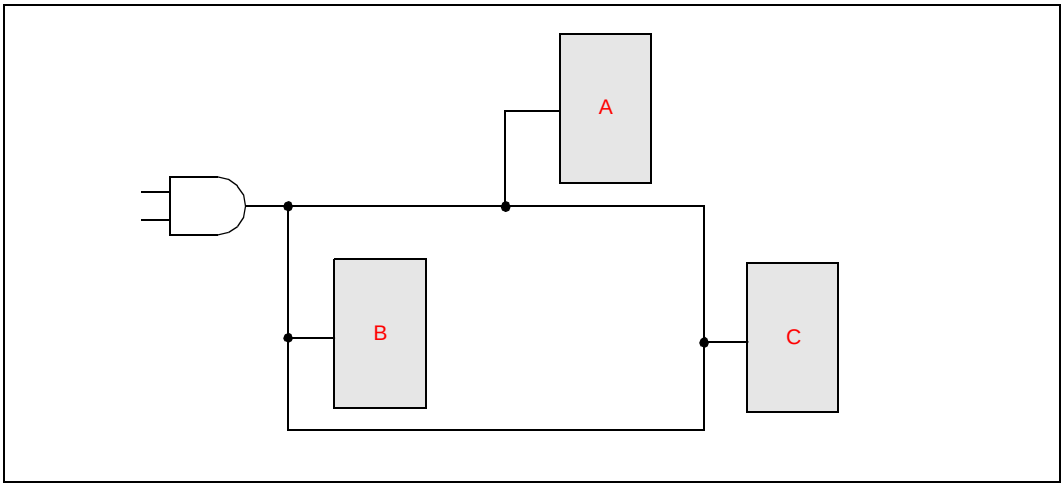
- Frequency Interference is the result of changing currents and voltages. The more frequent the changes, the greater the interference.
- Closeness-of-conductors Interference is due to electromagnetic and electrostatic fields whose effects are weaker further from the source.

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 12-10. 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.

Core Processor Local Bus Configuration 13

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.

13.1 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.

13.1.1 Physical Memory Attributes

The only programmable physical memory attribute for the i960® VH 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 80960VH provides one PMCON register for each region.The descriptions of the PMCON registers and instructions on programming them are found in Section 13.2, “Programming the Physical Memory Attributes (Pmcon Registers)” on page 13-3.

13.1.2 Logical Memory Attributes

The 80960VH 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 (for example, DRAM, SRAM) is cacheable or non-cacheable in the on-chip data cache.

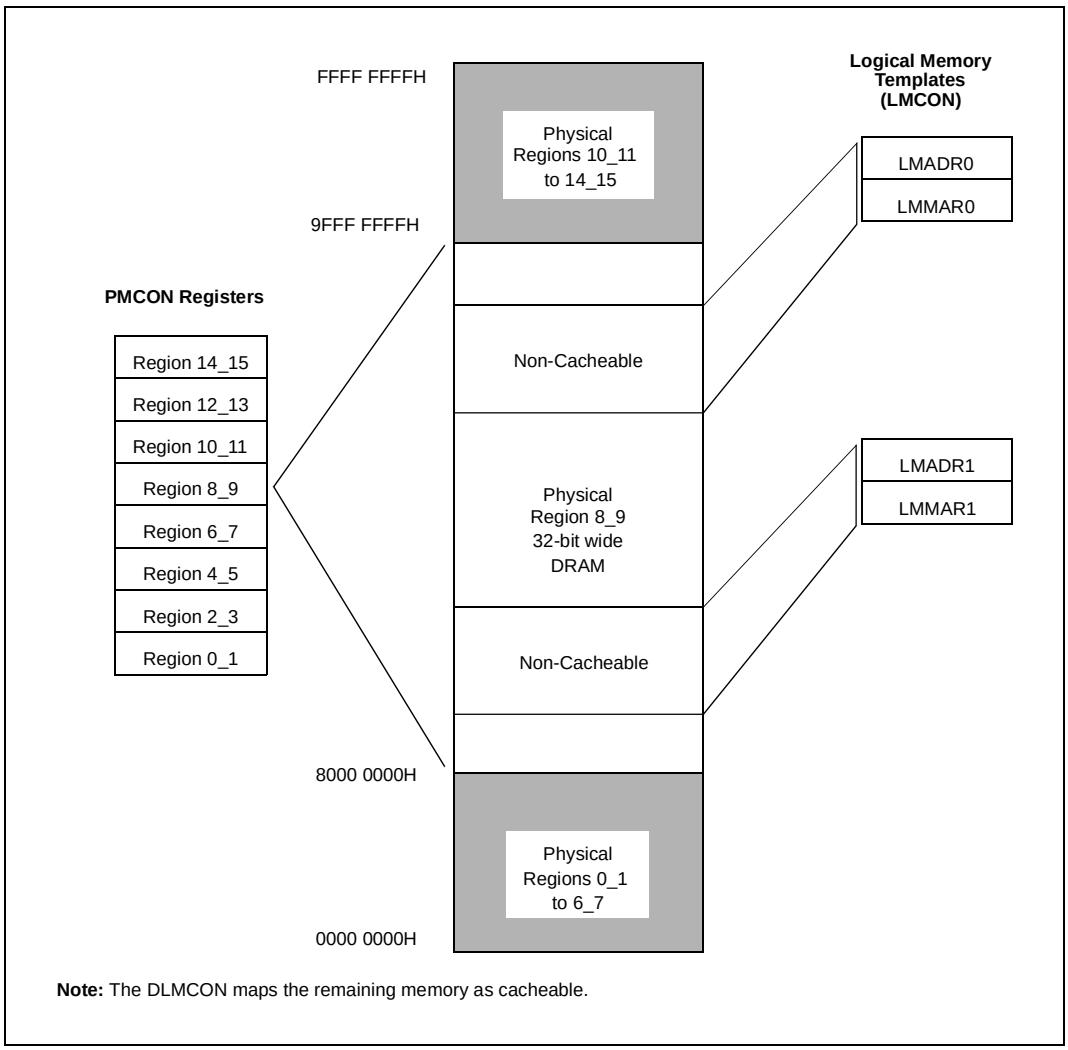
There are typically several different LMTs defined within a single memory subsystem. For example, data within one area of DRAM may be non-cacheable while data in another area is cacheable. Figure 13-1 shows the use of the Control Table (PMCON registers) with logical memory templates for a single DRAM 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 80960VH 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 13.5, “Programming The Logical Memory Attributes” on page 13-6.

Figure 13-1. PMCON and LMCON Example



Note: The DLMCON maps the remaining memory as cacheable.

13.2 Programming the Physical Memory Attributes (Pmcon Registers)

The Physical Memory Configuration registers, PMCON0_1 to PMCON14_15, are shown in Table 13-2. 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 13-1.

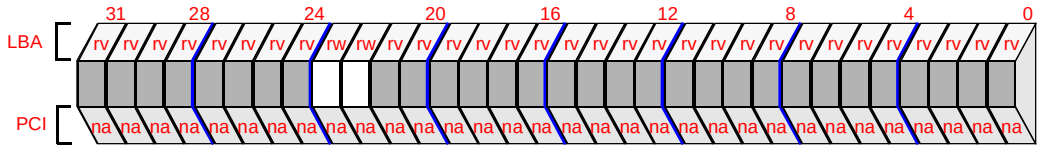
Table 13-1. PMCON Address Mapping

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 - 80960VH 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 - 80960VH outbound ATU translation windows[2] (See Figure 16-5., 80960 Local Bus Memory Map - Outbound Translation Window (pg. 16-10))
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 13-2. Physical Memory Control Registers – PMCON0:15



LBA: see Table 13-1 **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.

13.2.1 Local Bus Width

The local bus width for a region is controlled by the PMCON register. The operation of the 80960VH with different local bus width programming options is described in Section 14.3.4, “Bus Width” on page 14-6.

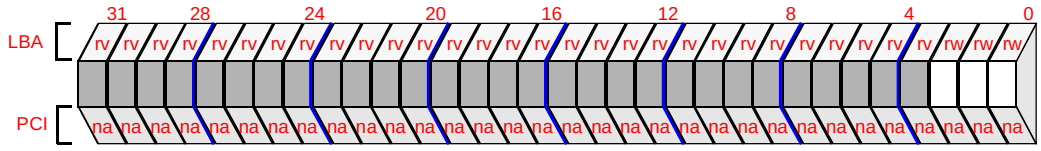
13.3 Physical Memory Attributes At Initialization

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 Section 12.4, “Initial Memory Image (IMI)” on page 12-10.

13.3.1 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 13-3. Bus Control Register – BCON



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 0000 H	Reserved.
02	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
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	Configuration Entries in Control Table Valid (0) = PMCON entries not valid, default to PMCON14_15 setting (1) = PMCON entries valid

13.4 Boundary Conditions For Physical Memory Regions

The following sections describe the operation of the PMCON registers during conditions other than “normal” accesses.

13.4.1 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.

13.4.2 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.

13.4.3 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.

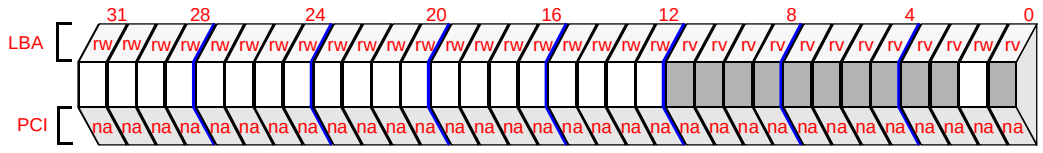
13.5 Programming The Logical Memory Attributes

Bit field definitions for Logical Memory Address Registers - LMADR1:0 and LMMR1:0 registers are shown in Table 13-4. LMCON registers reside within the i960 core processor memory-mapped control register space. (See Appendix C, “Memory-Mapped Registers”.)

13.5.1 Logical Memory Address Registers - LMADR0:1

The LMADR1:0 registers define the address for the logical data templates and template caching.

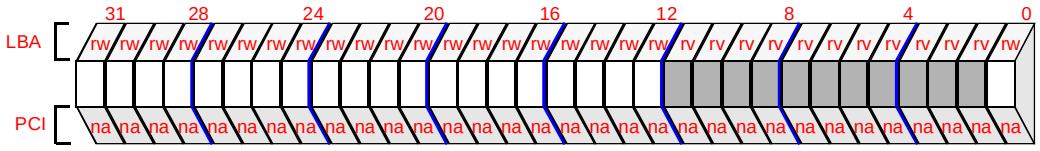
Table 13-4. Logical Memory Address Registers – LMADR0:1



LBA: CH0-8108H CH1-8110H **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	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 13-5. Logical Memory Mask Registers – LMMR0:1

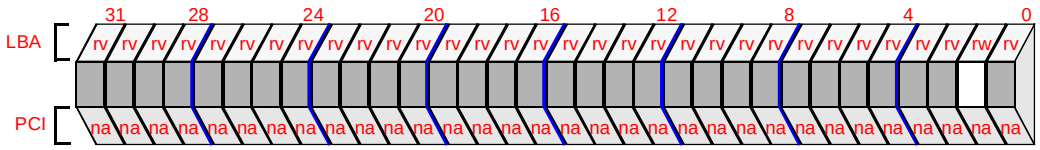


LBA: CH0-810CH **PCI:** CH1-8114H NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	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 13-6. 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 13-6. Default Logical Memory Configuration Register – DLMCON



LBA: 8100H **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 0000 H	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.

13.5.2 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...`

13.5.3 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.

13.5.4 Enabling the Logical Memory Template

LMMRx bit 0 activates the logical data template in the LMMR register for the programmed range.

13.5.5 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.

13.5.6 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.

13.5.6.1 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 80960VH 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.

13.5.6.2 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.

13.5.6.3 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.

13.5.7 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.

Local Bus 14

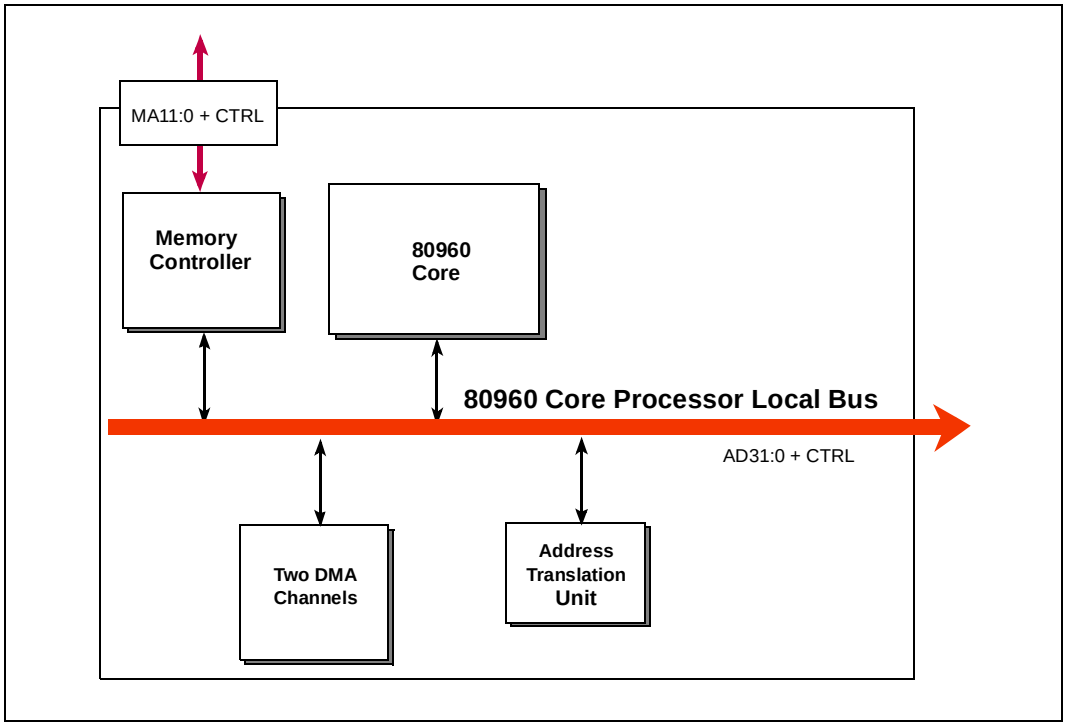
This chapter describes the bus interface of the i960® VH processor. It explains the following:

- 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 80960VH's address/data bus.

For information on programmable bus configuration, refer to Chapter 13, "Core Processor Local Bus Configuration".

Figure 14-1. The Local Bus



14.1 Overview

The local bus is the data communication path between the various components of an 80960VH 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 ATU and DMA units. While the i960 core processor is limited to a burst length of four transfers, the ATU 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 (P_CLK); therefore, the memory system control logic can be implemented as state machines.

Users who are familiar with i960 JT processor should note the following differences in functionality between the i960 JT processor and the 80960VH. See Table 14-1.

Table 14-1. Differences Between 80960JT and 80960VH Local Buses

Topic	80960JT	80960VH
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 ATU 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.

14.1.1 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.

14.2 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.

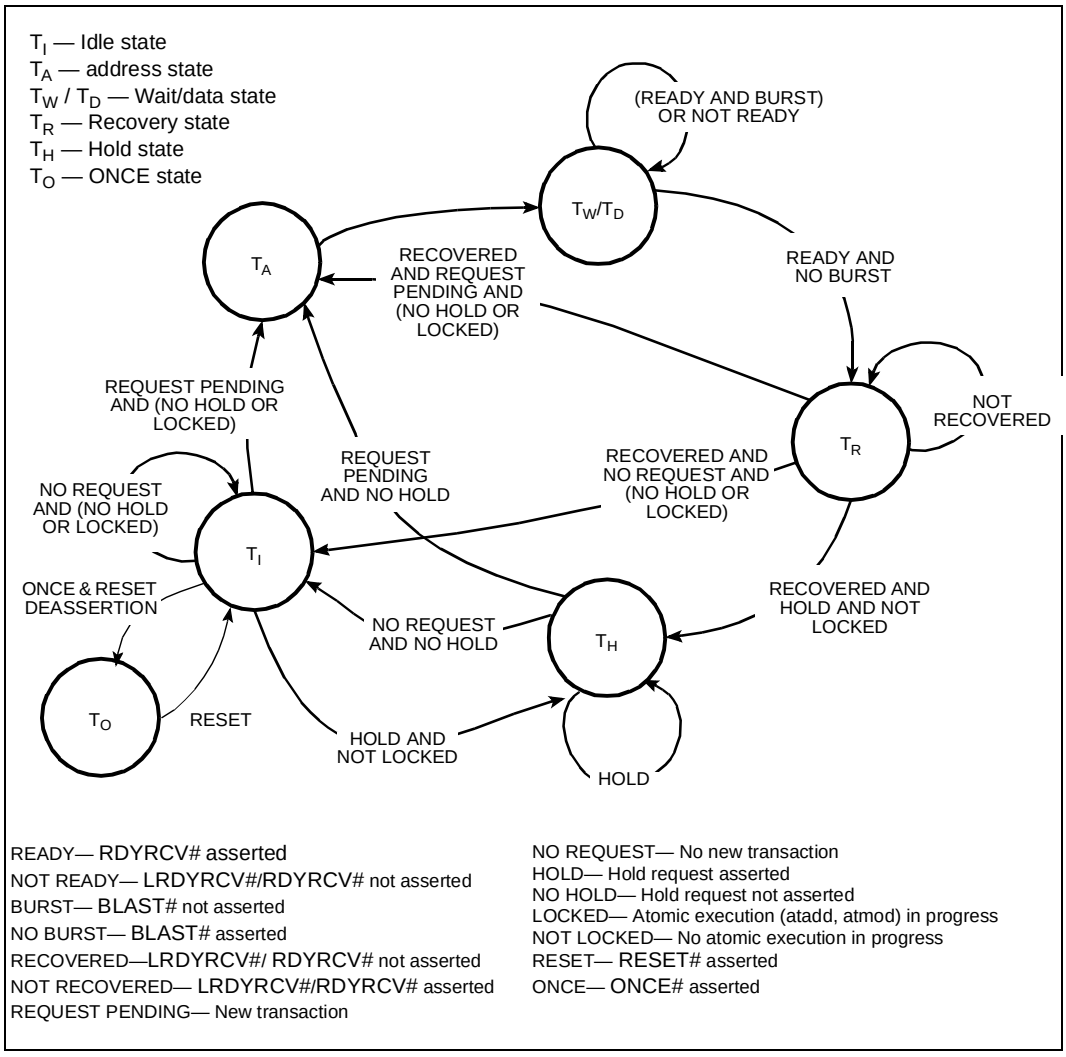
The bus occupies the idle (T_I) state when no address/data transactions are in progress and when P_RST# is asserted. When the processor needs to initiate a bus access, it enters the T_A state to transmit the address.

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.

Figure 14-2. Bus States with Arbitration



14.3 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 *80960VH Microprocessor* Data Sheets.

14.3.1 Clock Signal

The P_CLK input signal is the reference for all 80960VH 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 P_CLK. The processor asserts ALE directly from the rising P_CLK edge at the beginning of a TA state but deasserts them approximately half way through the state instead of the next rising P_CLK edge. All transitions on DT/R# are also referenced to a point halfway through the TA state instead of rising P_CLK edges.

14.3.2 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 14.3.4, “Bus Width” on page 14-6. These signals increment during burst accesses. The memory controller increments the addresses during bursts. See Chapter 15, “Memory Controller” for more information.

14.3.3 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 80960VH. 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.

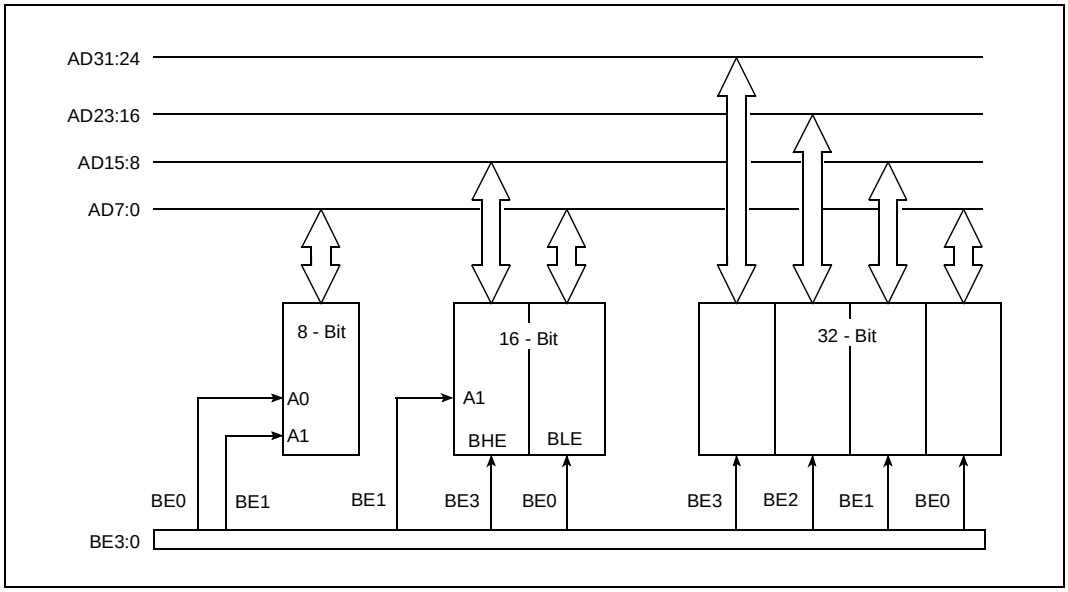
A bus access may be either non-burst or burst. A non-burst access ends after one data transfer to a single location. The processor asserts BLAST# (burst last) to indicate the last data cycle of an access in both burst and non-burst situations.

All 80960VH wait states to the local bus are controlled by either LRDYRCV# or RDYRCV#. See Section 14.3.7.1, “Recovery States” on page 14-19 for a description of these signals.

14.3.4 Bus Width

Each region's data bus width is programmed in a Physical Memory Region Configuration (PMCON) register (see Chapter 13). 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 14-3, 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 14-3. 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 80960VH 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.

Table 14-2. 8-Bit Bus Width Byte Enable Encodings

Byte	BE3# (Not Used)	BE2# (Not Used)	BE1# (Used as A1)	BE0# (Used as A0)
0	1	1	0	0
1	1	1	0	1
2	1	1	1	0
3	1	1	1	1

Table 14-3. 16-Bit Bus Width Byte Enable Encodings

Byte	BE3# (Used as BHE#)	BE2# (Not Used)	BE1# (Used as A1)	BE0# (Used as BLE#)
0,1	0	1	0	0
2,3	0	1	1	0
0	1	1	0	0
1	0	1	0	1
2	1	1	1	0
3	0	1	1	1

Table 14-4. 32-Bit Bus Width Byte Enable Encodings

Byte	BE3#	BE2#	BE1#	BE0#
0,1,2,3	0	0	0	0
0,1	1	1	0	0
2,3	0	0	1	1
0	1	1	1	0
1	1	1	0	1
2	1	0	1	1
3	0	1	1	1

14.3.5 Basic Bus Accesses

The basic transaction is a read or write of one data word. The first half of Figure 14-4 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 80960VH 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 14-4 shows LRDYRCV#/RDYRCV# asserted, so this state is a data state and the processor latches data on a rising P_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 14.4, “Bus and Control Signals During Recovery and Idle States” on page 14-22 for the values that are driven during T_I.

Figure 14-4. Non-Burst Read and Write Transactions Without Wait States, 32-Bit Bus

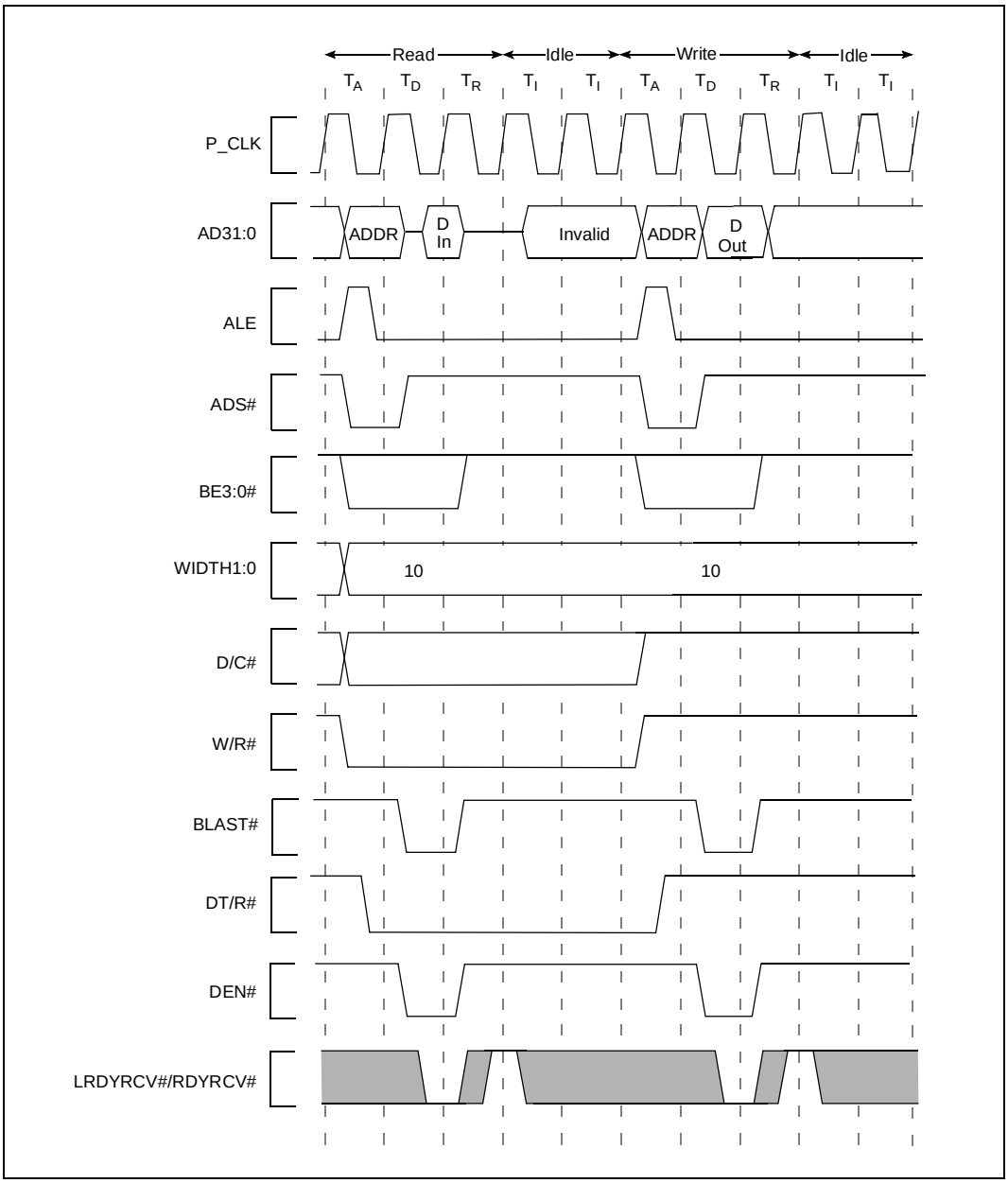


Figure 14-4 also shows a typical timing diagram for a non-burst, 32-bit write transaction. For the write operation, W/R# and DT/R# are high to denote the direction of the data flow. The D/C# signal is high since instruction code cannot be written. During the T_W/T_D state, the processor drives data on 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 14.4, “Bus and Control Signals During Recovery and Idle States” on page 14-22 for details.

14.3.6 Burst Transactions

A burst access is an address cycle followed by multiple data transfers. The 80960VH uses burst transactions to optimize local bus bandwidth. Burst transactions can be initiated by the i960 core processor, the ATU and the DMA units. Burst transactions initiated by the i960 core processor have the same burst length and alignment rules as the i960 JT processor. However, burst transactions initiated by the ATU 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 JT processor apply. If devices connected to the local bus are targeted by either the ATU or the DMA units, then those devices must support the additional local bus optimizations added by those units.

14.3.6.1 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 14-5 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 14-6 through Table 14-8 list all possible combinations of bus accesses resulting from aligned and unaligned requests. Figure 14-5 and Figure 14-6 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 12.4.2, “Process Control Block – PRCB” on page 12-15.

Table 14-5. 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 14-6. 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 14-7. 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 14-8. i960® Core Processor Summary of *n*-Word Load and Store Accesses (*n* = 1, 2, 3, 4) (Sheet 1 of 2)

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

Table 14-8. i960® Core Processor Summary of *n*-Word Load and Store Accesses (*n* = 1, 2, 3, 4) (Sheet 2 of 2)

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)
+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 14-5. i960® Core Processor Summary of Aligned and Unaligned Accesses (32-Bit Bus)

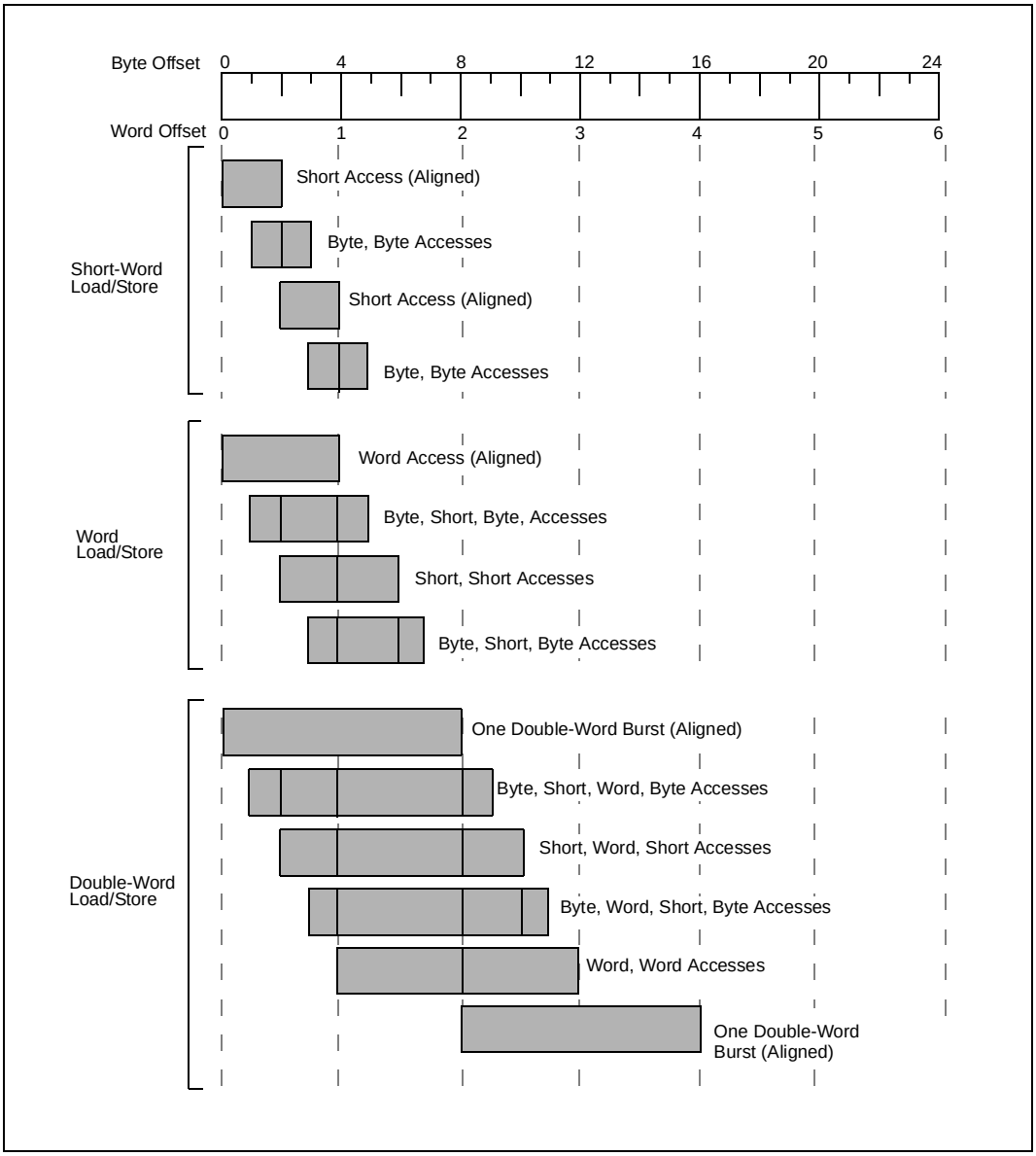


Figure 14-6. i960® Core Processor Summary of Aligned and Unaligned Accesses (32-Bit Bus) (Continued)

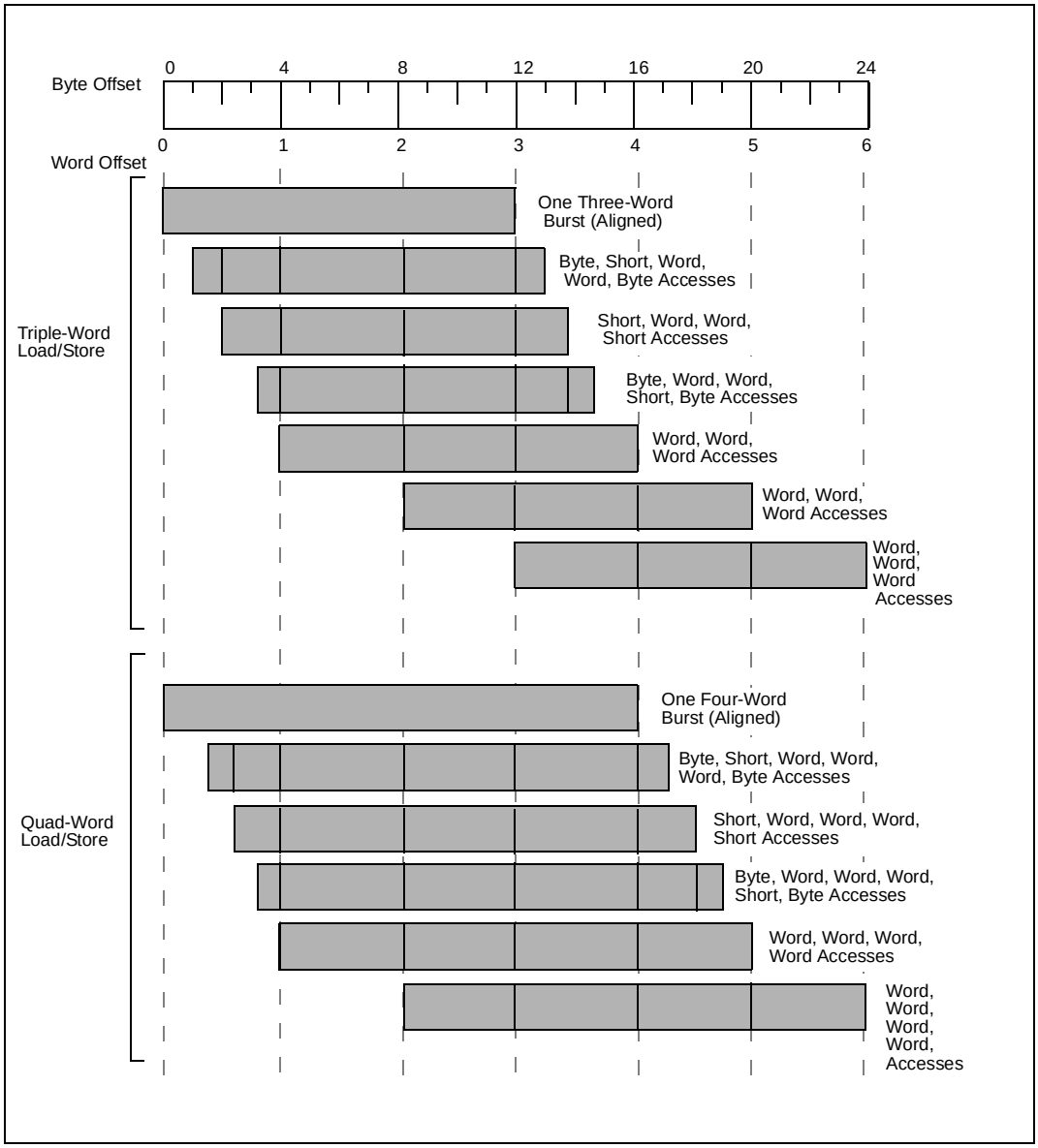


Figure 14-7. Burst Read and Write Transactions w/o Wait States, 8-bit Bus

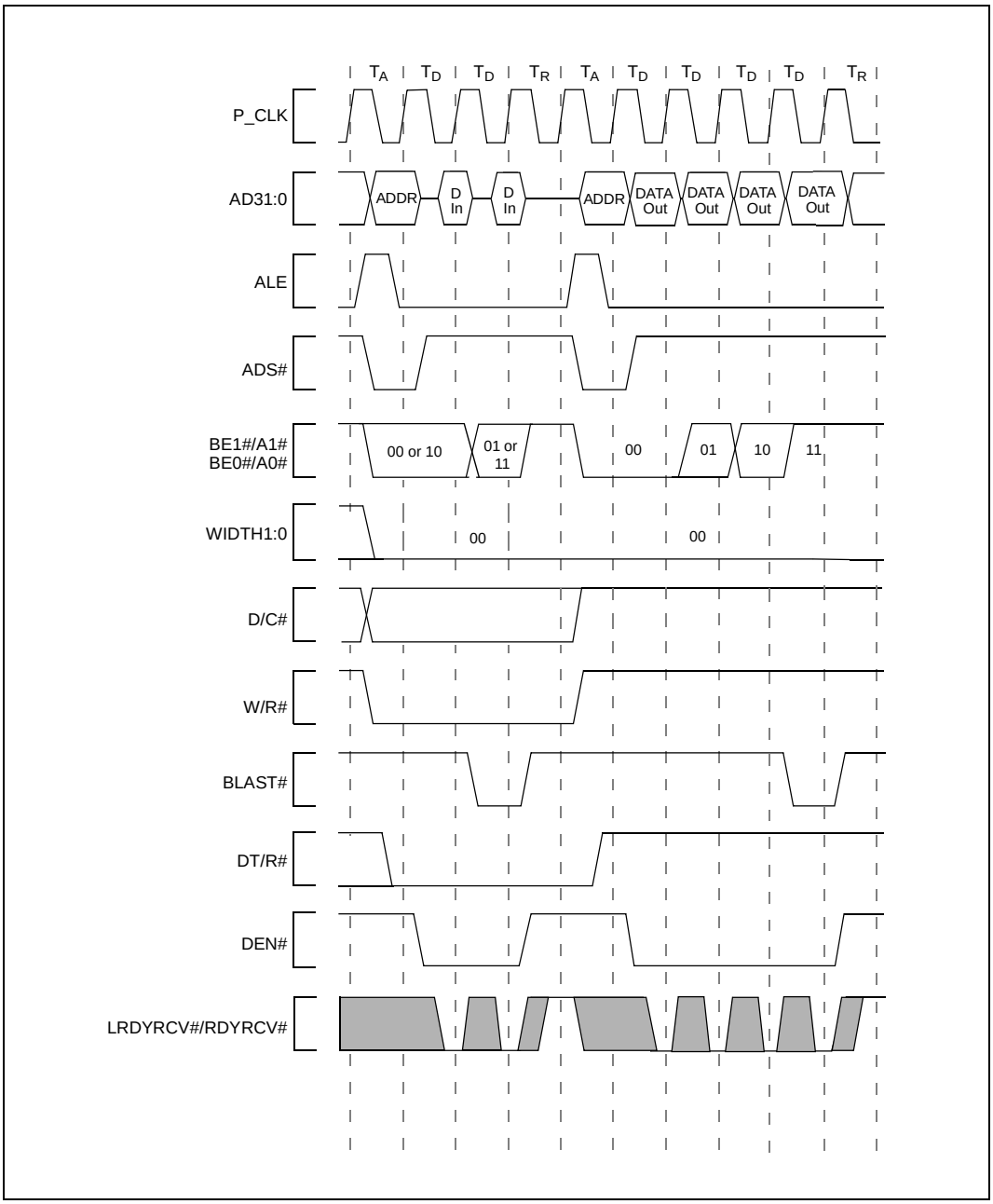
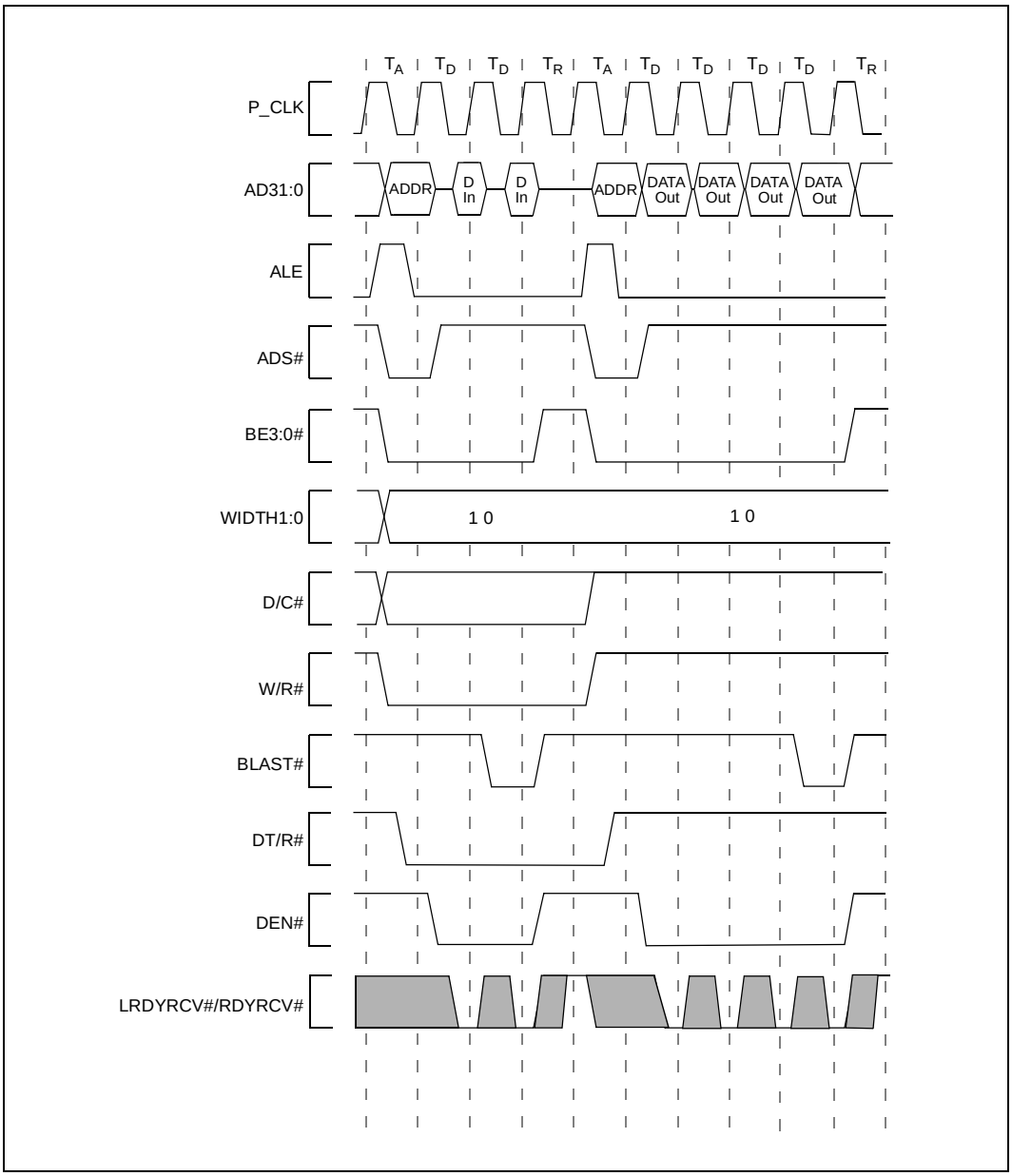


Figure 14-8. Burst Read and Write Transactions w/o Wait States, 32-bit Bus



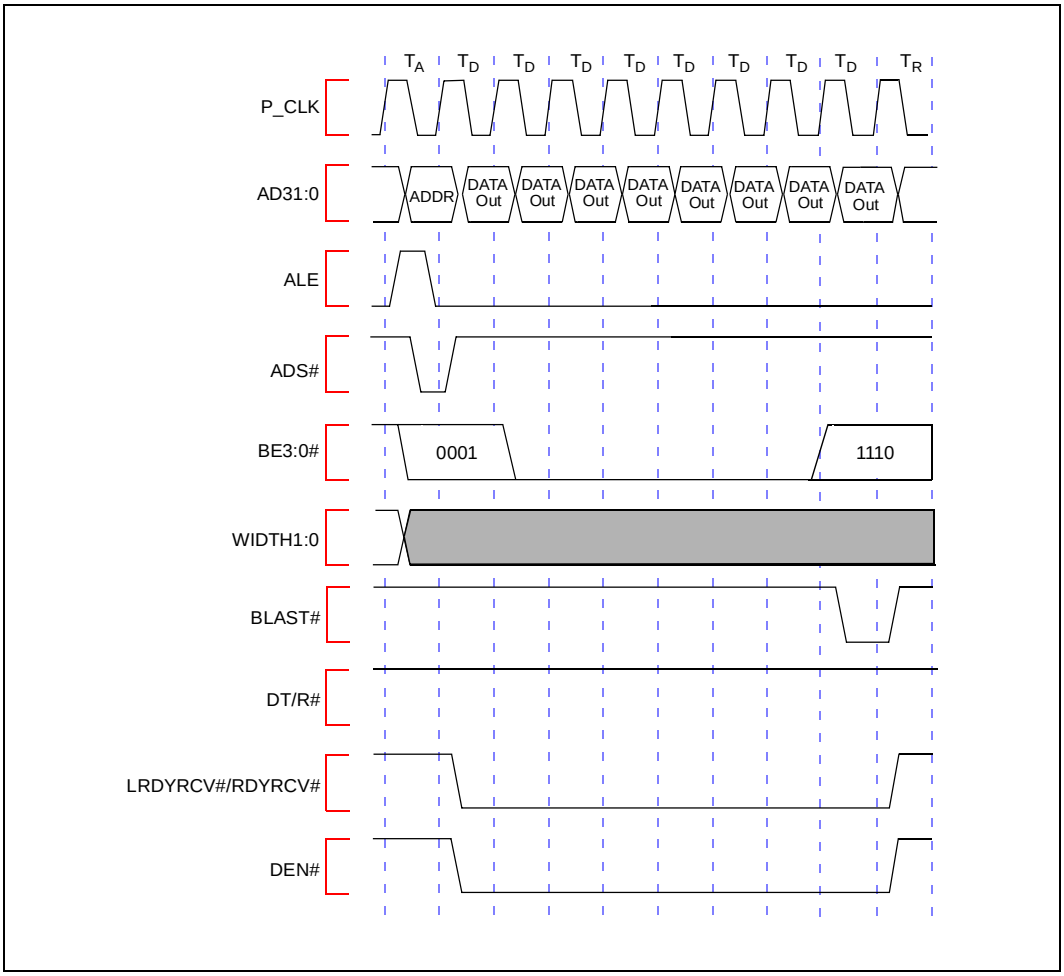
14.3.6.2 ATU and DMA Burst Transactions

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 80960VH 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 14-9 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 14-9. ATU or DMA 7-Word Unaligned Burst Transfer



14.3.7 Wait States

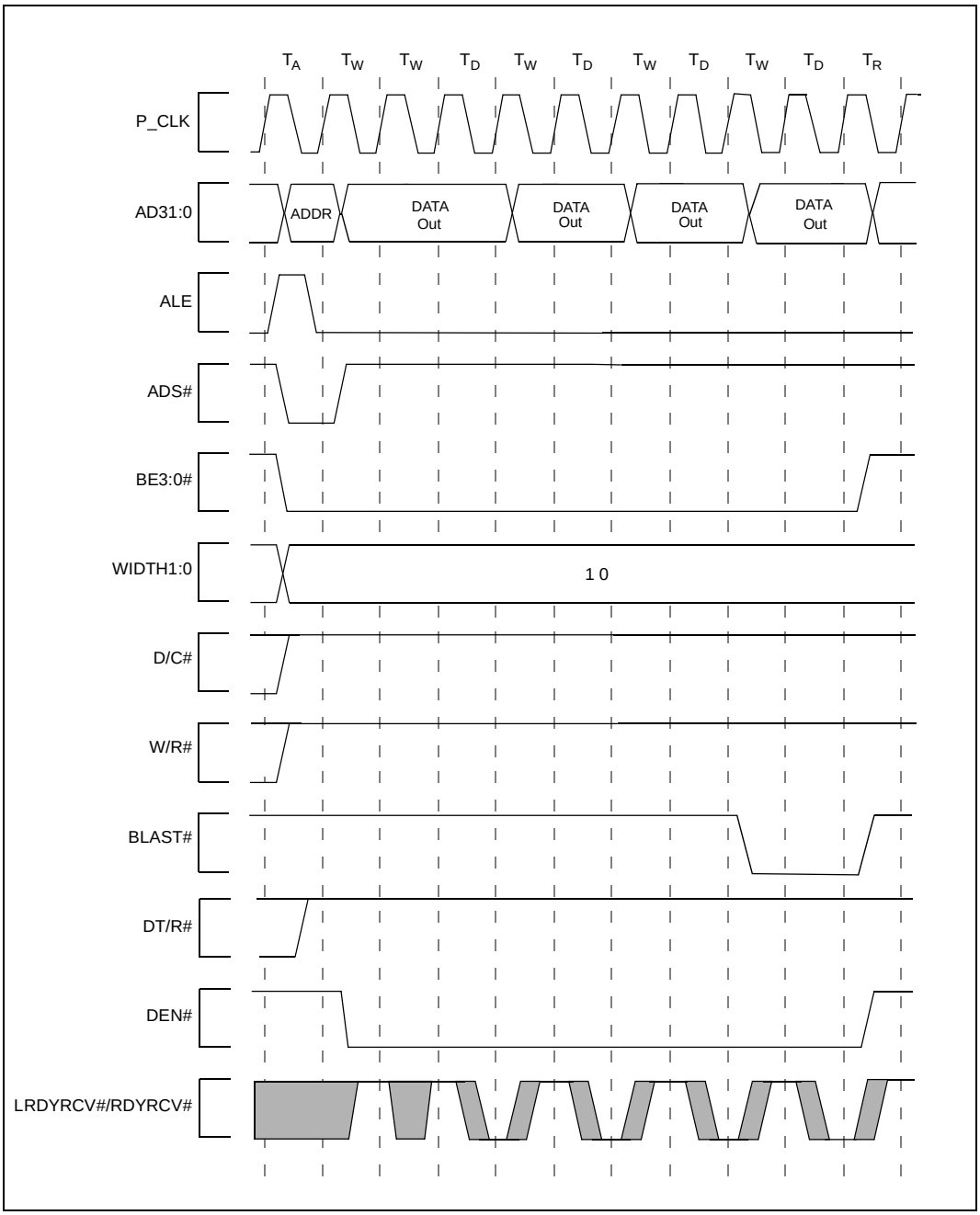
Wait states lengthen the processor's bus cycles, allowing data transfers with slow memory and I/O devices. The 80960VH supports three types of wait states: *address-to-data*, *data-to-data* and *turnaround* or *recovery*. All three types are controlled through the processor's LRDYRCV#/RDYRCV# signal. RDYRCV# is a synchronous input.

The processor's bus states follow the state diagram in Figure 14-2. 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 80960VH for accesses by the memory controller and does not have to be generated externally.

Figure 14-10 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 14-10. Burst Write Transactions With 2,1,1,1 Wait States, 32-bit Bus



14.3.7.1 Recovery States

The state following the last data transfer of an access is a recovery (T_R) state. By default, 80960VH 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 80960VH 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.

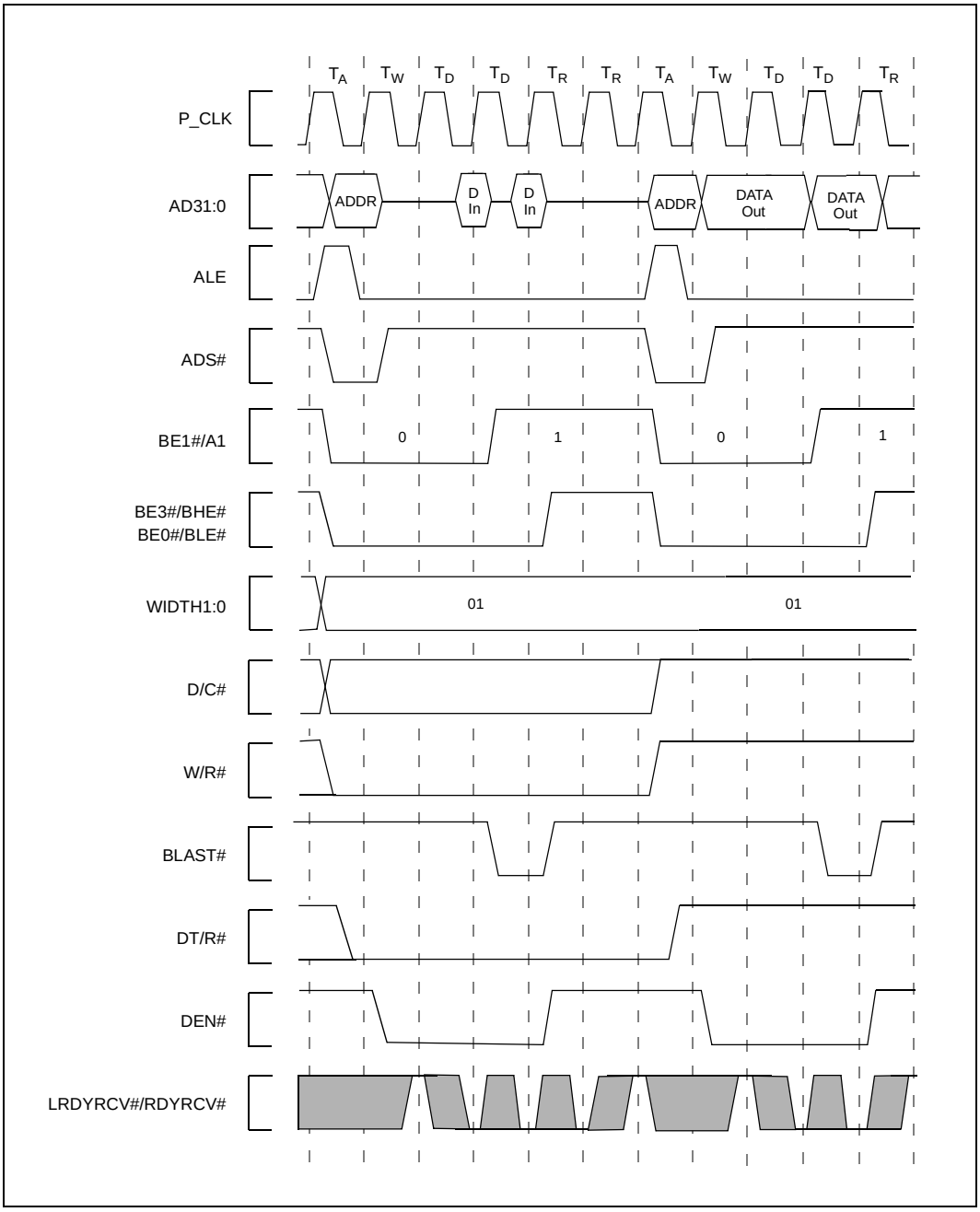
System ready logic is often described as normally-ready or normally-not-ready. Normally-ready logic asserts a microprocessor's input signal during all bus states, except when wait states are desired. Normally-not-ready logic deasserts a processor's input signal during all bus states, except when the processor is ready. The subtle nomenclature distinction is important for 80960VH 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 14-11 is a timing waveform of a read cycle followed by a write cycle, with an extra recovery state inserted into the read cycle.

Figure 14-11. Burst Read/Write Transactions with 1,0 Wait States - Extra T_R State on Read, 16-Bit Bus



14.4 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.

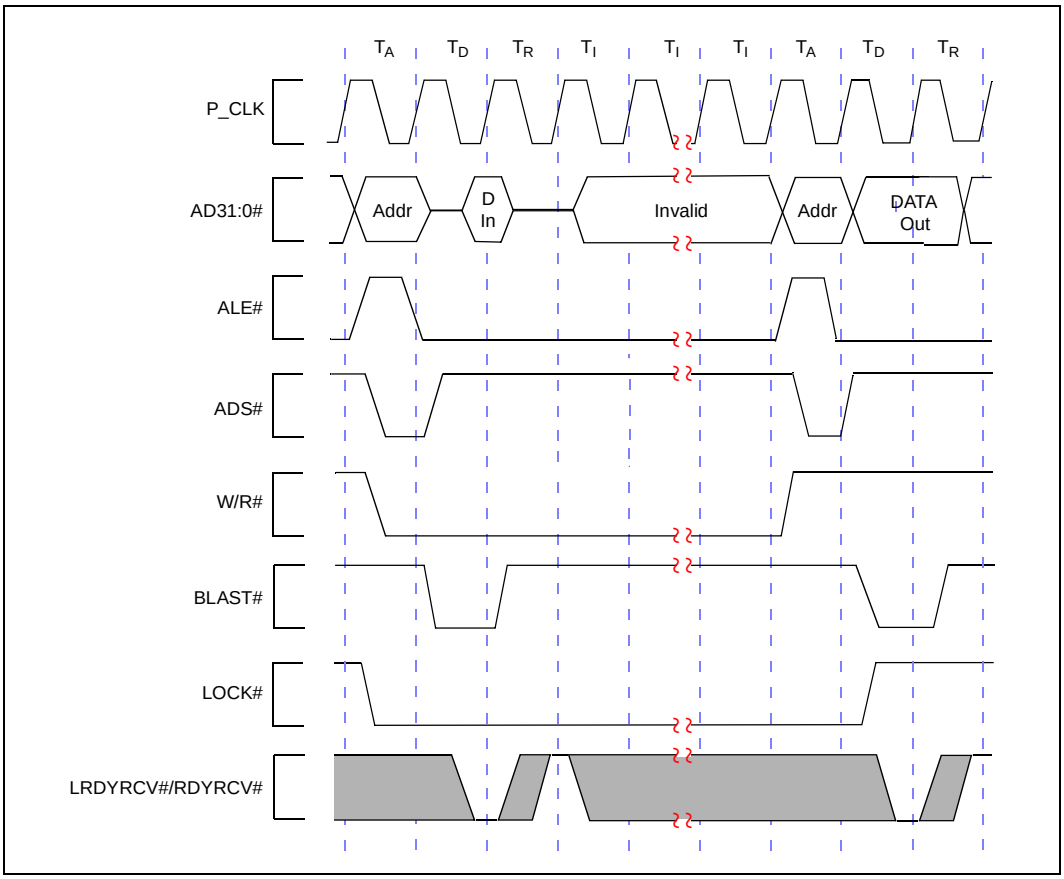
14.5 Atomic Bus Transactions

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.

For atomic bus accesses, the 80960VH 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 80960VH 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 80960VH does not acknowledge HOLD requests. This behavior is an enhancement over earlier i960 microprocessors. Figure 14-12 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 14-12. The LOCK# Signal



14.6 Bus Arbitration

The 80960VH 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 80960VH) 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 80960VH's internal local bus. Therefore, an external bus master cannot access any of the 80960VH's internal peripherals (for example, the Memory Controller, the i960 core, etc.).

Two processor signal signals comprise the bus arbitration signal group.

14.6.1 HOLD/HOLDA Protocol

In most cases, the 80960VH controls the bus; an I/O peripheral (for example, 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 80960VH 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 80960VH asserts HOLDA, it enters the Hold state (see Figure 14-2). 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.

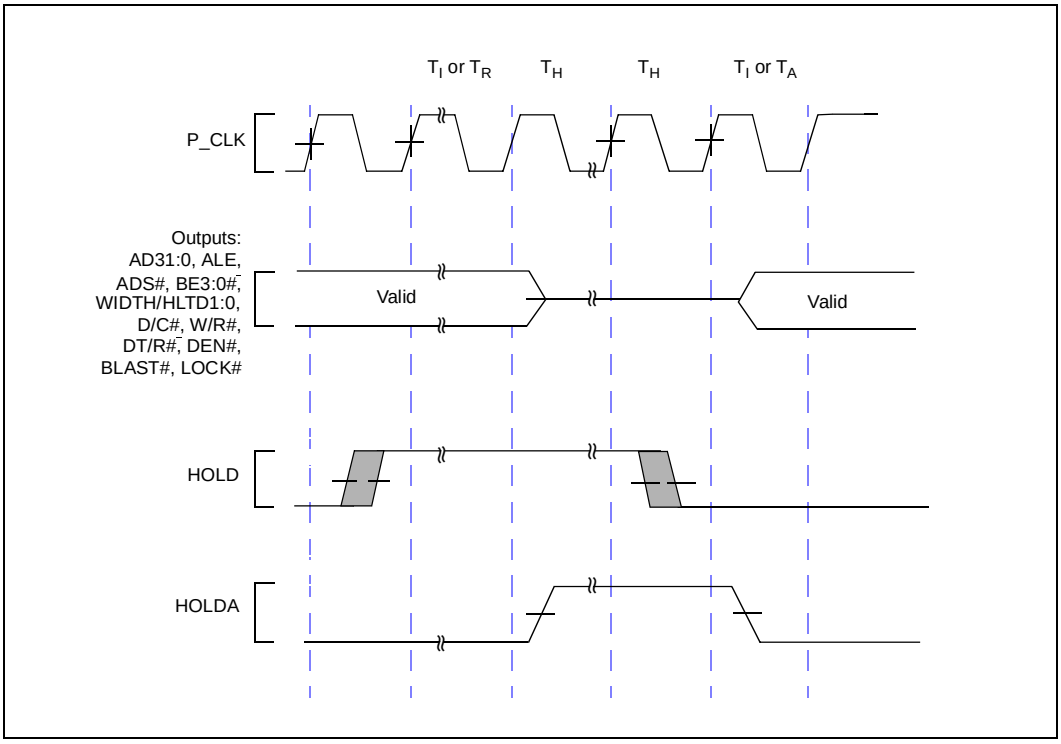
When the bus is in hold and the 80960VH needs to regain the bus to perform a transaction, the processor does not deassert HOLDA.

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 80960VH, 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 14-13 illustrates a HOLD/HOLDA arbitration sequence.

Note: External bus masters do not have access to the 80960VH internal, local bus. Therefore, an external bus master can not access any of the 80960VH internal peripherals (for example, memory controller, i960 core processor, and memory-mapped registers).

Figure 14-13. Arbitration Timing Diagram for a Bus Master

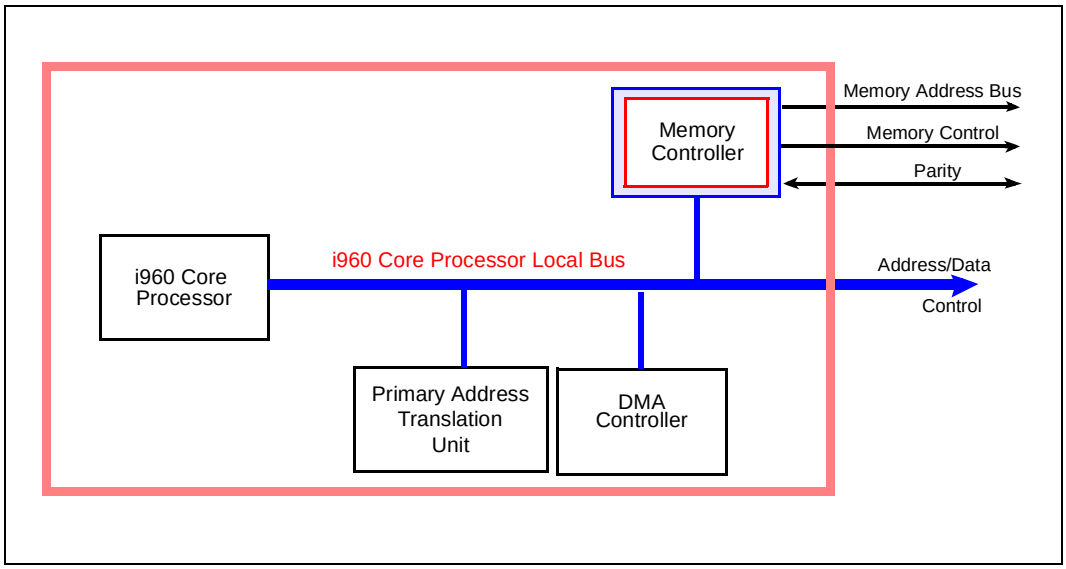


The 80960VH arbitration logic enables external bus masters to control 80960VH local bus. The Local Bus Arbitration Unit maintains the basic 80960VH protocol for the HOLD/HOLDA except that the 80960VH processor will not respond to the assertion of the HOLD signal (i.e., assert the HOLDA signal) during reset. This includes Processor Reset and Local Bus Reset.

Memory Controller 15

This chapter describes the i960® VH 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 15-1 provides an overview of the 80960VH's integrated memory controller.

Figure 15-1. i960® VH Processor Integrated Memory Controller



15.1 Supported Memory Types

The 80960VH 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

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.

15.2 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.

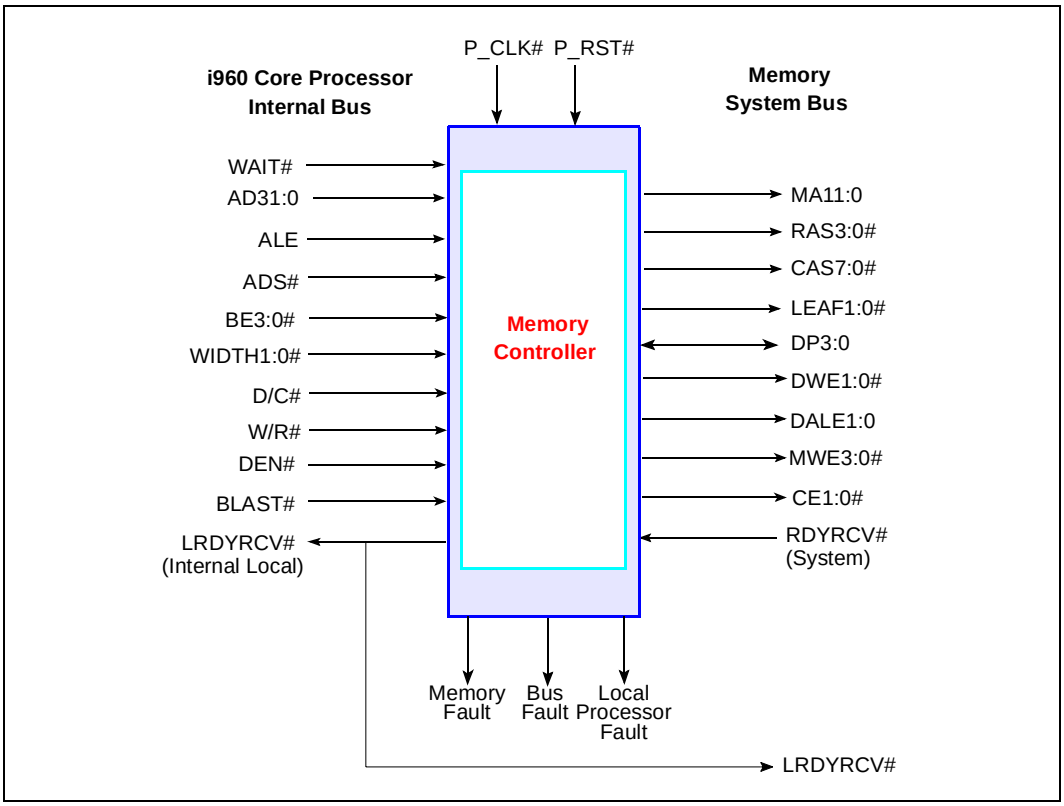
Configuration registers select characteristics associated with each type of memory used in a system. The memory controller configuration registers are located in the address range 0000 1500H to 0000 15FFH. The memory-mapped registers are summarized in Appendix C, "Memory-Mapped Registers". Once configured, 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 15-2 shows the interface signals. Refer to the *80960VH Microprocessor* for a complete description.

Figure 15-2. Memory Controller Signal Overview



15.3 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 15.5.3, on page 15-9. 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#.

15.4 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 15.10, “Overlapping Memory Regions” on page 15-39. The memory controller asserts the chip enable signals (CE1:0#) when the address on the 80960VH 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 15-1 summarizes the memory controller signals and the local bus signals used when connecting SRAM/ROM to the memory banks.

Table 15-1. 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
80960VH 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 15-3 shows an example of a 2 Mbyte, 32-bit ROM or SRAM system connected to memory bank 0.

Figure 15-3. Bank0 32-Bit ROM or SRAM System

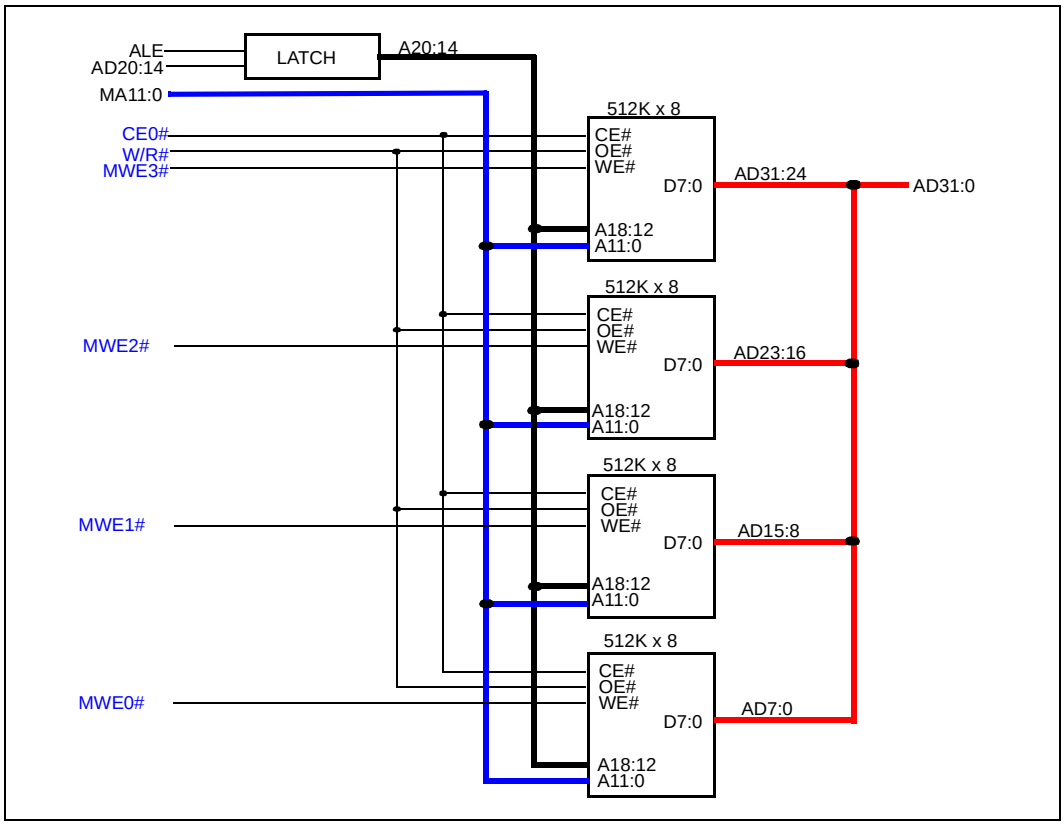
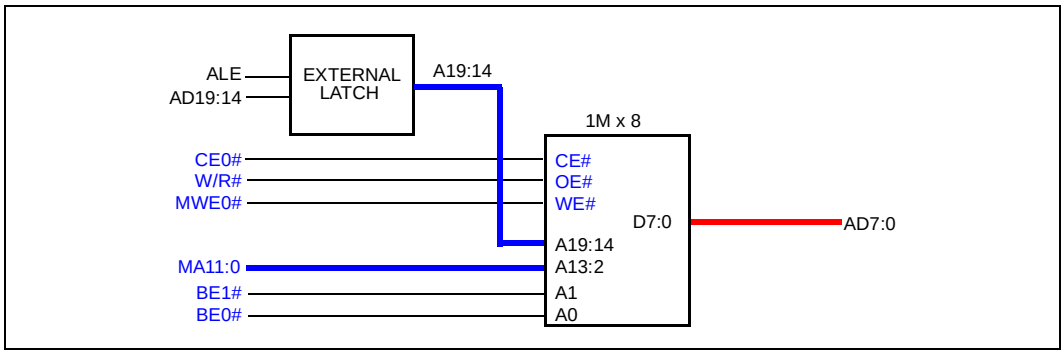


Figure 15-4 shows an example of an 1 Mbyte, 8-bit ROM or SRAM system connected to memory bank 0.

Figure 15-4. 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 15-4. 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 15-3. If either memory bank 0 or 1 is used for SRAM, then the memory write enable bit must be set to enable the assertion of the MWE3:0# signals for memory write transactions.

15.5 Memory Bank Programming Registers

Seven memory-mapped registers provide independent control of memory banks 0 and 1:

Table 15-2. Memory Bank Register Summary

Section	Register Name, Acronym	Page	Size (Bits)	Channel	80960 Local Bus Address	PCI Config Addr Offset
15.5.1	Memory Bank Control Register - MBCR	15-6	32		0000 1500H	NA
15.5.2	Memory Bank Base Address Registers - MBBAR0:1	15-8	32	0 1	0000 1504H 0000 1510H	NA
15.5.3.1	Memory Bank Read Wait State Registers - MBRWS0:1	15-10	32	0 1	0000 1508H 0000 1514H	NA
15.5.3.2	Memory Bank Write Wait State Registers - MBWWS0:1	15-11	32	0 1	0000 150CH 0000 1518H	NA

Refer to Appendix C, "Memory-Mapped Registers" for the memory-mapped registers address mappings.

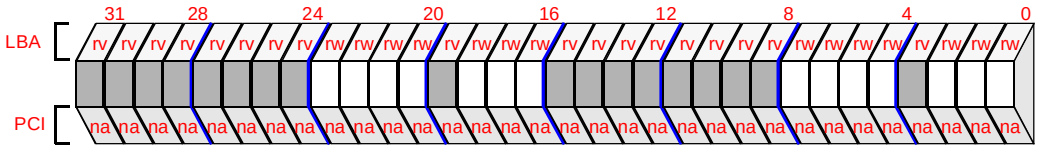
15.5.1 Memory Bank Control Register - MBCR

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 15-3 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 15.5.2, “Memory Bank Base Address Registers - MBBAR0:1” on page 15-8 for additional information.

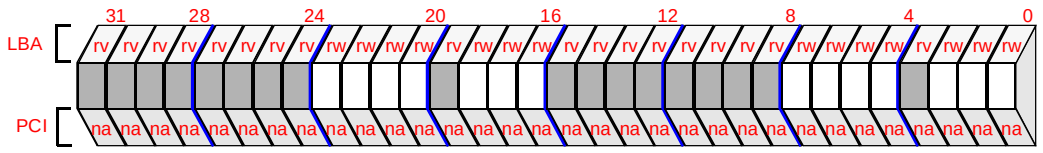
Table 15-3. Memory Bank Control Register – MBCR (Sheet 1 of 2)



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. 000064 Kbytes 0001128 Kbytes 0010256 Kbytes 0011512 Kbytes 01001 Mbyte 01012 Mbytes 01104 Mbyte 01118 Mbytes 1xxx16 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 P_CLK 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.
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

Table 15-3. Memory Bank Control Register – MBCR (Sheet 2 of 2)



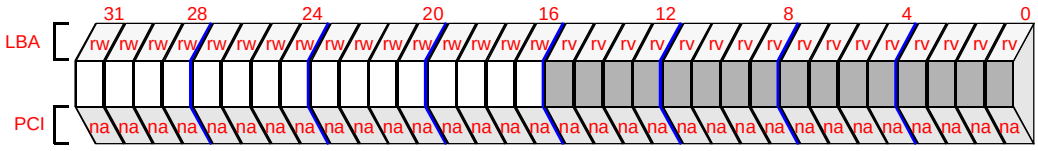
LBA: 1500H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
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 P_CLK 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.

15.5.2 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 15-4.

Table 15-4. Memory Bank Base Address Registers – MBBAR0:1



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.

15.5.3 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 15-2 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 P_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:

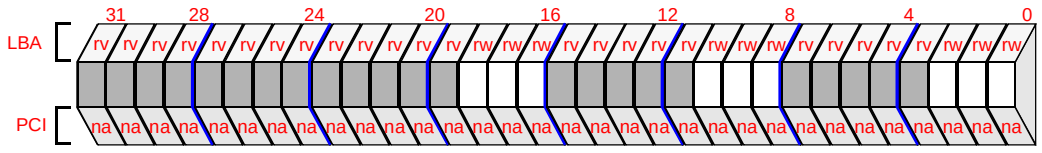
- Address-to-Data wait states
- Data-to-Data wait states
- Data-to-Address wait states (i.e., turnaround cycles)

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 15-5.

15.5.3.1 Memory Bank Read Wait State Registers - MBRWS0:1

The Memory Bank Read Wait State Register (MBRWS) describes the wait states during Read cycles.

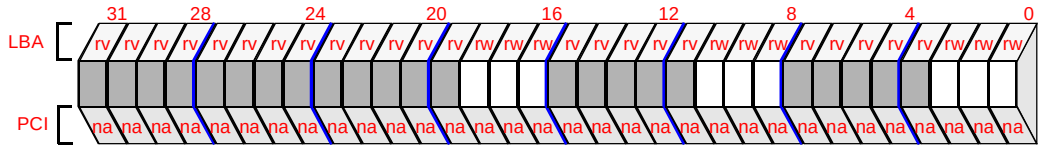
Table 15-5. Memory Bank Read Wait States Register – MBRWS0:1 (Sheet 1 of 2)



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

Table 15-5. Memory Bank Read Wait States Register – MBRWS0:1 (Sheet 2 of 2)



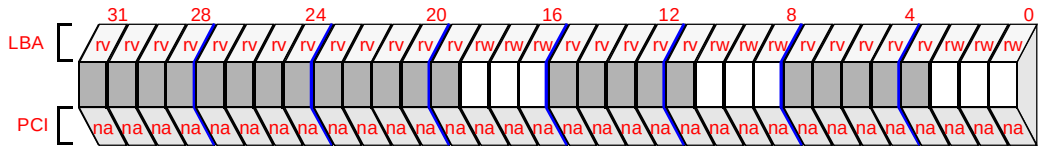
LBA: Bank 0 = 1508H Bank 1 = 1514H **PCI:** N/A	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
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

15.5.3.2 Memory Bank Write Wait State Registers - MBWWS0:1

The Memory Bank Write Wait State Register (MBWWS) describes wait states during write cycles.

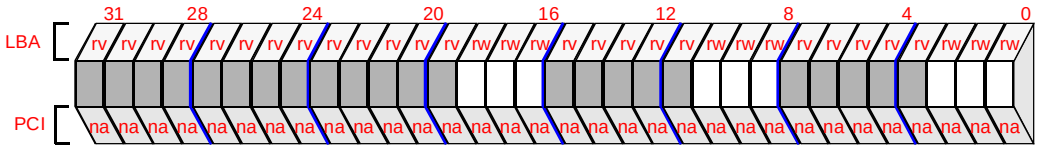
Table 15-6. Memory Bank Write Wait States Register – MBWWS0:1 (Sheet 1 of 2)



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

Table 15-6. Memory Bank Write Wait States Register – MBWWS0:1 (Sheet 2 of 2)



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
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

15.5.4 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 15-5 shows a burst read transaction with a wait state profile of 2,1,1,1.

Table 15-7. 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 15-5. 32-Bit Bus, Burst Flash Memory, Read Access with 2,1,1,1 Wait States

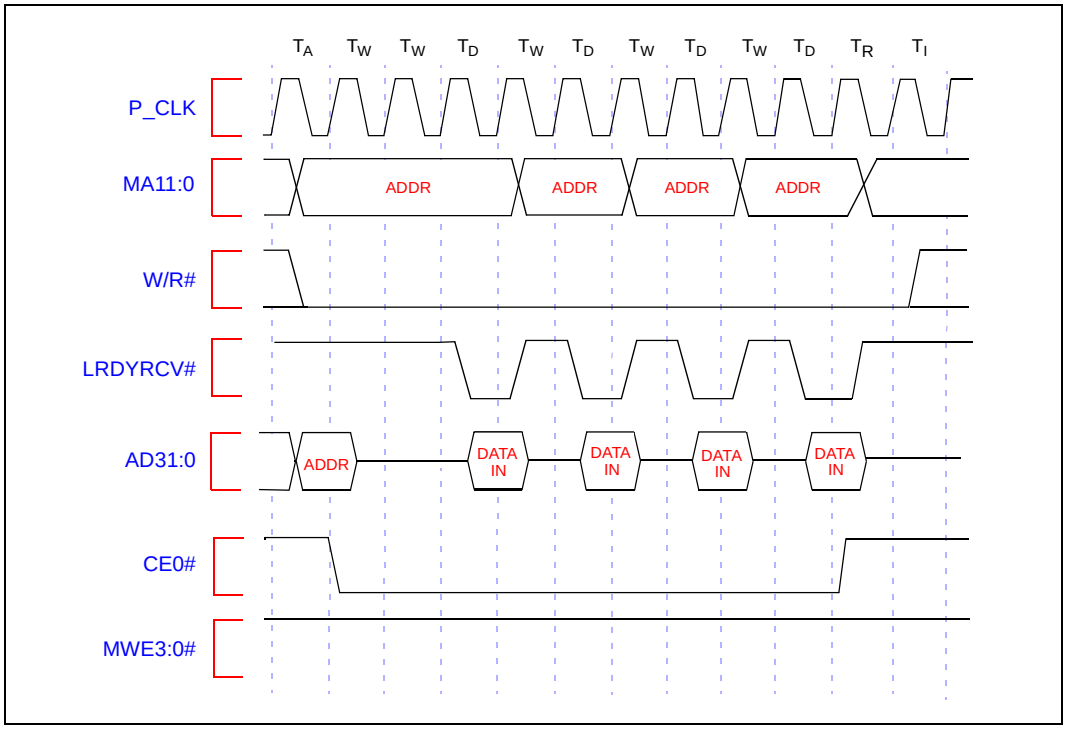
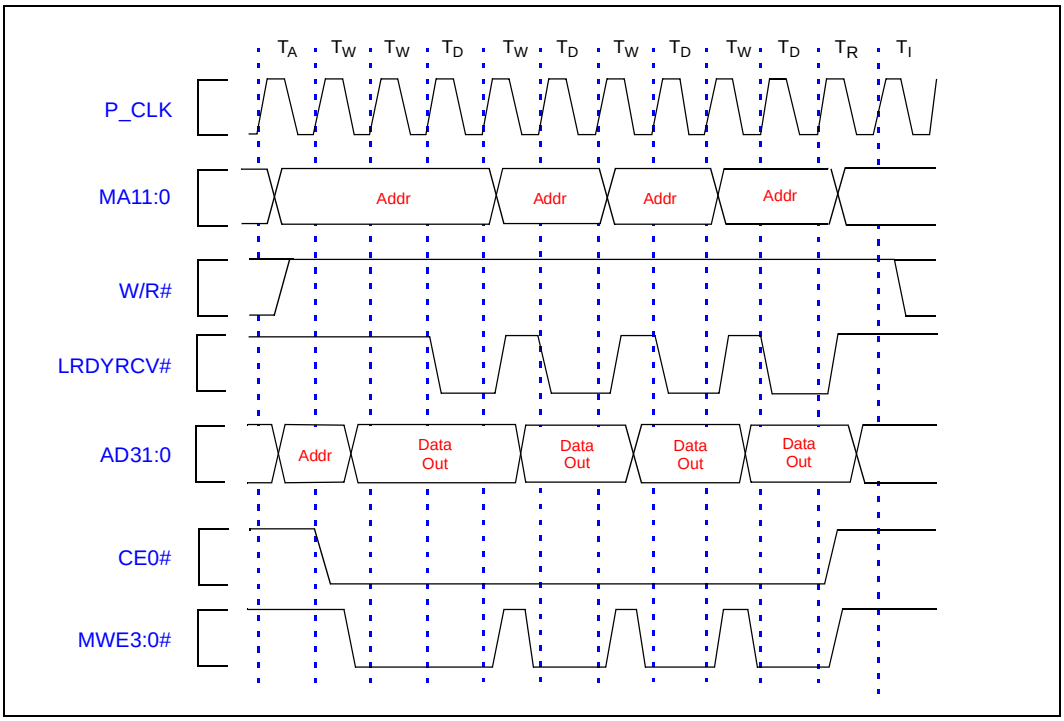


Figure 15-6 represents a burst write transaction to SRAM with a wait state profile of 2,1,1,1. The Extended MWE3:0# control bit in the MBCR is cleared in this example.

Table 15-8. 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 15-6. 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 15-7 shows a read transaction with 0 wait state SRAM.

Table 15-9. 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 15-7. 32-Bit Bus, SRAM Read Accesses with 0 Wait States

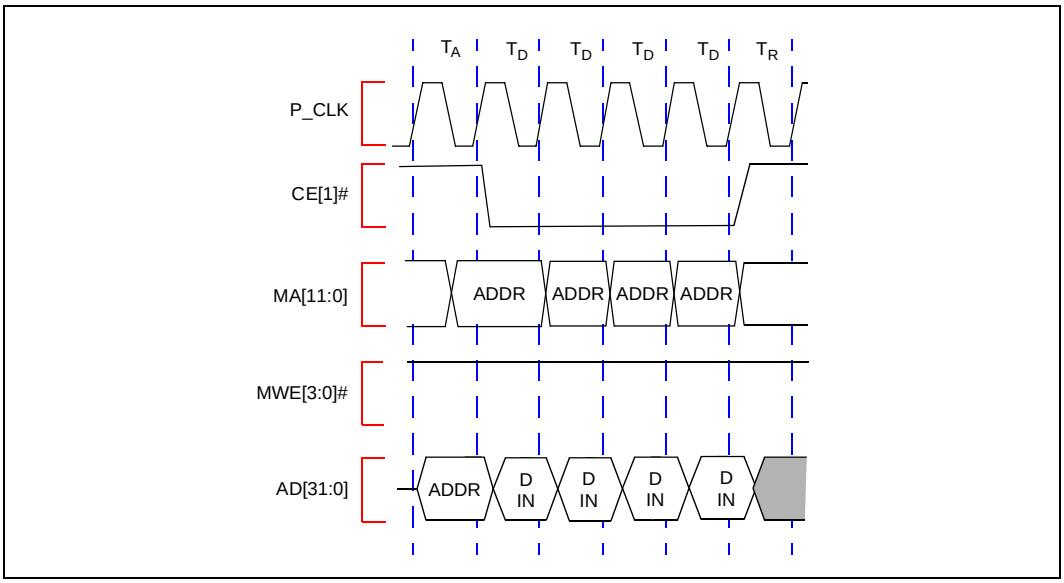
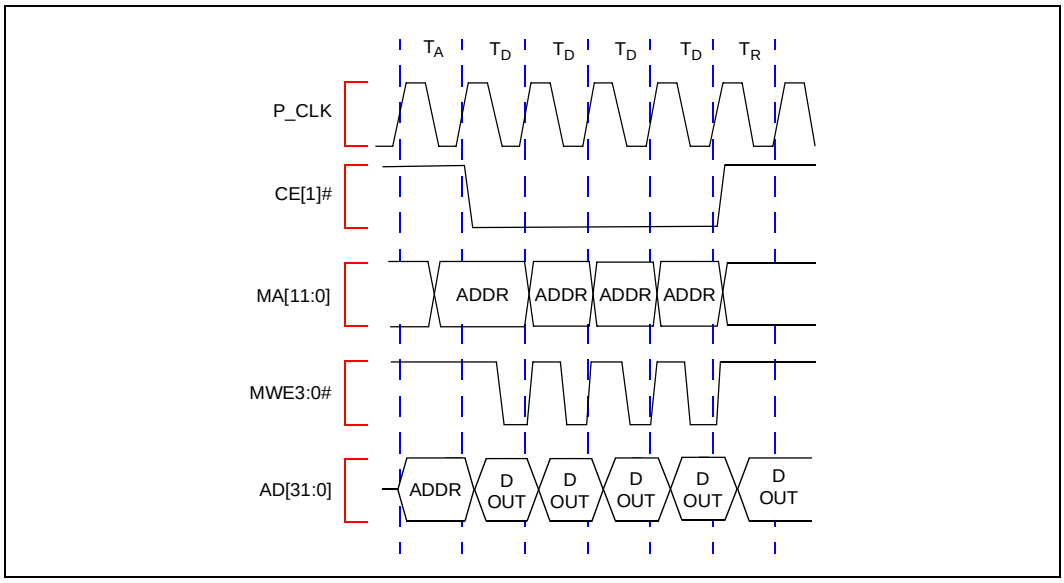


Figure 15-8 represents a 0 wait state write transaction to SRAM.

Table 15-10. 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 15-8. 32-Bit Bus, SRAM Write Access With 0 Wait States



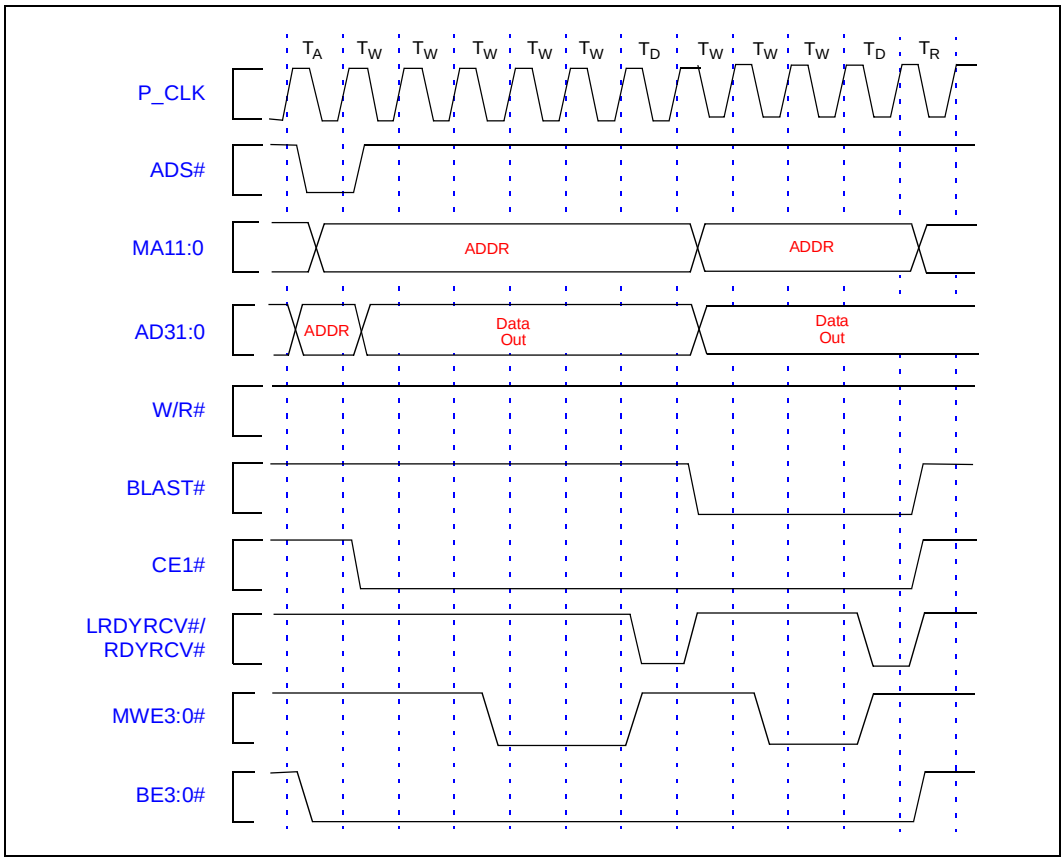
15.5.5 Extending Memory Write Enable Signals

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 15-9 shows a 2-word burst of an extended MWE3:0# write cycle.

Table 15-11. 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 15-9. 32-Bit Bus, Write Access with Extended MWE3:0#



15.6 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 15-12.

Table 15-12. 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 two different types of DRAM: Fast Page-Mode (FPM) and Extended Data Out (EDO). Interleaved Fast Page-Mode DRAM is also supported. DRAM refresh is supported through the programmable DRAM refresh counter.

15.6.1 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 80960VH 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 15-13 summarizes the supported DRAM organization and type.

Table 15-13. Supported DRAM Configurations

Interleaved DRAM (Fast Page-mode DRAM Only)	Non-Interleaved DRAM (FPM, EDO)
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 15-10. As shown, the 80960VH is a direct connect to the non-interleaved memory subsystem (no additional logic is required).

Figure 15-10. Non-Interleaved, 32-Bit, Single Bank, DRAM System

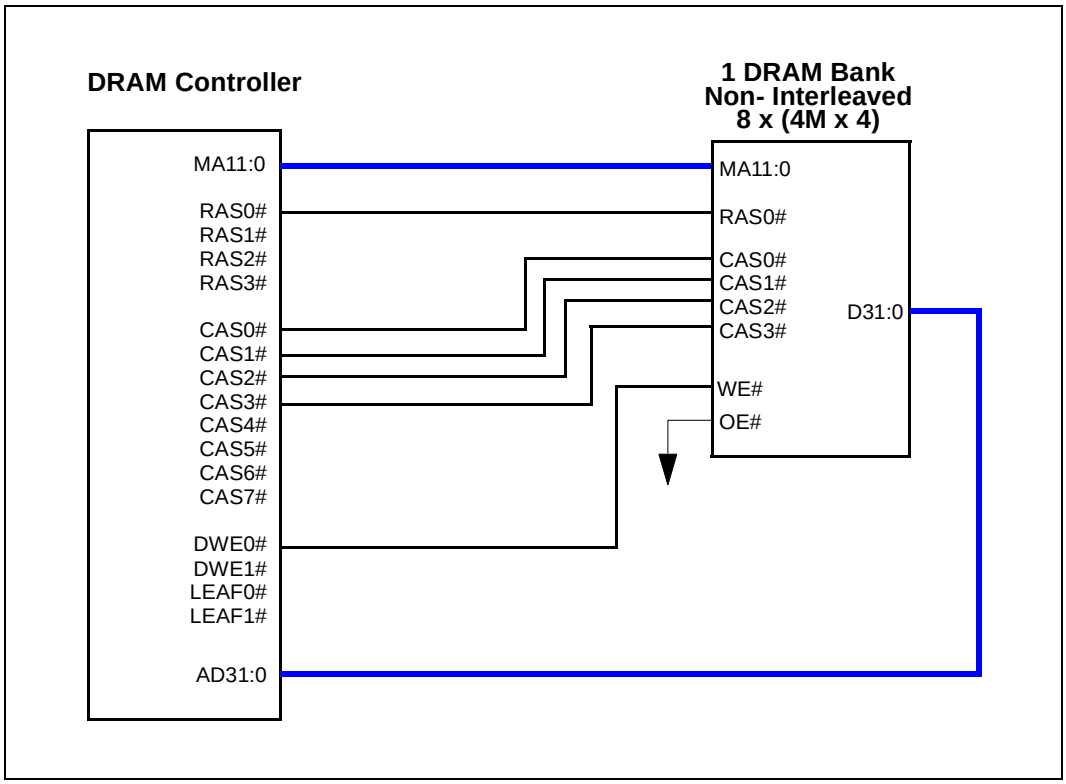
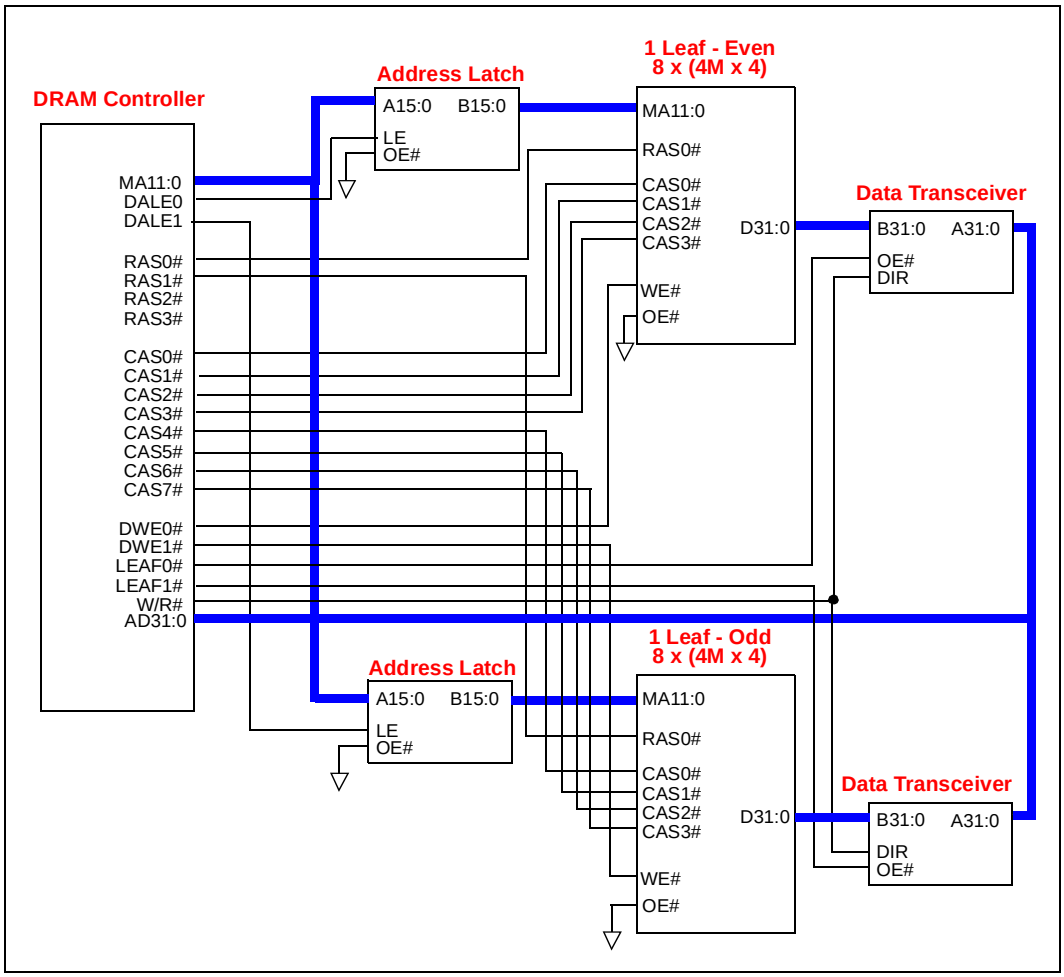


Figure 15-11 shows a sample memory system using 32-bit interleaved DRAM. The memory controller provides eight CAS# signals for the support of interleaved memory. The CAS3:0# signals provide the byte selection for one leaf, while CAS7:4# provide for the second leaf. It is necessary to control external 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 15-11. 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 15-14.

Table 15-14. Supported DRAM Configurations (Symmetric Addressing Only) (Sheet 1 of 2)

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

NOTE:
1. Every bank (or leaf) must use the same memory type. Mixed combinations of FPM or EDO are not permitted. The DRAM bank size must also remain the same among banks (or leaves).

Table 15-14. Supported DRAM Configurations (Symmetric Addressing Only) (Sheet 2 of 2)

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.
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

NOTE:
1. Every bank (or leaf) must use the same memory type. Mixed combinations of FPM or EDO are not permitted. The DRAM bank size must also remain the same among banks (or leaves).

15.6.2 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 15-15 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 15-15. 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

15.6.3 DRAM Registers

The DRAM controller provides registers for configuring and controlling DRAM. Six memory-mapped registers control the memory controller for independent operation:

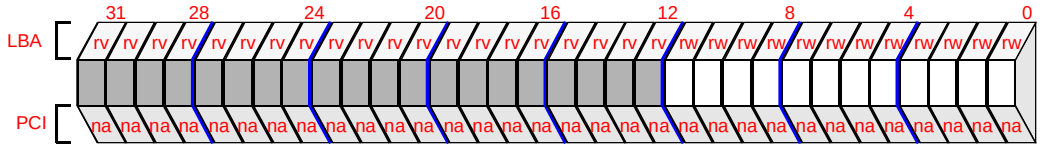
Table 15-16. DRAM Register Summary

Section	Section, Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
15.6.4	DRAM Bank Control Register — DBCR	15-22	32	0000 151CH	N/A
15.6.5	DRAM Base Address Register — DBAR	15-23	32	0000 1520H	N/A
15.6.6	DRAM Read Wait State Register — DRWS	15-24	32	0000 1524H	N/A
15.6.7	DRAM Write Wait State Register — DWWS	15-25	32	0000 1528H	N/A
15.6.8	DRAM Refresh Interval Register — DRIR	15-27	32	0000 152CH	N/A
15.7.1	DRAM Parity Enable Register — DPER	15-29	32	0000 1530H	N/A

15.6.4 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 15-17 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.

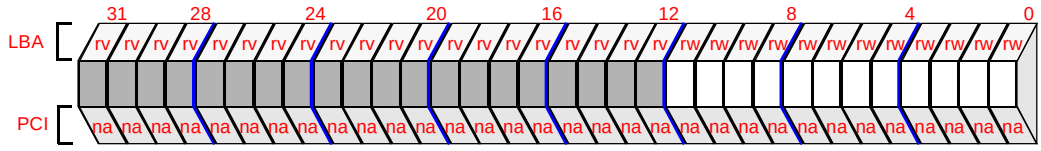
Table 15-17. DRAM Bank Control Register — DBCR (Sheet 1 of 2)



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.
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 or EDO 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 10 Reserved 010 11 Extended Data Out (EDO) DRAM, 4 Banks 1xx xx Reserved

Table 15-17. DRAM Bank Control Register — DBCR (Sheet 2 of 2)



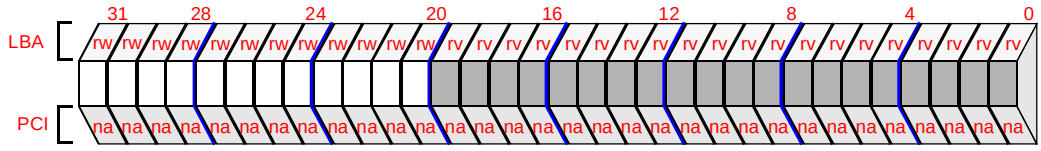
LBA: 151CH **PCI:** N/A	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
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.

15.6.5 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 15-18.

Table 15-18. DRAM Base Address Register — DBAR



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.

15.6.6 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 P_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 15-12. The register definitions for the DRAM Bank Read Wait States Register are shown in Table 15-19. The number of t_{RRC}, t_{RCP} and t_{RRCV} wait states is encoded in two-bit fields, which are also shown in Table 15-19.

Figure 15-12. DRAM Read Cycle Programmable Parameter Example

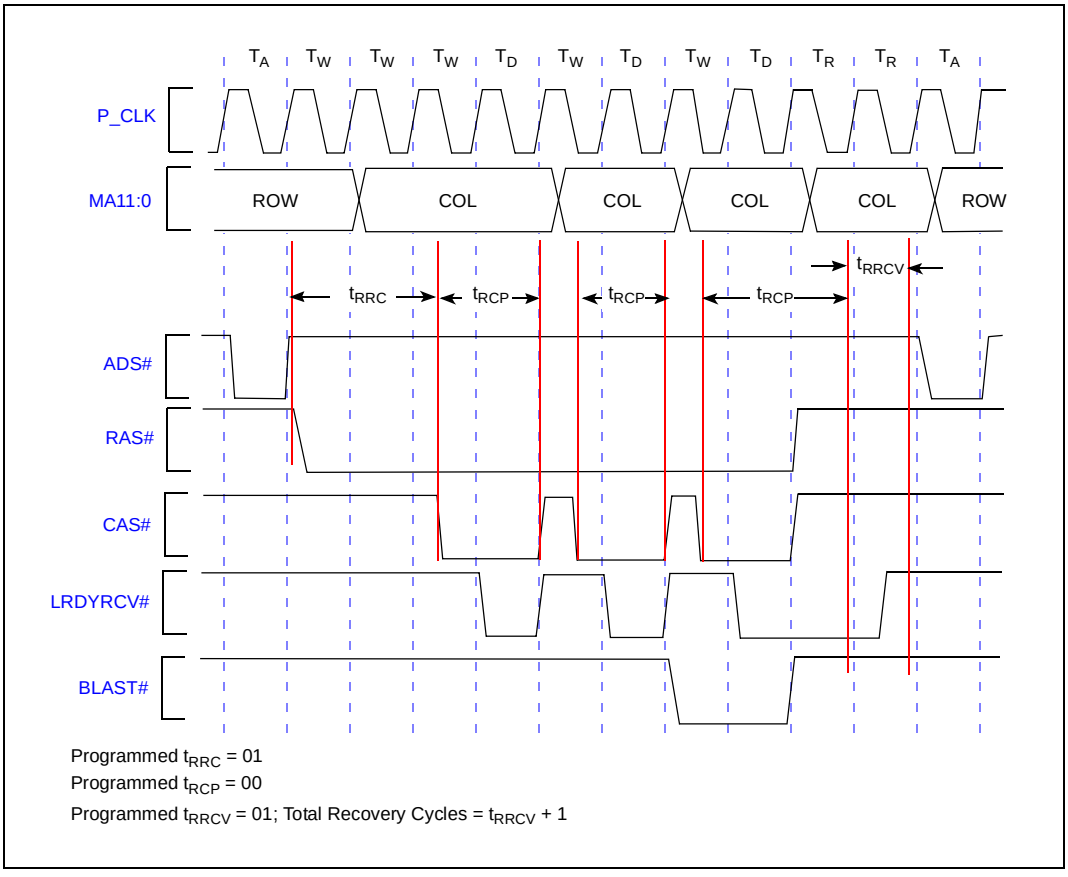
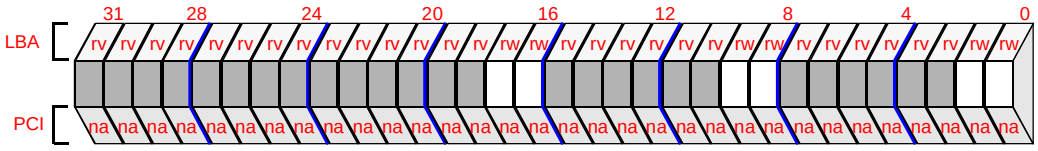


Table 15-19. DRAM Bank Read Wait State Register — DRWS



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 00 1.5 cycles / 1 cycles 01 2.5 cycles / 2 cycles 10 3.5 cycles / 3 cycles 11 4.5 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 (this parameter is fixed for EDO DRAM type): 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

15.6.7 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 P_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 15-13. The number of t_{WRC}, t_{WCP} and t_{WRCV} wait states is encoded in two-bit fields as shown in Table 15-20.

Figure 15-13. DRAM Write Cycle Programmable Parameter Example

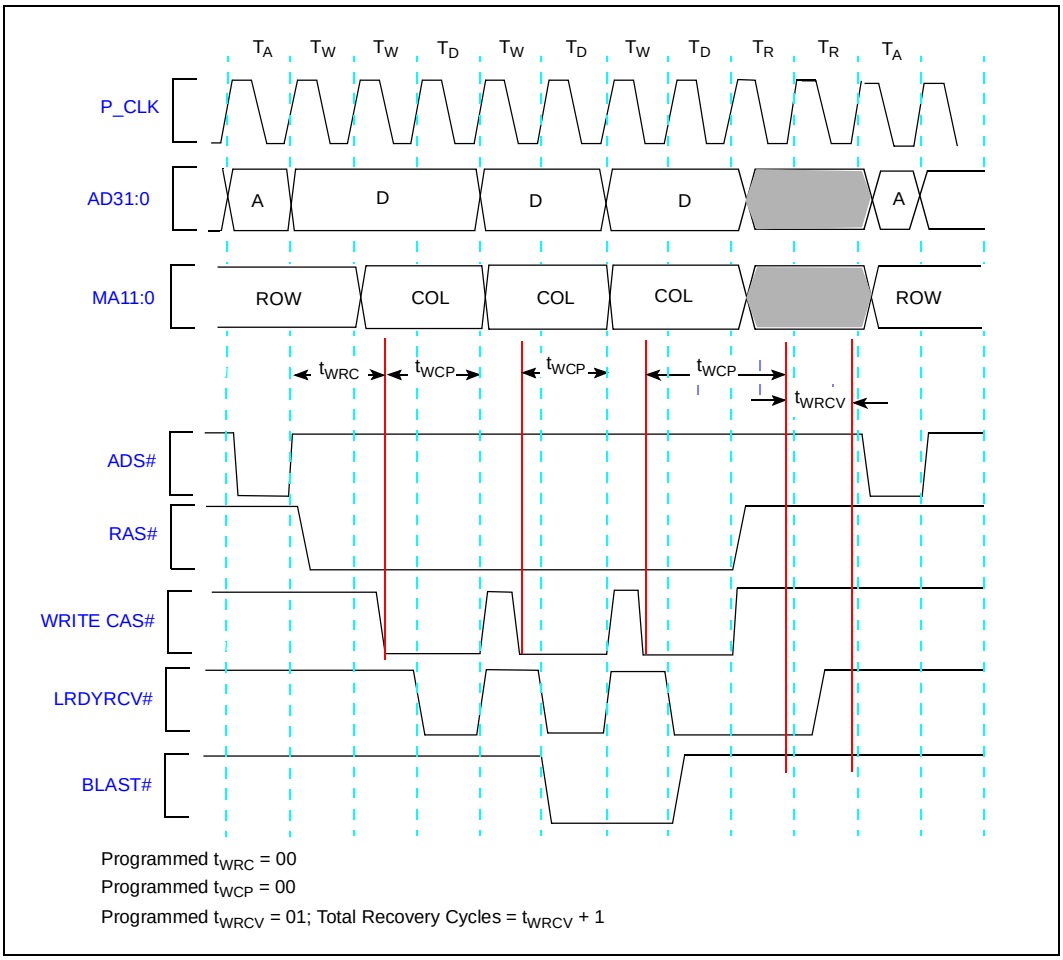
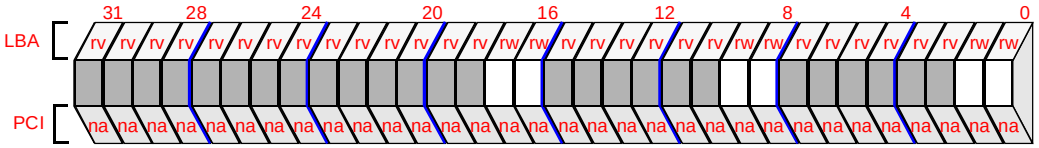


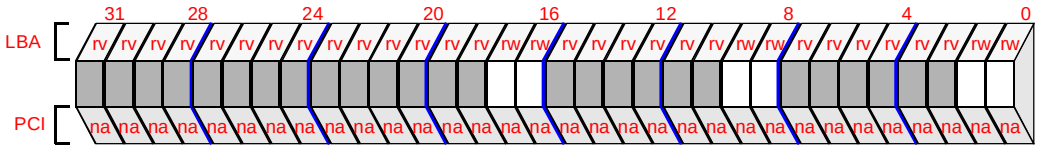
Table 15-20. DRAM Bank Write Wait State Register — DWWS (Sheet 1 of 2)



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

Table 15-20. DRAM Bank Write Wait State Register — DWWS (Sheet 2 of 2)

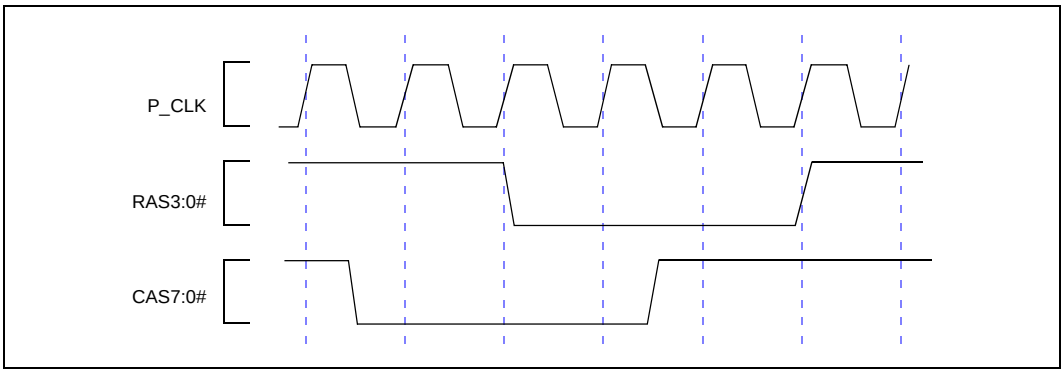


LBA: 1528H **PCI:** N/A		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
17:16	00_2	**DRAM Write cycle RAS-to-CAS delay (t_{WRC})** - This field affects the number of cycles between the assertion of RAS3:0# and the assertion of CAS7:0#. FPM EDO 00 1.5 cycles 1.5 cycles 01 2.5 cycles 2.5 cycles 10 3.5 cycles 3.5 cycles 11 4.5 cycles 4.5 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 (this parameter is fixed for EDO DRAM type): 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

15.6.8 DRAM Refresh Interval Register – DRIR

The memory controller supports CAS# Before RAS# (CBR) refresh cycles for DRAM devices. Figure 15-14 shows an example of a typical CBR refresh cycle.

Figure 15-14. CAS#-Before-RAS# DRAM Refresh



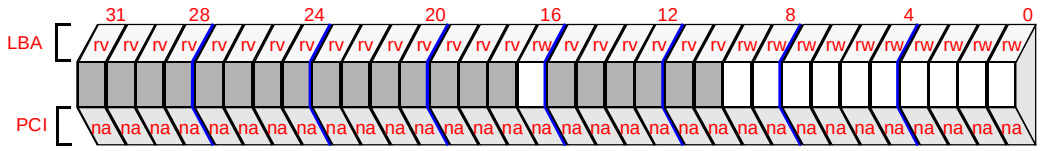
The internal DRAM Refresh Interval Register (DRIR) (Table 15-21) provides the time delay between DRAM refresh cycles and is programmed in increments of P_CLK. The value programmed is determined as follows:

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 15-21. DRAM Refresh Interval Register — DRIR



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 P_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 80960VH.

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.

15.7 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.

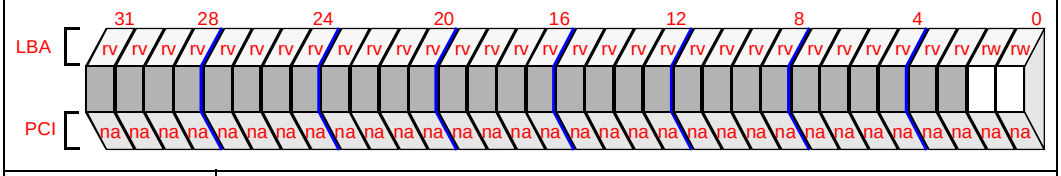
Table 15-22. Error Checking and Reporting Register Summary

Section	Section, Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
15.7.1	DRAM Parity Enable Register — DPER	15-29	32	0000 1530H	N/A
15.7.2	Bus Monitor Enable Register — BMER	15-30	32	0000 1534CH	N/A
15.7.3	Memory Error Address Register — MEAR	15-31	32	0000 1538H	N/A
15.7.4	Local Processor Interrupt Status Register — LPISR	15-32	32	0000 153CH	N/A

15.7.1 DRAM Parity Enable Register – DPER

The use of parity is programmable through the DRAM Parity Enable Register (DPER), shown in Figure 15-23. 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 15-23. DRAM Parity Enable Register — DPER



LBA: 1530H **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
31:2	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.

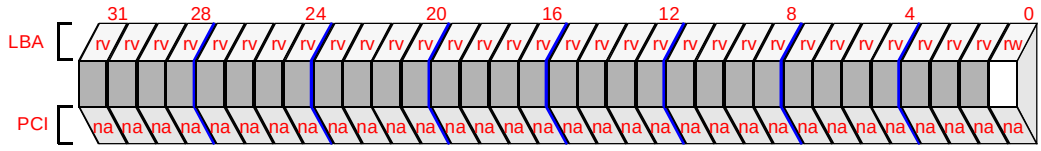
Upon detection of a parity error, the 30-bit address of the faulty memory location is latched and stored in the Memory Error Address Register (MEAR), Table 15-25. The memory controller detects parity errors for any on-chip bus master. These include the primary ATU, DMA channel 0, DMA channel 1 or the 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.

15.7.2 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 P_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 15-24. On memory controller reset, the bus monitor interrupt is disabled.

Table 15-24. Bus Monitor Enable Register — BMER



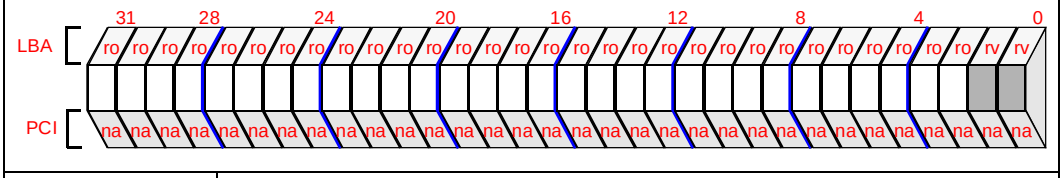
LBA: 1534H **PCI:** N/A	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:1	0000 0000 H	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.

15.7.3 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 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 15-25. Memory Error Address Register — MEAR



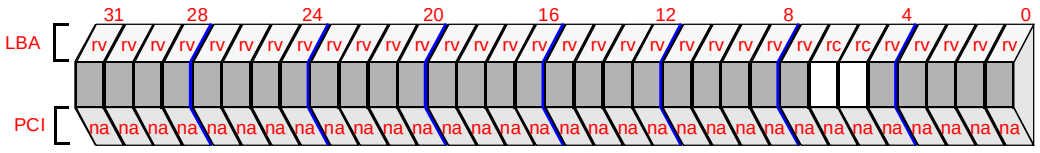
LBA: 1538H **PCI:** N/A	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000 H	**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 ATU 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

15.7.4 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.

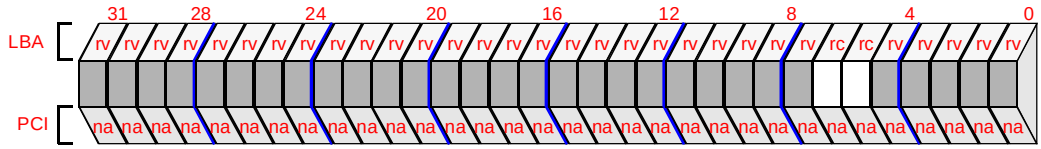
Table 15-26. Local Processor Interrupt Status Register — LPISR (Sheet 1 of 2)



LBA: 153CH **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
31:7	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.

Table 15-26. Local Processor Interrupt Status Register — LPISR (Sheet 2 of 2)



LBA: 153CH **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
4:0	00H	Reserved

15.8 DRAM Waveforms

Waveforms showing FPM and EDO DRAM read and write cycles are contained in the following sections. Also included are interleaved and non-interleaved FPM examples.

15.8.1 Non-Interleaved Fast Page-Mode DRAM Waveform

Figure 15-15 and Figure 15-16 represent non-interleaved FPM DRAM system read and write cycle waveforms. The programmed timings used in these two examples are shown in Table 15-27.

Table 15-27. 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 15-15. FPM DRAM System Read Access, Non-Interleaved, 3,1,1,1, Wait States

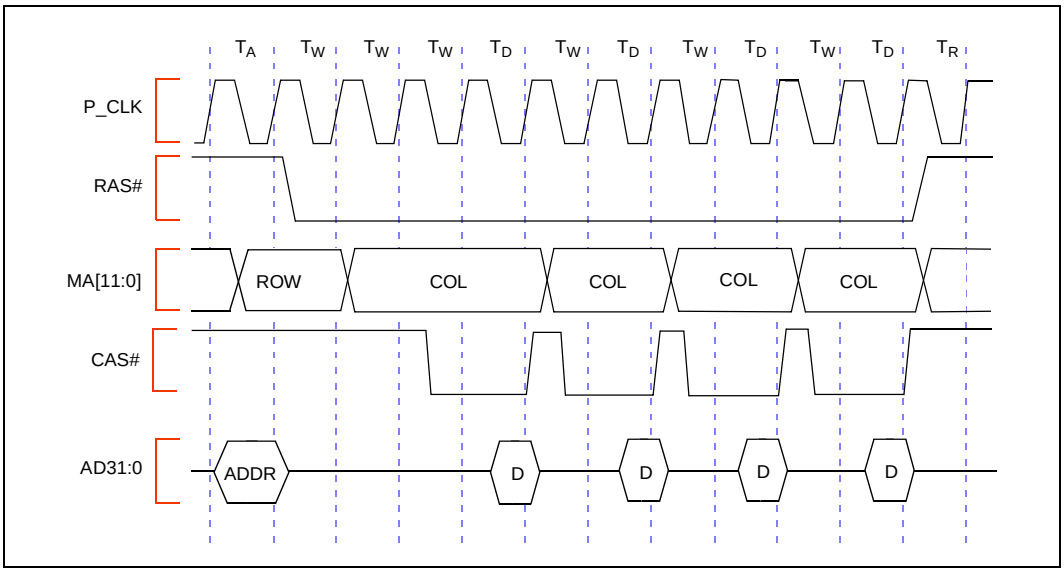
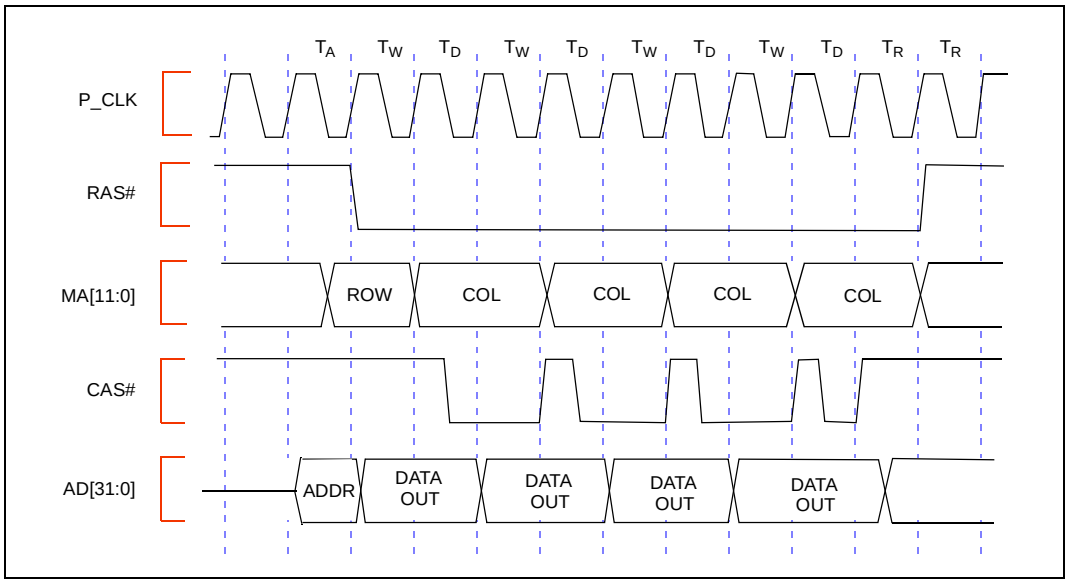


Figure 15-16. FPM DRAM System Write Cycle



15.8.2 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 15.6.4, "DRAM Bank Control Register — DBCR" on page 15-22.

The QA31:0 and QB31:0 signals refer to the even and odd leaf (respectively) data transceiver outputs between the DRAM and the 80960VH. For an interleaved DRAM system, Figure 15-17 and Figure 15-18 represent typical read and write transactions. The programmed timings used in the examples are shown in Table 15-28.

Table 15-28. 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 15-17. FPM DRAM System Read Access, Interleaved, 2,0,0,0 Wait States

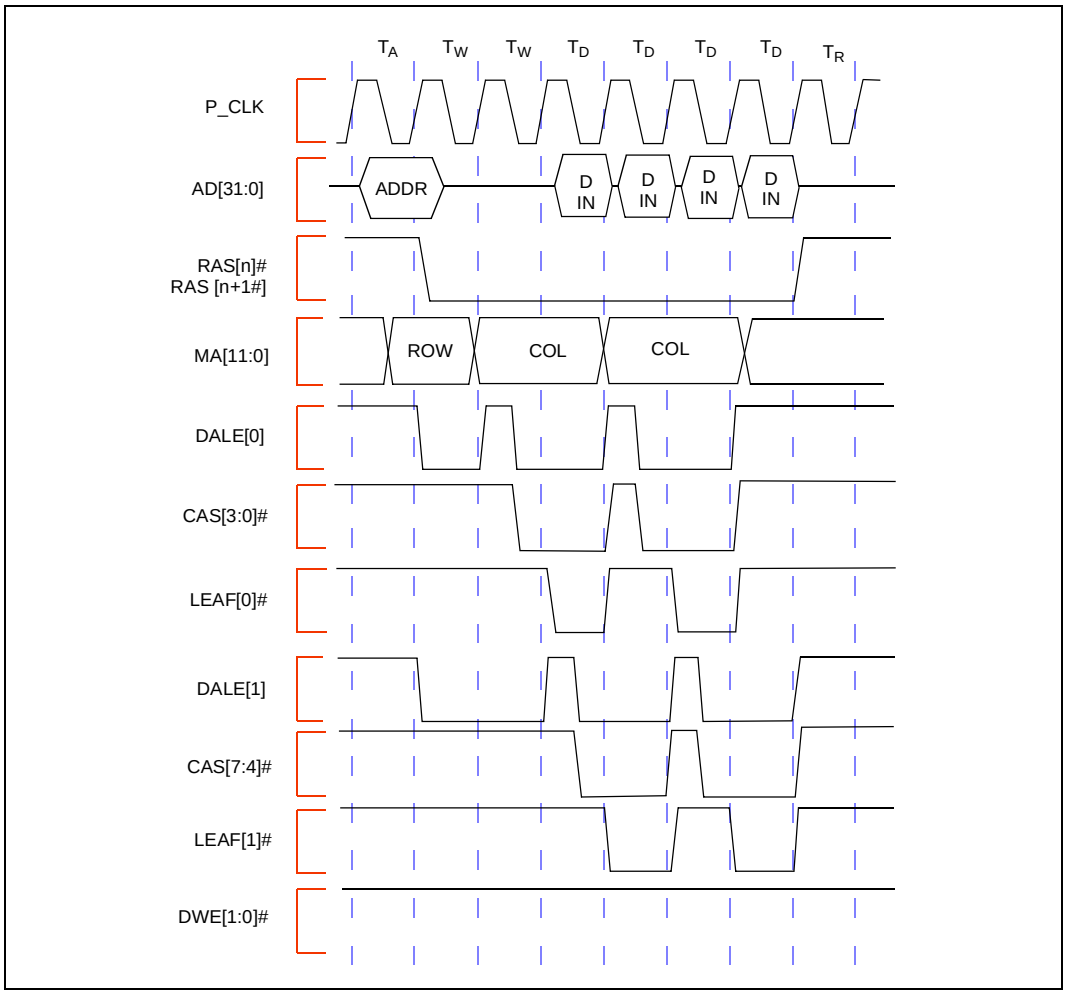
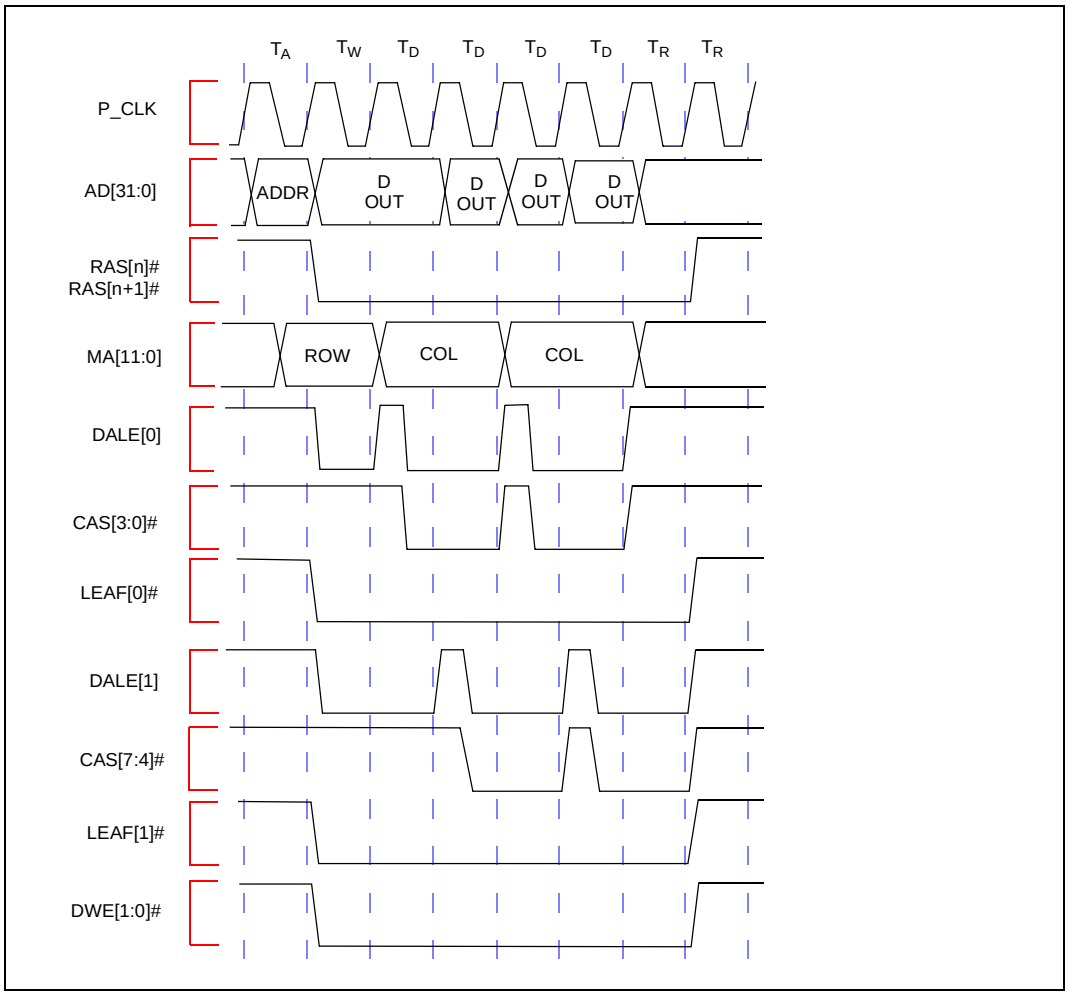


Figure 15-18. FPM DRAM System Write Access, Interleaved, 1,0,0,0 Wait States



15.8.3 EDO DRAM Waveform

Figure 15-19 and Figure 15-20 represent EDO DRAM system read and write cycle waveforms. The programmed timings are shown in Table 15-29.

Table 15-29. 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 15-19. EDO DRAM System Read Access, 2,0,0,0, Wait States

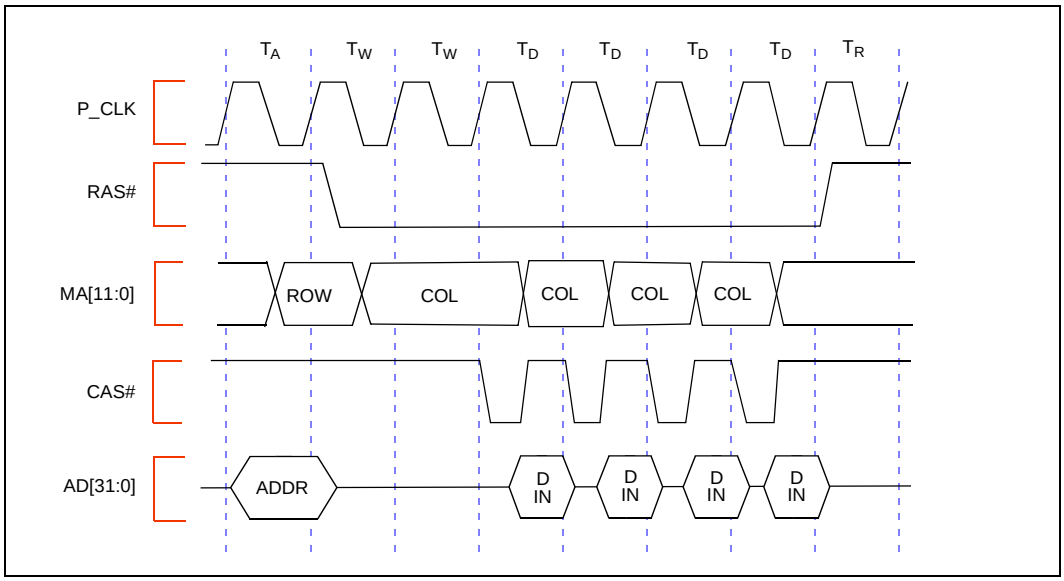
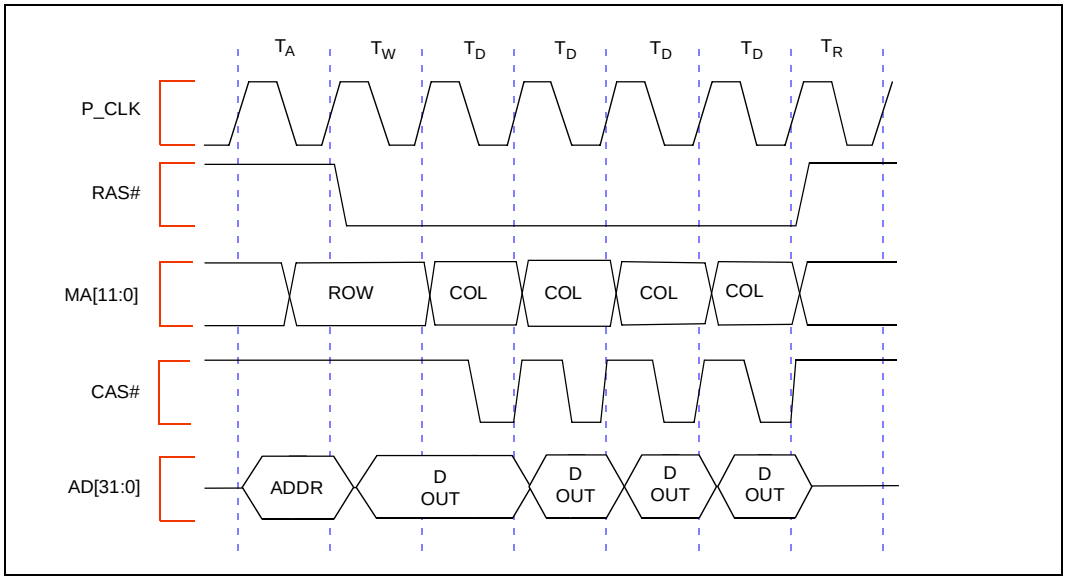


Figure 15-20. EDO DRAM System Write Access, 1,0,0,0 Wait States



15.9 Initializing Dram Devices

Both DRAM types, FPM and EDO, require a minimum of eight CAS# Before RAS# cycles prior to the first memory access.

To satisfy the initialization cycles required by all 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.

15.10 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 so that they overlap the reserved address space and other memory controller windows. Table 15-30 summarizes memory precedence used when this overlapping occurs.

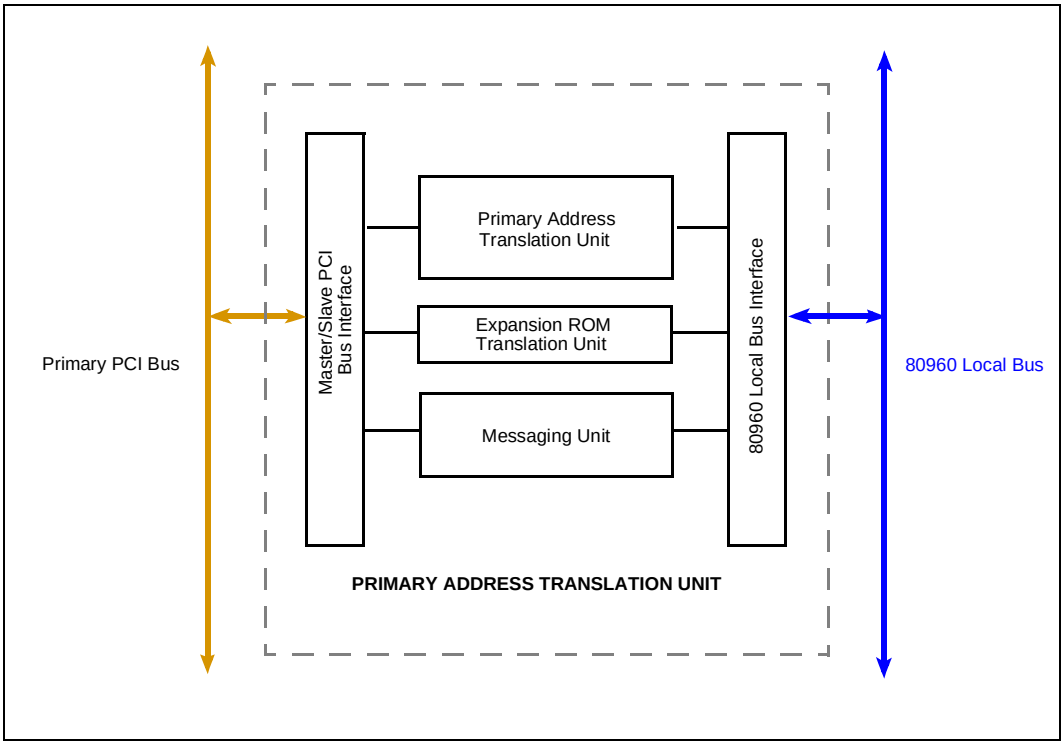
Table 15-30. Memory Precedence

Priority	Address Region
Highest	Memory-Mapped Register Address Space
	Primary Outbound Address Translation Unit Address Space
	DRAM Address Space
	Memory Bank 0 Address Space
Lowest	Memory Bank 1 Address Space

Address Translation Unit 16

The Address Translation Unit (ATU) is the interface between the PCI bus and the 80960 local bus. This chapter describes the ATU operation modes, setup, and interface.

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 the Address Translation Unit and the Messaging Unit (MU); described in Chapter 17, “Messaging Unit”. The MU allows the system processor and the 80960VH to transfer control information.

The ATU supports both inbound and outbound address translation. The ATU provides direct access between the primary PCI bus and the 80960 local bus. The primary ATU and MU share PCI address space.

The ATU and the MU appear as a single PCI device on the primary PCI bus. Collectively, these units are PCI function 0 of the 80960VH 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 the PCI bus are referred to as *outbound transactions* (80960 local bus to PCI).

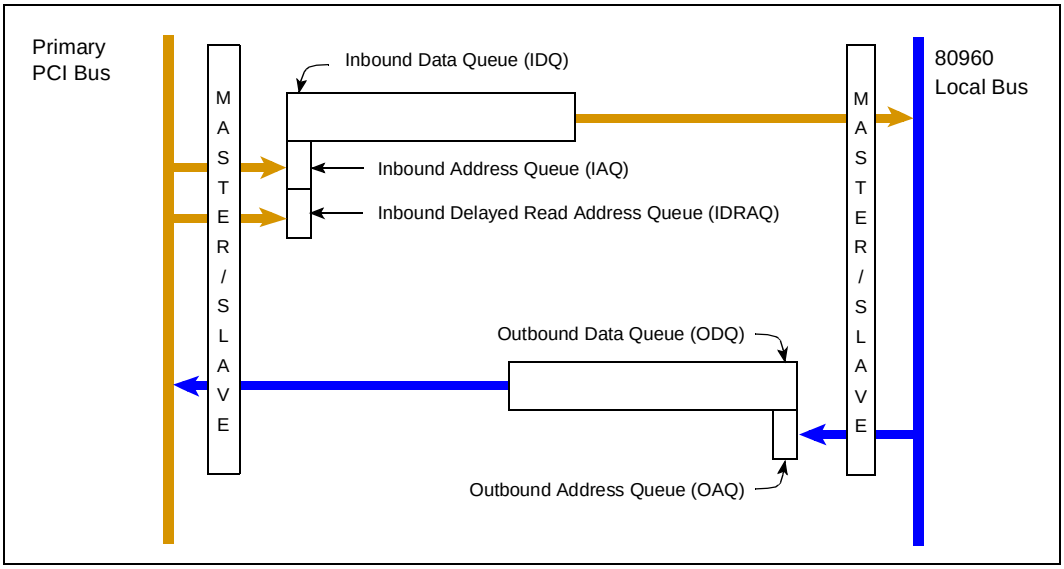
During inbound transactions, the ATU converts PCI addresses (initiated by a PCI bus master) to 80960 local bus addresses and initiates the data transfer on the 80960 local bus. During outbound transactions, the ATU converts 80960 local bus addresses to PCI addresses and initiates the data transfer on the PCI bus.

The ATU does not support outbound transactions generated by the DMA controller.

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.

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 ATU implements 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” on page 16-4
- The address windowing mechanism for outbound translation is described in Section 16.3.6, “Outbound Address Translation” on page 16-8

The primary ATU contains a data path between the primary PCI bus and 80960 local bus.

The ATU unit allows for recognition and generation of multiple PCI cycle types. Table 16-1 shows the PCI commands supported by both inbound and outbound ATU. The type of operation seen by the inbound ATU is determined by the PCI master 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 ATU is determined by the 80960 local address and the fixed outbound windowing scheme. See Section 16.3.6, “Outbound Address Translation” on page 16-8 for the full details on outbound PCI cycle selection.

The ATU does not support exclusive access using the PCI LOCK# signal. To achieve exclusive access, use a software protocol or the Messaging Unit.

The ATU does 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	No

16.3.1 Inbound Address Translation

The ATU allows PCI bus masters to directly access the 80960 local bus. These PCI bus masters can read or write 80960VH 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.
   - — 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 ATU uses 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” on page 16-18 for details on inbound translation register definition and programming constraints.

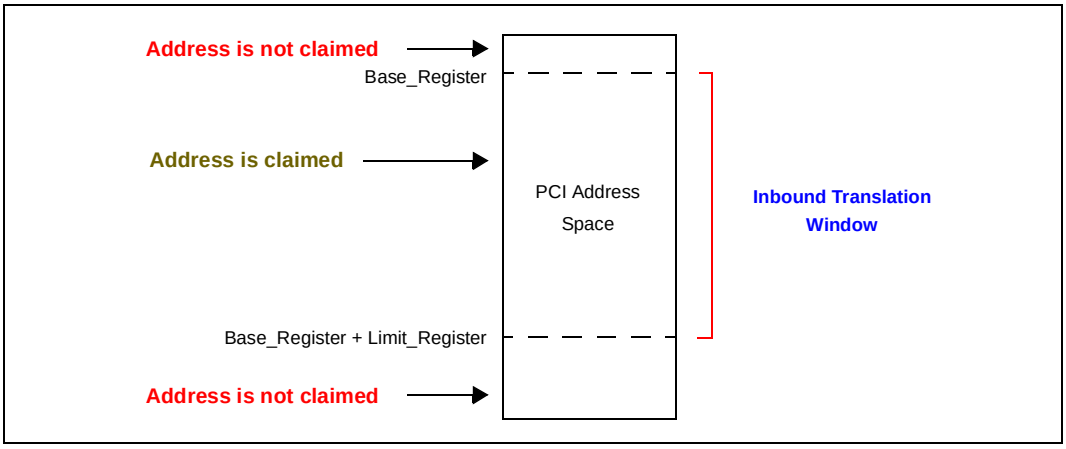
By convention, primary inbound ATU addresses are primary PCI addresses.

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" on page 16-15.

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

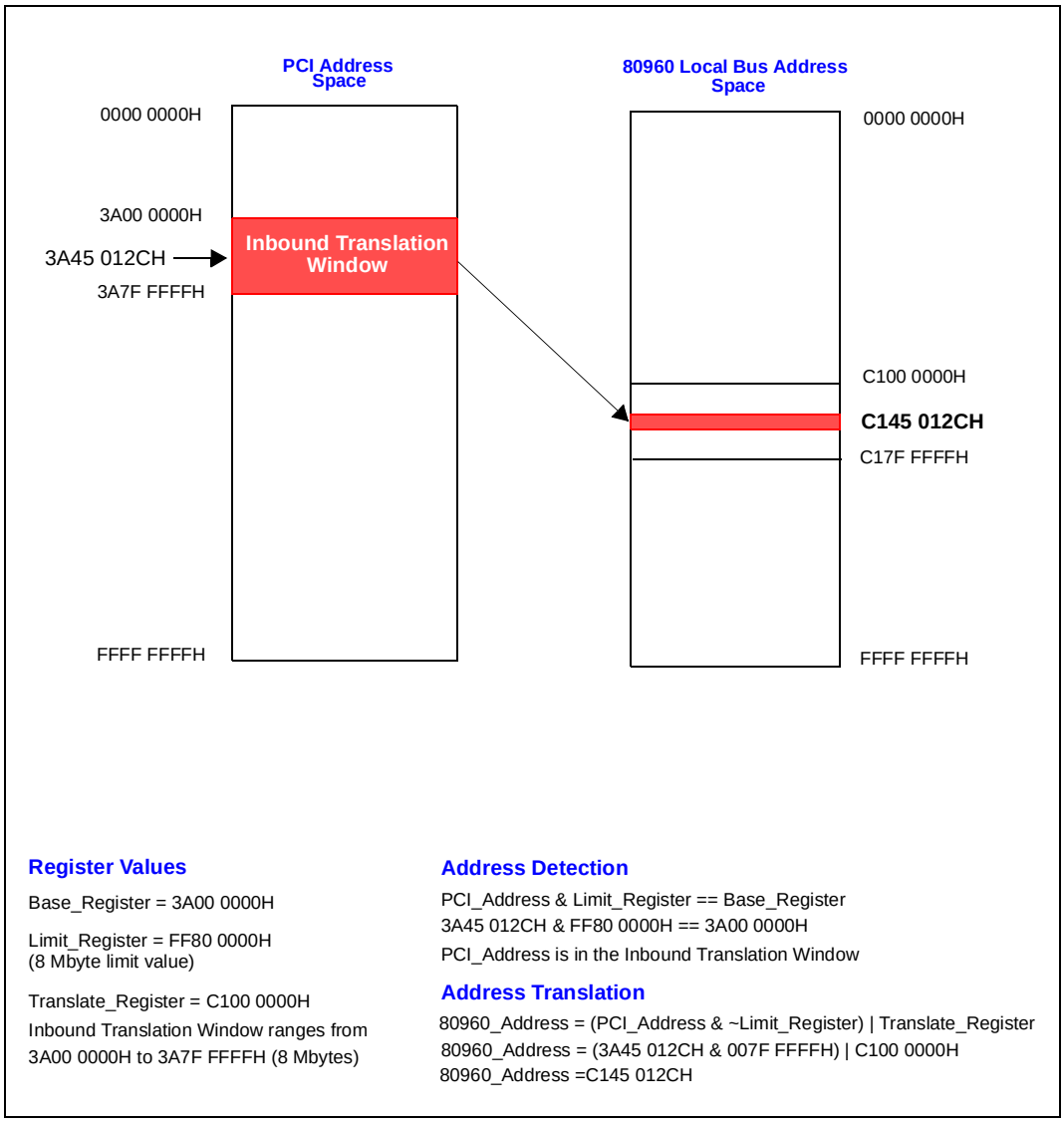
The incoming 32-bit PCI address is first bitwise ANDed with the bitwise inverse of the limit register. This result is bitwise ORed with the translate value register and the result is the 80960 local address. The translate value register must be aligned on the limit register boundary. For example, if the limit register is 8 Mbytes, then 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, "Inbound Configuration Cycle Translation" on page 16-8. 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-bit wide. See Section 13.1.1, "Physical Memory Attributes" on page 13-1. The only exception is the expansion ROM window can be in 8-bit wide memory.

Figure 16-4 shows an inbound translation example.

Figure 16-4. Inbound Translation Example



16.3.2 Inbound Write Transaction

An inbound write transaction is initiated by a PCI master 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" on page 16-3.
- 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 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-word 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 zero.

The ATU configuration space can be accessed using PCI configuration cycles from the primary PCI bus using function 0 configuration space. All inbound configuration cycles are processed as delayed transactions.

16.3.5 Discard Timers

The ATU implements a discard timer for inbound delayed transactions. The timer prevents 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 Core Select Register (Section 11.2.3).

16.3.6 Outbound Address Translation

In addition to providing the mechanism for inbound translation, the ATU translates 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 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 ATU supports two different outbound translation modes:

- Address Translation Windows
- Direct Addressing Window

Figure 16-5 shows a 80960VH memory map with all reserved address locations highlighted. The outbound translation windows exist from 8000 0000H to 9001 FFFFH. This is a 64 Mbyte window and a 64 Kbyte window. 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. When the Bus Master Enable bit is clear or the Outbound ATU Enable (bit 1 of the ATUCR) is clear, the ATU does 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. The ATU has two windows- one is 64 Mbyte and one is 64 Kbyte. The primary outbound memory window range from 8000 0000H to 83FF FFFFH (64 Mbyte). After this window, the primary outbound I/O window range from 9000 0000H to 9000 FFFFH (64 Kbyte).

The memory window is 64 Mbytes and the 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 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.

The 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

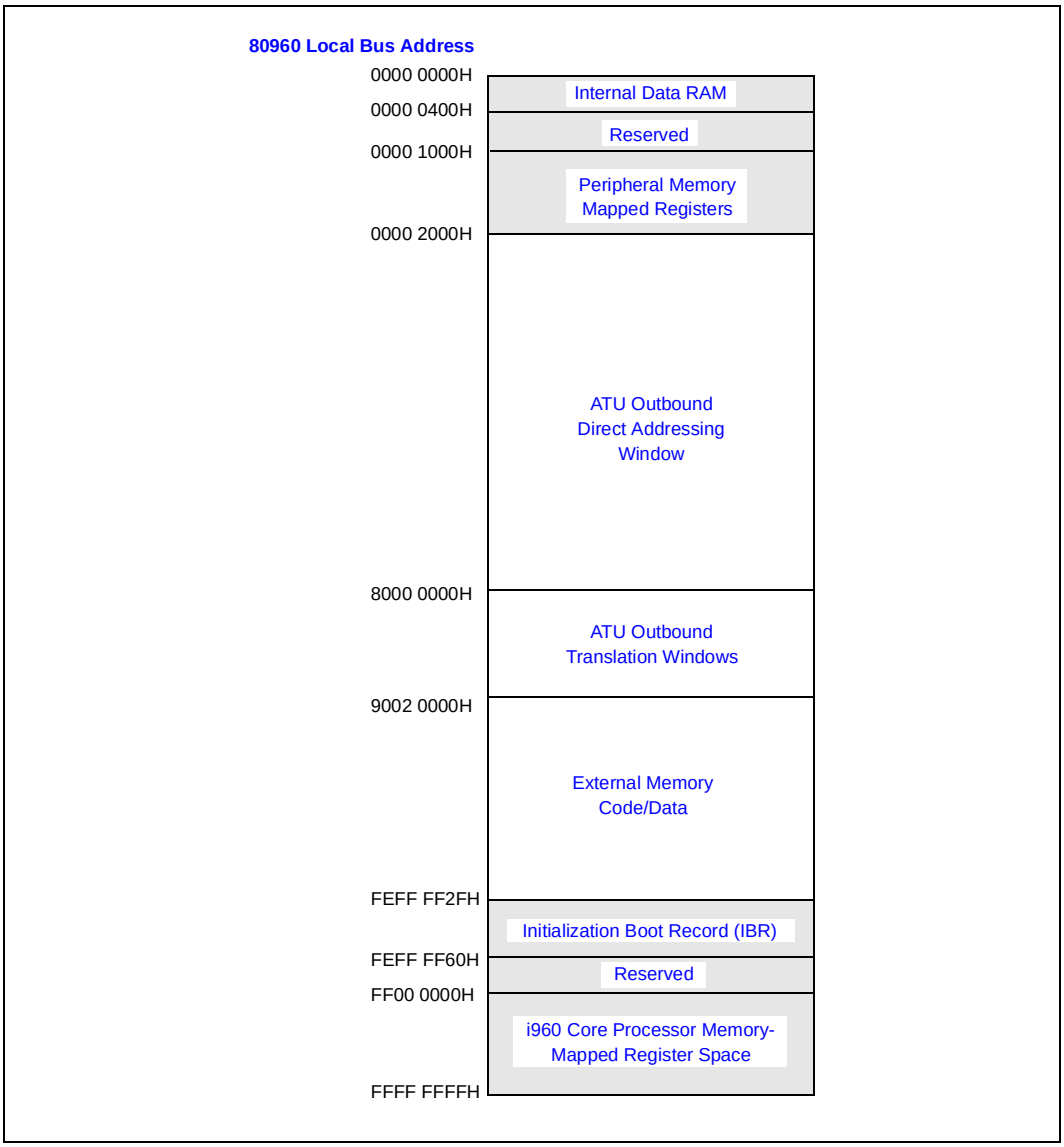
Refer to Figure 16-6 for the sub-window addresses involved in the outbound translation.

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.

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 ATU uses the following registers in outbound address translation:

- Outbound Memory Window Value Register
- Outbound I/O Window Value Register
- Outbound Configuration Cycle Address Register

See Section 16.7, “Register Definitions” on page 16-18 for details on outbound translation register definition and programming constraints.

The translation algorithm used, as stated, is very similar to inbound translation. For memory transactions the algorithm is:

```
PCI_Address = (80960_Address & 03FF FFFFH) | Translate_Register
```

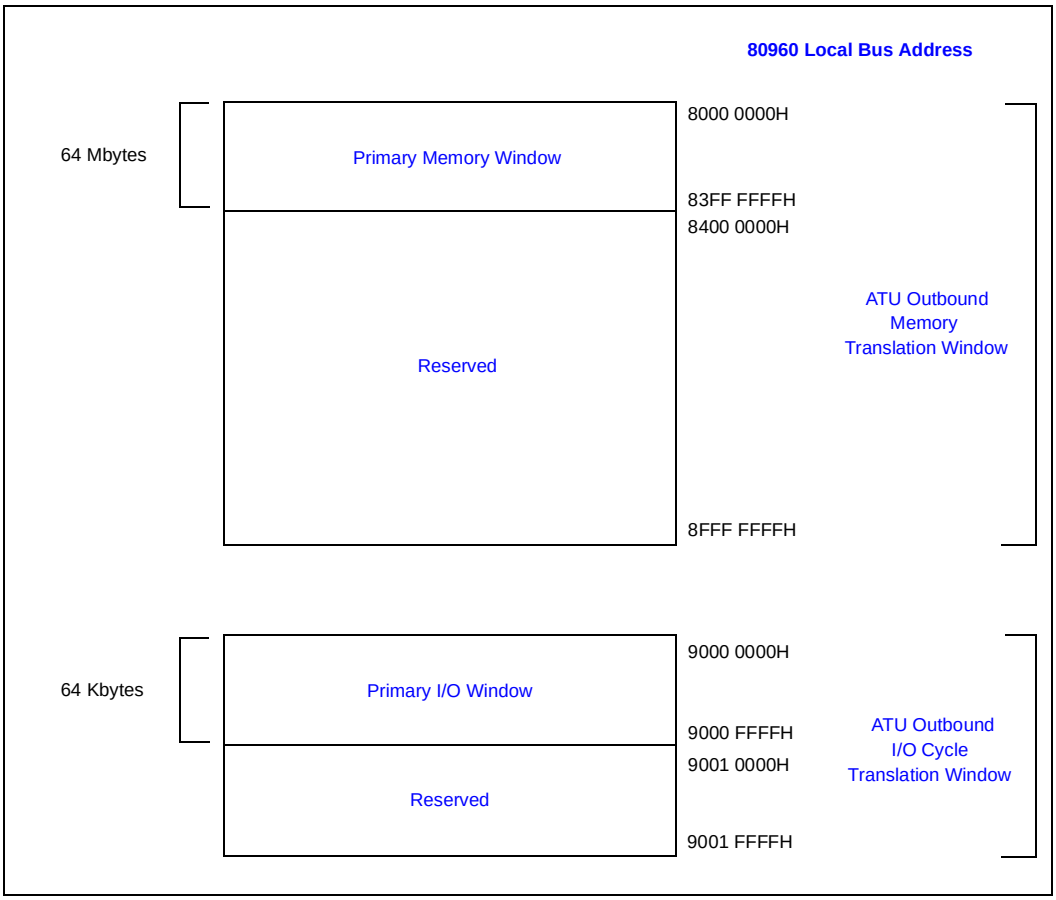
For memory transactions, the 80960 local bus address is bitwise ANDed with the inverse of 64 Mbytes which clears the upper 6 bits of address. The result is bitwise ORed with the outbound window value register to create the 32-bit PCI address. The translate value must be aligned on a 64 Mbyte boundary.

```
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. The translate value must be aligned on a 64 Kbyte boundary. 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

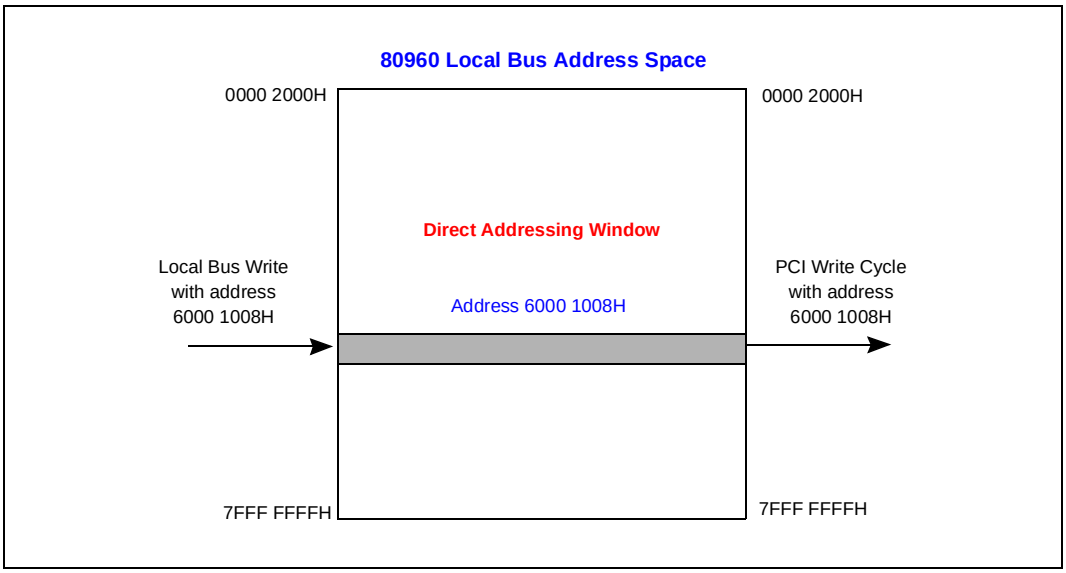


16.3.6.2 Direct Addressing Window

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, 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.

16.3.7 Outbound Write Transaction

An outbound write transaction is initiated by the i960 core processor and is targeted at a PCI slave. 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 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.

16.3.8 Outbound Read Transaction

An outbound read transaction is initiated by the i960 core processor and is targeted at a PCI slave. 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.

16.3.9 Outbound Configuration Cycle Translation

The outbound ATU provides a port programming model for outbound configuration cycles. Performing an outbound configuration cycle involves up to two 80960 local bus cycles:

1. Writing the Outbound Configuration Cycle Address Register 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. The i960 core processor cycle initiates the transaction. A read causes a configuration cycle read to the Primary PCI bus with the address in the outbound configuration cycle address register. Similarly, a write initiates a configuration cycle write to the Primary 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. 16-18) 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 80960VH 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. 16-31), Section 16.7.24 (pg. 16-38) and Section 16.7.25 (pg. 16-38).

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” on page 16-4. The ROM width supported is 8 bit only.

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 15, “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 Command Register. See Section 16.7, “Register Definitions” on page 16-18.

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.

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	80960VH 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	80960VH 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#
	80960VH 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 table (Table 16-6) summarizes 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 ATU Status Register records 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 ATU Interrupt Status Register records 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

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 Master Abort	Master Abort bit (bit 13) set	PCI Master Abort bit (bit 3) set	Yes
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

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. The ATU configuration space is function number zero of the 80960VH 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”.

Figure 16-8. ATU Configuration Space Header

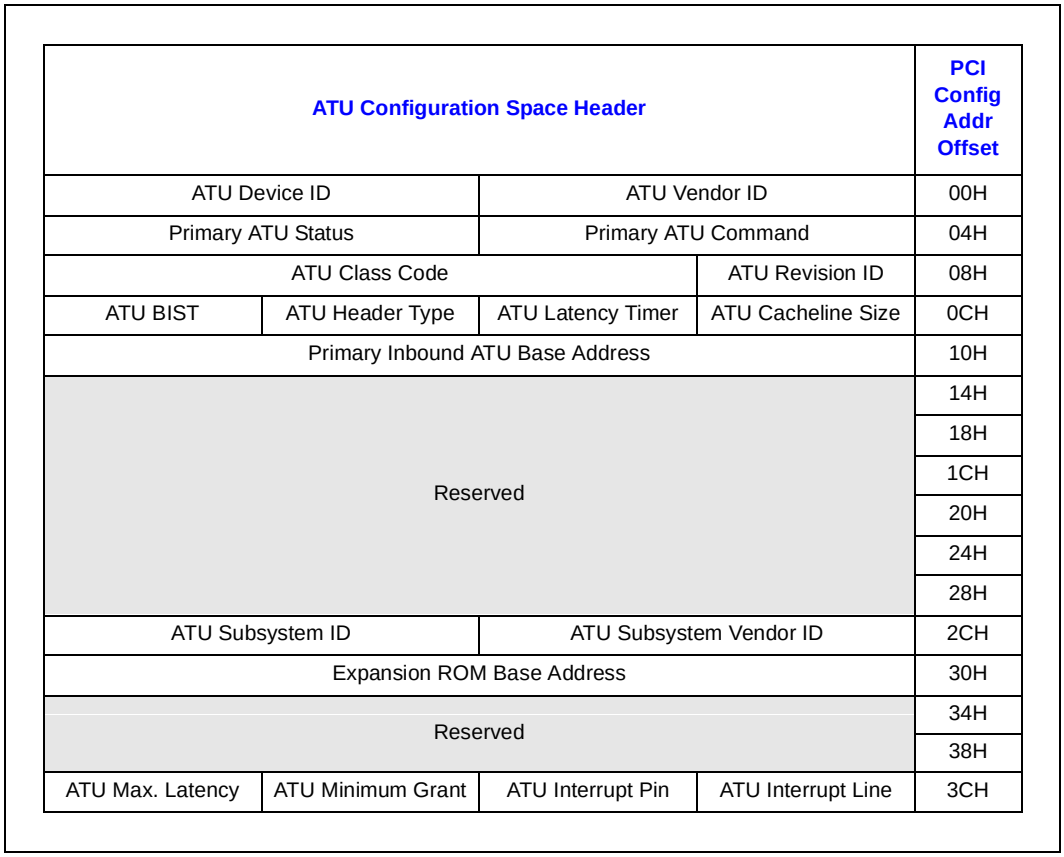


Table 16-7. ATU Configuration Space Register Summary (Sheet 1 of 3)

Section	Register Name and Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
16.7.1	ATU Vendor ID Register - ATUVID	16-21	16	0000 1200H	00H
16.7.2	ATU Device ID Register - ATUDID	16-22	16	0000 1202H	02H
16.7.3	Primary ATU Command Register - PATUCMD	16-22	16	0000 1204H	04H
16.7.4	Primary ATU Status Register - PATUSR	16-23	16	0000 1206H	06H
16.7.5	ATU Revision ID Register - ATURID	16-24	8	0000 1208H	08H
16.7.6	ATU Class Code Register - ATUCCR	16-25	24	0000 1209H	09H
16.7.7	ATU Cacheline Size Register - ATUCLSR	16-25	8	0000 120CH	0CH
16.7.8	ATU Latency Timer Register - ATULT	16-26	8	0000 120DH	0DH
16.7.9	ATU Header Type Register - ATUHTR	16-26	8	0000 120EH	0EH
16.7.10	ATU BIST Register - ATUBISTR	16-27	8	0000 120FH	0FH

Table 16-7. 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	16-28	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	16-30	16	0000 122CH	2CH
16.7.14	ATU Subsystem ID Register - ASIR	16-31	16	0000 122EH	2EH
16.7.15	Expansion ROM Base Address Register - ERBAR	16-31	32	0000 1230H	30H
	Reserved		32	0000 1234H	34H
			32	0000 1238H	38H
16.7.16	ATU Interrupt Line Register - ATUILR	16-32	8	0000 123CH	3CH
16.7.17	ATU Interrupt Pin Register - ATUIPR	16-33	8	0000 123DH	3DH
16.7.18	ATU Minimum Grant Register - ATUMGNT	16-34	8	0000 123EH	3EH
16.7.19	ATU Maximum Latency Register - ATUMLAT	16-34	8	0000 123FH	3FH
16.7.20	Primary Inbound ATU Limit Register - PIALR	16-35	32	0000 1240H	40H
16.7.21	Primary Inbound ATU Translate Value Register - PIATVR	16-36	32	0000 1244H	44H
	Reserved		32	0000 1248H	48H
	Reserved		32	0000 124CH	4CH
	Reserved		32	0000 1250H	50H
16.7.22	Primary Outbound Memory Window Value Register - POMWVR	16-36	32	0000 1254H	54H
	Reserved		32	0000 1258H	58H
16.7.23	Primary Outbound I/O Window Value Register - POIOWVR	16-37	32	0000 125CH	5CH
	Reserved		32	0000 1260H	60H
	Reserved		32	0000 1264H	64H
	Reserved		32	0000 1268H	68H
	Reserved		32	0000 126CH	6CH
	Reserved		32	0000 1270H	70H
16.7.24	Expansion ROM Limit Register - ERLR	16-38	32	0000 1274H	74H
16.7.25	Expansion ROM Translate Value Register - ERTVR	16-38	32	0000 1278H	78H

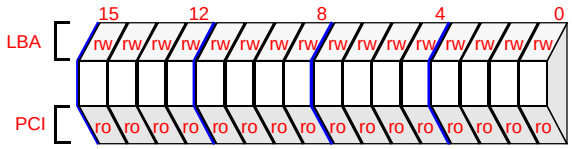
Table 16-7. ATU Configuration Space Register Summary (Sheet 3 of 3)

Section	Register Name and Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
	Reserved		32	0000 127CH	7CH
			32	0000 1280H	80H
			32	0000 1284H	84H
16.7.26	ATU Configuration Register - ATUCR	16-39	32	0000 1288H	88H
	Reserved		32	0000 128CH	8CH
16.7.27	Primary ATU Interrupt Status Register - PATUISR	16-40	32	0000 1290H	90H
	Reserved		32	0000 1294H	94H
	Reserved		32	0000 1298H	98H
	Reserved		32	0000 129CH	9CH
	Reserved		32	0000 12A0H	A0H
16.7.28	Primary Outbound Configuration Cycle Address Register - POCCAR	16-41	32	0000 12A4H	A4H
	Reserved		32	0000 12A8H	A8H
16.7.29	Primary Outbound Configuration Cycle Data Port - POCCDP	16-42	32	0000 12ACH	ACH
	Reserved		32	0000 12B0H	B0H
	Reserved		32	0000 12B4H	B4H
	Reserved		32	0000 12B8H	B8H
	Reserved		32	0000 12BCH	BCH
	Reserved		32	0000 12C0H	C0H
16.7.30	Reset/Retry Control Register - RRCR	16-42	32	0000 12C4H	C4H
16.7.31	PCI Interrupt Routing Select Register PIRSR	16-42	32	0000 12C8H	C8H
16.7.32	Core Select Register - CSR	16-43	32	0000 12CCH	CCH
	Reserved		0000 12D0H 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 16-8. ATU Vendor ID Register - ATUVID



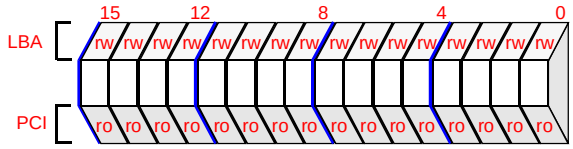
LBA: 1200H **PCI:** 00H	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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 16-9. ATU Device ID Register - ATUDID



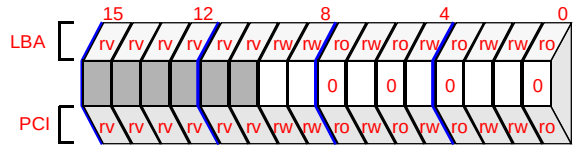
LBA: 1202H **PCI:** 02H	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:00	6960H	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 16-10. Primary ATU Command Register - PATUCMD



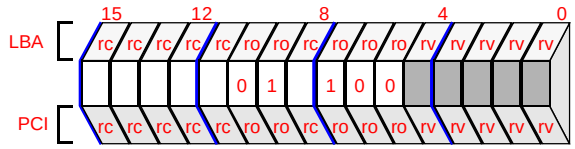
LBA: 1204H **PCI:** 04H	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
15:10	00H	Reserved.
09	0_2	Fast Back to Back Enable - Allows the ATU to generate fast back-to-back cycles on its bus. Not implemented and a reserved bit field.
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 16-11. Primary ATU Status Register - PATUSR



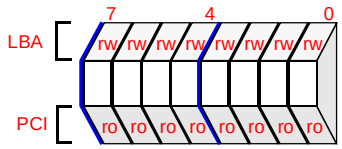
LBA: 1206H **PCI:** 06H	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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	0_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. Not implemented and a reserved bit field.
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 16-12. ATU Revision ID Register - ATURID



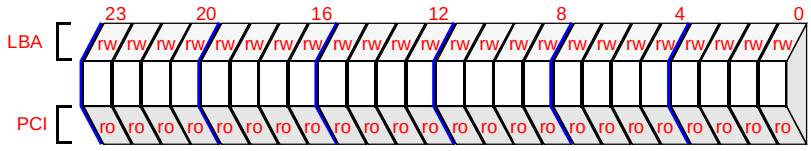
LBA: 1208H **PCI:** 08H		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
07:00	XXH	ATU Revision - Identifies the 80960VH's revision number.[1]

1. These numbers vary with stepping, refer to the *i960® VH Processor Specification Update (273174)* 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 16-13. ATU Class Code Register - ATUCCR

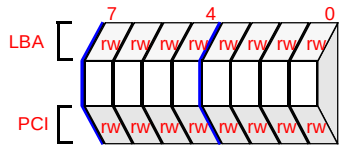


LBA: 1209H **PCI:** 09H		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
23:16	05H	Base Class - Memory Controller
15:08	80H	Sub 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".

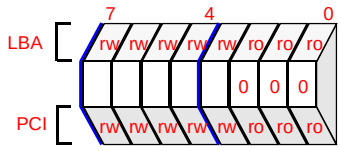
Table 16-14. ATU Cacheline Size Register - ATUCLSR



LBA: 120CH **PCI:** 0CH		**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	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

Table 16-15. ATU Latency Timer Register - ATULT

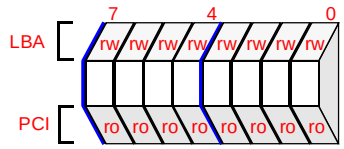


LBA: 120DH **PCI:** 0DH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
07:03	0H	Programmable Latency Timer - This field varies 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 16-16. ATU Header Type Register - ATUHTR

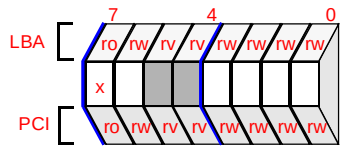


LBA: 120EH **PCI:** 0EH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
07	0_2	Single Function/Multi-Function Device - Identifies the 80960VH as a single 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

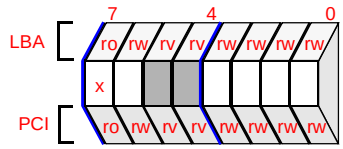
The ATU BIST Register controls the functions the i960 core processor performs when BIST is initiated. This register is the interface between the host processor requesting BIST functions and the i960 core processor replying with the results from the software implementation of BIST functionality.

Table 16-17. ATU BIST Register - ATUBISTR (Sheet 1 of 2)



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 16.7.28, 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.

Table 16-17. ATU BIST Register - ATUBISTR (Sheet 2 of 2)



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
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.

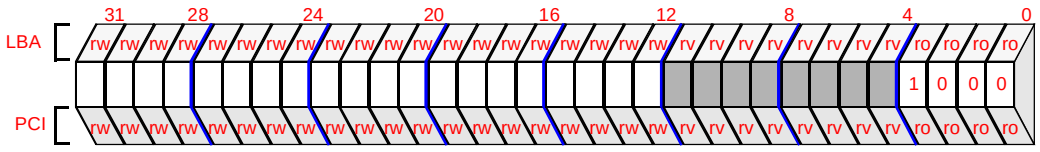
16.7.11 Primary Inbound ATU Base Address Register - PIABAR

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 16-29.

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.

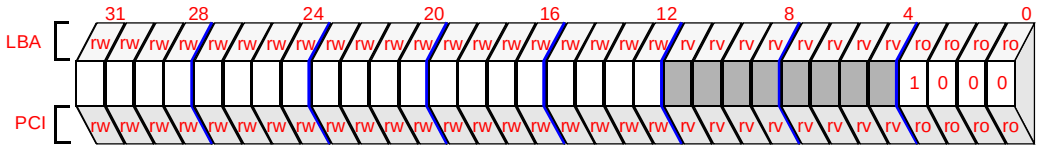
Table 16-18. Primary Inbound ATU Base Address Register - PIABAR (Sheet 1 of 2)



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.

Table 16-18. Primary Inbound ATU Base Address Register - PIABAR (Sheet 2 of 2)



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
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 Inbound ATU Base Address Register and Expansion ROM Base Address Register use their associated limit registers for 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 16-19 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 16-19 describes the relationship between the values read back and the byte sizes the base address register requires.

Table 16-19. Instructions for Base Address Register

Device Part Number	Value Written to the BAR	Effect of writing the value to the Base Address Register
80960VH 100/3.3 A-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 16-20. Any access to the base address register can be performed as on 8-, 16-, or 32-bit access.

Table 16-20. Memory Block Size Read Response

Response After Writing all 1's 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
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 0008H. Bit zero is a zero, so the device requires memory address space. Bits 2:1 are 00_2, 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.

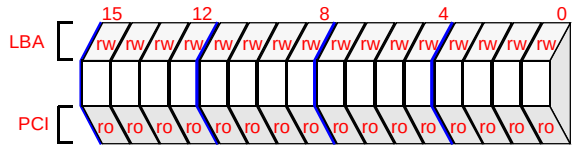
Table 16-21. Base Address and Limit Register Descriptions

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.
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 16-22. ATU Subsystem Vendor ID Register - ASVIR

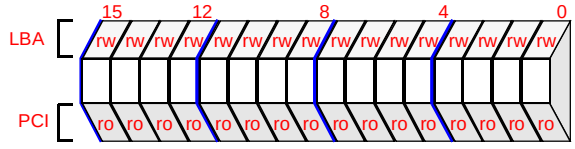


LBA: 122CH **PCI:** 2CH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
15:00	0000H	Subsystem Vendor ID - This register uniquely identifies the add-in board or subsystem vendor.

16.7.14 ATU Subsystem ID Register - ASIR

ATU Subsystem ID Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1.

Table 16-23. ATU Subsystem ID Register - ASIR



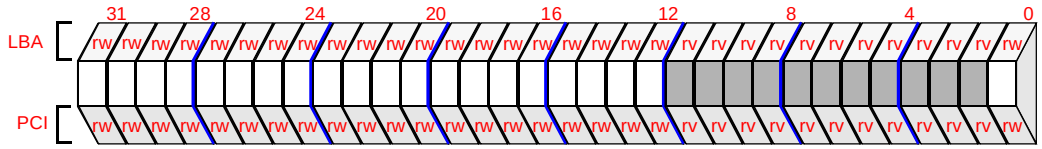
LBA: 122EH **PCI:** 2EH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
15:00	0000H	Subsystem ID - uniquely identifies the add-in board or subsystem

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 16-29. 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 16-24. 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

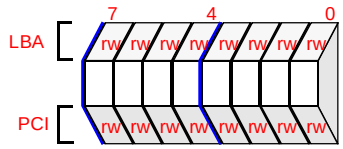
ATU Interrupt Line Register bit definitions adhere to *PCI Local Bus Specification*, revision 2.1. This register identifies the system interrupt controller's interrupt request lines which connect to the device's PCI interrupt request lines (as specified in the interrupt pin register).

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 16-25. ATU Interrupt Line Register - ATUILR

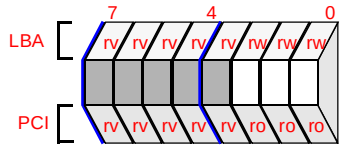


LBA: 123CH **PCI:** 3CH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
07:00	FFH	Interrupt Assigned - system-assigned value 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).

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 80960VH 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 16-26. ATU Interrupt Pin Register - ATUIPR



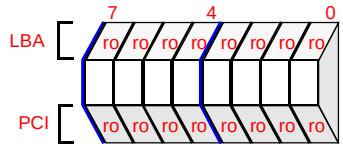
LBA: 123DH **PCI:** 3DH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
07:03	00H	Reserved.
02:00	001_2	Interrupt Used - Selects 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.

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 16-27. 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

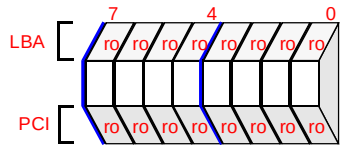
Bit	Default	Description
07:00	00H	This register specifies how long a burst period the device needs in increments of 8 PCI clocks. A zero value indicates the device has no stringent requirement.

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 16-28. ATU Maximum Latency Register - ATUMLAT



LBA: 123FH **PCI:** 3FH		**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	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.

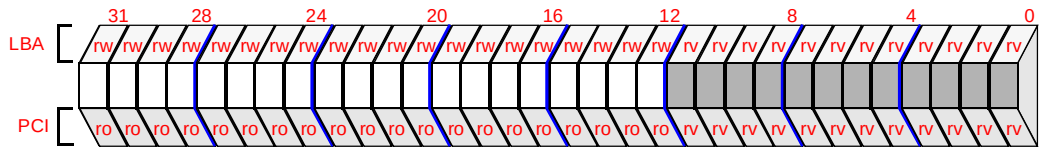
16.7.20 Primary Inbound ATU Limit Register - PIALR

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, “Primary Inbound ATU Base Address Register - PIABAR” on page 16-28. When determining block size requirements — as described in Section 16.7.12, “Determining Block Sizes for Base Address Registers” on page 16-29 — 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, “Inbound Address Translation” on page 16-4.

Table 16-29. Primary Inbound ATU Limit Register - PIALR

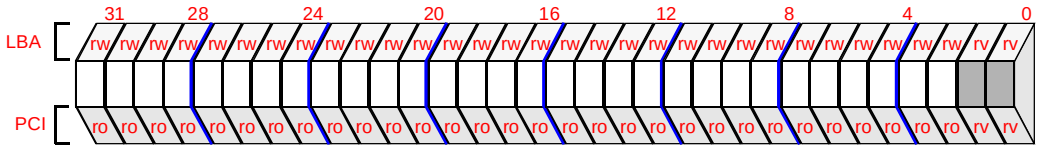


LBA: 1240H **PCI:** 40H		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:12	FFFF FH	Primary Inbound Translation Limit - This value (Table 16-21) 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 16-30. Primary Inbound ATU Translate Value Register - PIATVR



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

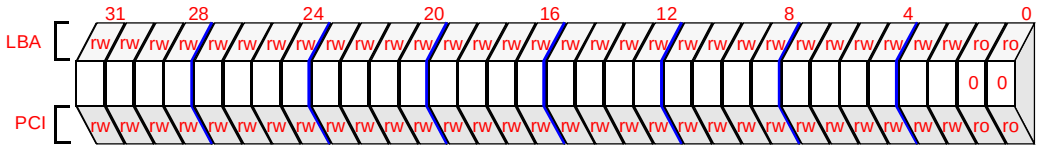
Bit	Default	Description
31:02	0000 100H	Primary Inbound ATU Translation Value - This value is used to convert the primary PCI address to local addresses. This value must be word-aligned on the 80960 local bus. The default address allows the ATU to access the internal 80960 local bus memory-mapped registers.
01:00	00_2	Reserved.

16.7.22 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, “Outbound Address Translation” on page 16-8 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 16-31. Primary Outbound Memory Window Value Register - POMWVR



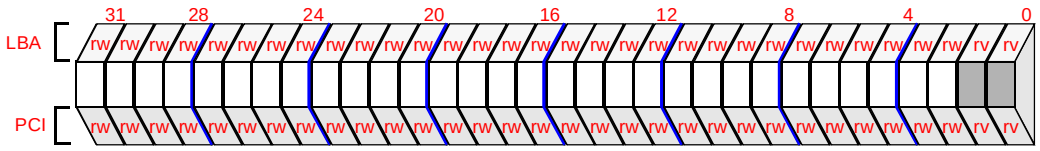
LBA: 1254H **PCI:** 54H		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:02	0000 0000 H	Primary Outbound MW Value - Used to 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.

16.7.23 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” on page 16-8 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 16-32. Primary Outbound I/O Window Value Register - POIOWVR



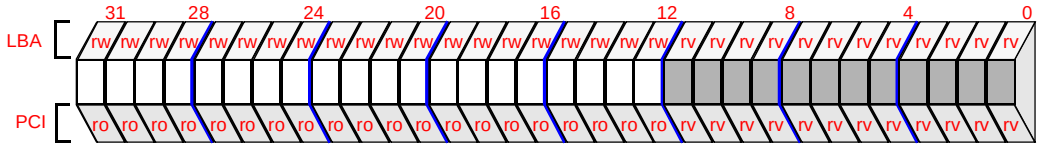
LBA: 125CH **PCI:** 5CH		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:02	0000 0000 H	Primary Outbound I/O Window Value - Used to convert local addresses to PCI addresses.
01:00	00_2	Reserved.

16.7.24 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” on page 16-31. When determining the block size requirements, the Expansion ROM Limit Register provides the block size requirements for the Expansion ROM Base Address Register.

Table 16-33. Expansion ROM Limit Register - ERLR



LBA: 1274H
PCI: 74H

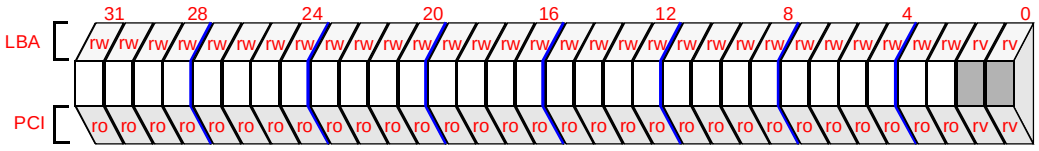
Legend: NA = Not Accessible RO = Read Only
RV = Reserved PR = Preserved RW = Read/Write
RS = Read/Set RC = Read Clear
LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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.

16.7.25 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 16-34. Expansion ROM Translate Value Register - ERTVR



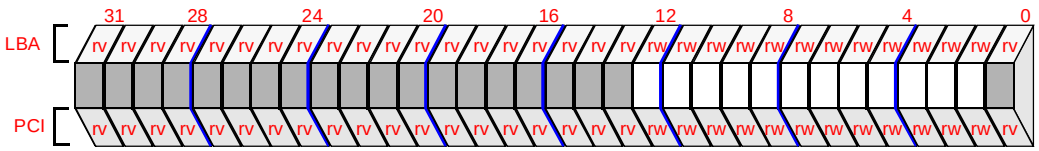
LBA: 1278H **PCI:** 78H	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:02	0000 0000 H	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.

16.7.26 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 and contains a bit for Expansion ROM width.

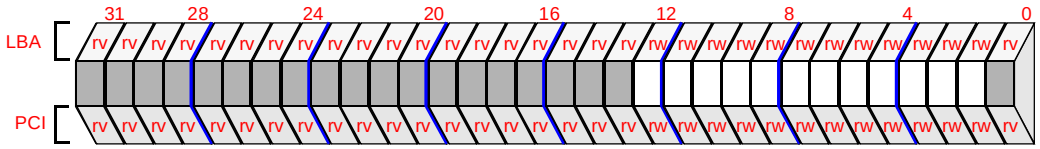
Table 16-35. ATU Configuration Register - ATUCR (Sheet 1 of 2)



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
31:10	0000 00H	Reserved.
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 ATU. Local bus cycles with an address between 0000 2000H and 7FFF FFFFH are automatically forwarded to the PCI bus with no address translation.
07	0_2	Reserved.
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). The i960 VH processor supports 8 bit Expansion ROM only. This bit must always be zero (0).

Table 16-35. ATU Configuration Register - ATUCR (Sheet 2 of 2)



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

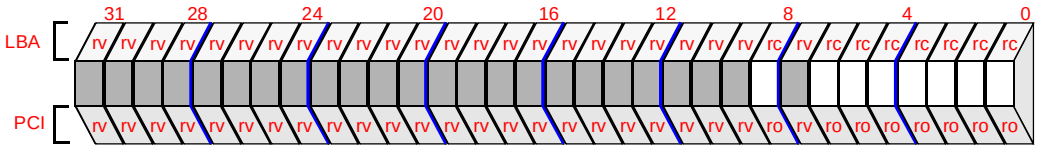
Bit	Default	Description
05	0_2	Reserved.
04	0_2	Primary ATU PCI Error Interrupt Enable - This bit acts as a mask for Primary ATU Interrupt Status Register bits 4:0. When set, enables an interrupt to the i960 core processor when any of these bits are set in the PATUISR. When cleared, disables the interrupt.
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	Reserved.
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.

16.7.27 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 16-36. Primary ATU Interrupt Status Register - PATUISR



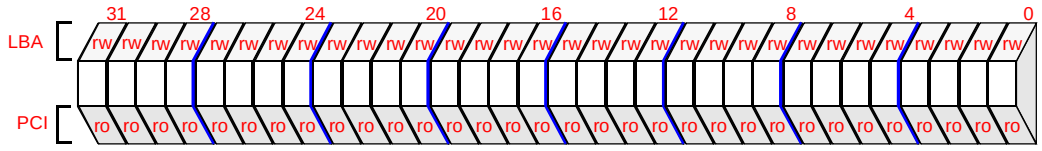
LBA: 1290H **PCI:** 90H		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:09	0000 00H	Reserved.
08	0_2	ATU BIST Interrupt - When set, the host processor has set the start BIST, ATUBISTR register bit 6, and the ATU BIST interrupt enable, ATUCR register bit 3, 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.
00	0_2	PCI Master Parity Error - The ATU interface sets this bit when three conditions are met: • bus agent asserted S_PERR# • 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)

16.7.28 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 16-37. Primary Outbound Configuration Cycle Address Register - POCCAR



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 0000 H	Primary Configuration Cycle Address - These bits define the 32-bit PCI address used during an outbound configuration read or write cycle.

16.7.29 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 0 cycles or ATU inbound transactions.

16.7.30 Reset/Retry Control Register - RRCR

Refer to Section 11.2.1, “Reset/Retry Control Register - RRCR” on page 11-1.

16.7.31 PCI Interrupt Routing Select Register PIRSR

Refer to Section 11.2.2, “PCI Interrupt Routing Select Register - PIRSR” on page 11-2.

16.7.32 Core Select Register - CSR

Refer to Section 11.2.3, “Core Select Register - CSR” on page 11-2.

16.8 Powerup/Default Status

The default/powerup values for all registers are shown within each register description.

16.9 Reset Modes

See Section 11.2.1, “Reset/Retry Control Register - RRCR” on page 11-1.

Messaging Unit 17

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 two messaging mechanisms. Each allows a host processor or external PCI agent and the i960® VH processor to communicate through message passing and interrupt generation. Each mechanism and corresponding sections are summarized as:

- Section 17.2, “Message Registers” on page 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” on page 17-2. These two registers support software interrupts. Interrupts are generated when a Doorbell Register bit is set.

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 (ATU). This PCI address window is used for PCI transactions that access the 80960VH’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” on page 16-3 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 two 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

17.2 Message Registers

The 80960VH 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 80960VH.
- Outbound messages are sent by the 80960VH 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.

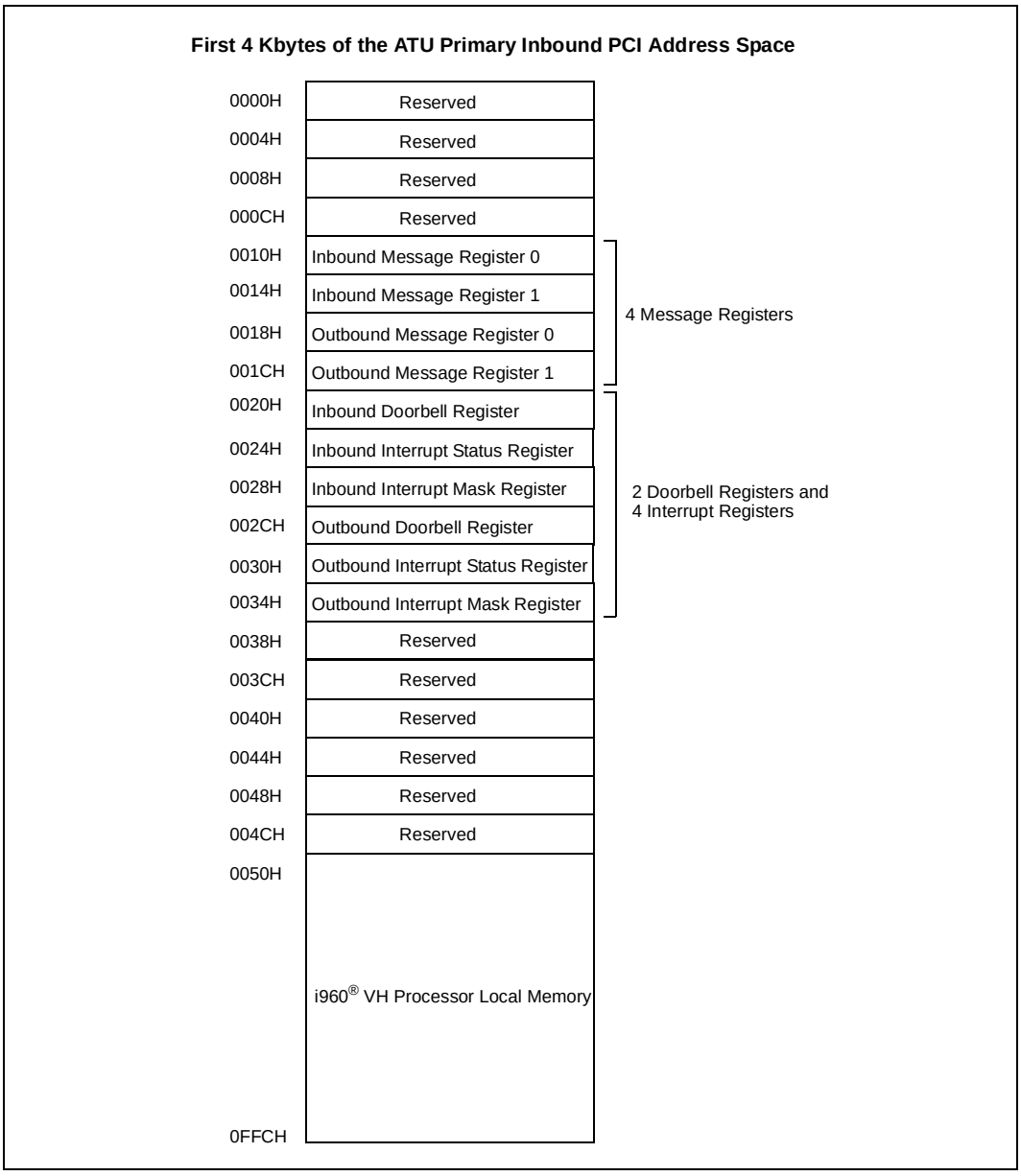
One bit in the Inbound Doorbell Register is reserved for an NMI interrupt.

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 Register Definitions

Figure 17-1 shows the PCI memory map and identifies the first 4 Kbytes of ATU Primary Inbound PCI address space. Registers in Table 17-2 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.

Figure 17-1. PCI Memory Map



Registers in Table 17-2 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.

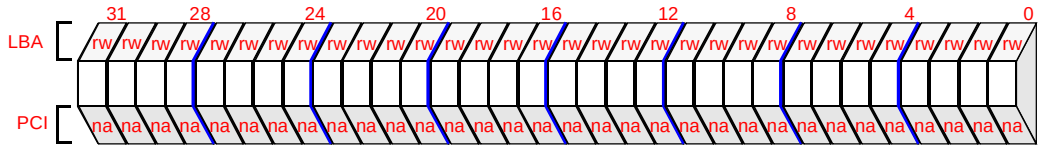
Table 17-2. Peripheral Memory-Mapped Register Summary

Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
	Reserved		32	0000 1300H	NA
	Reserved		32	0000 1308H	NA
17.4.1	Inbound Message Registers - IMRx	17-5	32	0 - 0000 1310H 1 - 0000 1314H	NA
17.4.2	Outbound Message Registers - OMRx	17-6	32	0 - 0000 1318H 1 -0000 131CH	NA
17.4.3	Inbound Doorbell Register - IDR	17-6	32	0000 1320H	NA
17.4.4	Inbound Interrupt Status Register - IISR	17-7	32	0000 1324H	NA
17.4.5	Inbound Interrupt Mask Register - IIMR	17-8	32	0000 1328H	NA
17.4.6	Outbound Doorbell Register - ODR	17-9	32	0000 132CH	NA
17.4.7	Outbound Interrupt Status Register - OISR	17-10	32	0000 1330H	NA
17.4.8	Outbound Interrupt Mask Register - OIMR	17-11	32	0000 1334H	NA
	Reserved		32	0000 1350H	NA
	Reserved		32	0000 1354H	NA
	Reserved		32	0000 1360H	NA
	Reserved		32	0000 1364H	NA
	Reserved		32	0000 1368H	NA
	Reserved		32	0000 136CH	NA
	Reserved		32	0000 1370H	NA
	Reserved		32	0000 1374H	NA
	Reserved		32	0000 1378H	NA
	Reserved		32	0000 137CH	NA
	Reserved		32	0000 1380H	NA

17.4.1 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 17-3. Inbound Message Register - IMRx



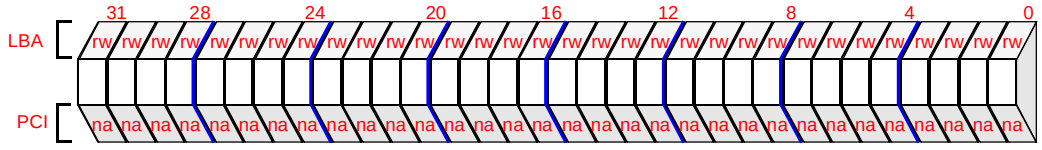
LBA: CH. 0 = 1310H CH. 1 = 1314H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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.

17.4.2 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 17-4. Outbound Message Register - OMRx



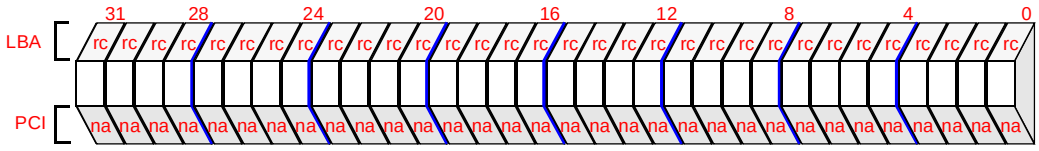
LBA: CH. 0 = 1318H CH. 1 = 131CH **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	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.

17.4.3 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" on page 8-20.

Table 17-5. Inbound Doorbell Register - IDR



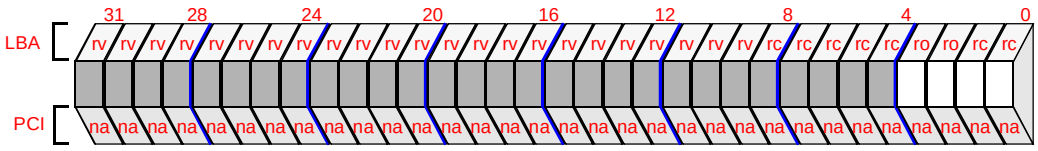
LBA: 1320H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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.

17.4.4 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, and the Doorbell Registers. All interrupts are routed to the i960 core processor's XINT7 interrupt input, except for the NMI Doorbell Interrupt which is 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.

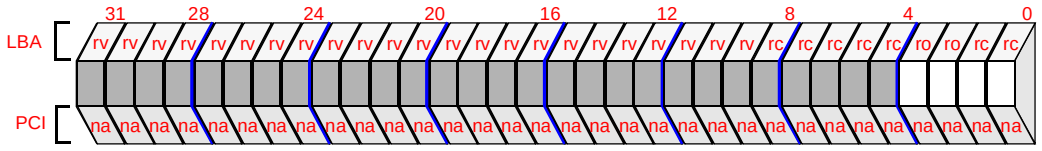
Table 17-6. Inbound Interrupt Status Register - IISR (Sheet 1 of 2)



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	Reserved
07	0_2	Reserved
06	0_2	Reserved
05	0_2	Reserved
04	0_2	Reserved

Table 17-6. Inbound Interrupt Status Register - IISR (Sheet 2 of 2)



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

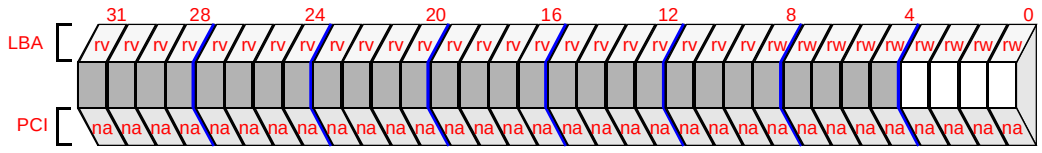
Bit	Default	Description
03	0_2	NMI Doorbell Interrupt - set when the Inbound Doorbell Register NMI Interrupt is set. To clear this bit (and the interrupt), the Inbound Doorbell Register NMI Interrupt bit in the Inbound Doorbell Register must be clear.
02	0_2	Inbound Doorbell Interrupt - set when at least 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.
00	0_2	Inbound Message 0 Interrupt - set when the Inbound Message 0 Register has been written.

17.4.5 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.

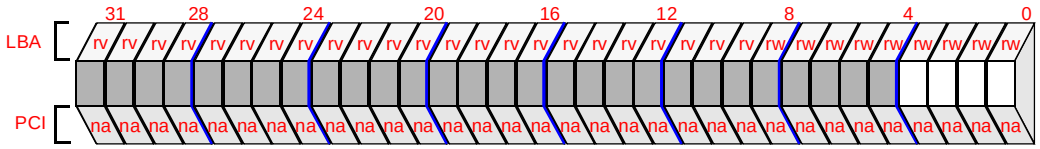
Table 17-7. Inbound Interrupt Mask Register - IIMR (Sheet 1 of 2)



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	Reserved
07	0_2	Reserved

Table 17-7. Inbound Interrupt Mask Register - IIMR (Sheet 2 of 2)



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

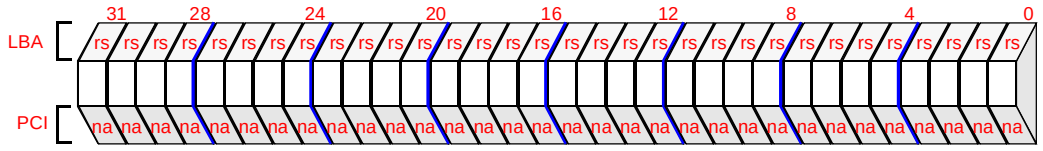
Bit	Default	Description
06	0_2	Reserved
05	0_2	Reserved
04	0_2	Reserved
03	0_2	NMI Doorbell Interrupt Mask - When set, this bit masks the NMI Interrupt when the Inbound Doorbell Register NMI Interrupt bit is set.
02	0_2	Inbound Doorbell Interrupt Mask - When set, this bit masks the interrupt generated 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.

17.4.6 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 17-8. Outbound Doorbell Register - ODR



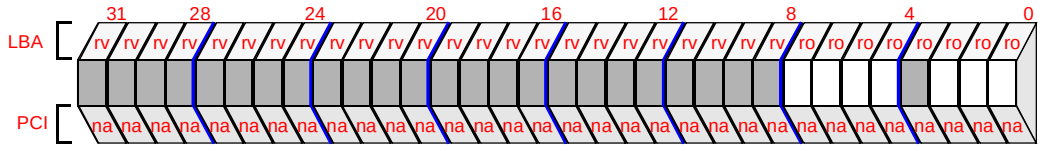
LBA: 132CH **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

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.

17.4.7 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, and Doorbell Registers. 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.

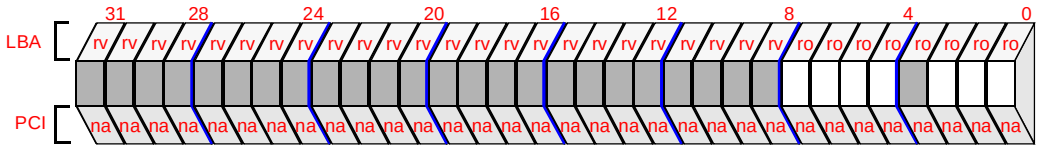
Table 17-9. Outbound Interrupt Status Register - OISR (Sheet 1 of 2)



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.

Table 17-9. Outbound Interrupt Status Register - OISR (Sheet 2 of 2)



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

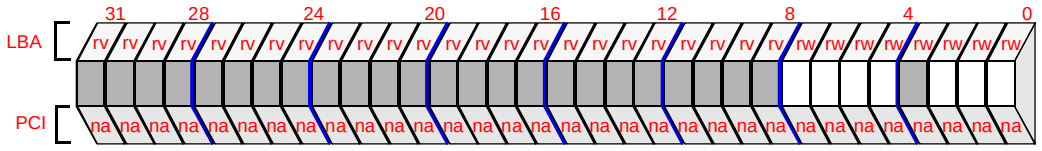
Bit	Default	Description
07	0_2	PCI Interrupt D - set when the PCI Interrupt D bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt D bit in the Outbound Doorbell Register must be cleared.
06	0_2	PCI Interrupt C - set when the PCI Interrupt C bit is set 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	Reserved
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.

17.4.8 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 17-10. Outbound Interrupt Mask Register - OIMR



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	Reserved
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

Bus Arbitration 18

This chapter describes the bus arbitration units of the i960® VH processor. The two arbitration units include: the internal local bus arbiter and the primary PCI bus arbiter for internal 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, and local bus backoff.

Chapter 1, "Introduction" contains a block diagram of the 80960VH in Figure 1-1, which shows the two arbitration units.

18.1 Overview

The 80960VH requires an arbitration mechanism to control 80960 local bus ownership. Bus masters connected to the local bus consist of:

- Two DMA channels
- Primary 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.

In addition to the local bus arbiter, the 80960VH contains an internal PCI arbitration unit. 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. The internal arbiters are not programmable.

18.2 Local Bus Arbitration Unit

The 80960 local bus arbitration unit supports up to five local bus masters. Table 18-1 shows the five 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
Primary 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

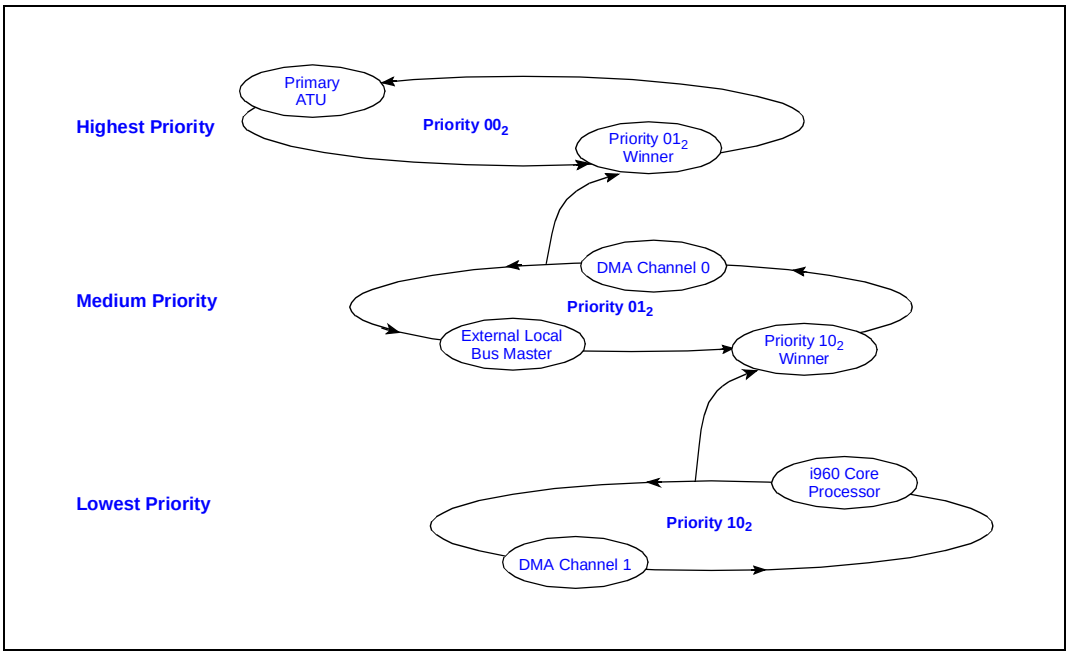


Table 18-3. Priority Programming for Local Bus Arbitration Example

Bus Master	Programmed Priority
Primary ATU	High - 00_2
DMA Channel 0	Medium - 01_2
External Bus Master	Medium - 01_2
DMA Channel 1	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	–	–	–

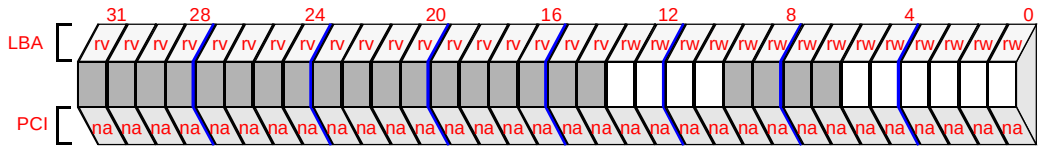
NOTE: In this example, all bus masters are continually requesting the bus.

The winning bus master pattern for the bus arbitration example in Table 18-4 would continue on as follows: **ABACABACABACABAC**.

18.2.1 Local Bus Arbitration Control Register - LBACR

The Local Bus Arbitration Control Register (LBACR - Table 18-5) 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 18-5. Local Bus Arbitration Control Register – LBACR



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	Reserved
07:06	00_2	Reserved
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” on page 18-5 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 ATU, 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 80960VH 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 80960VH does not respond to the HOLD signal (i.e., assert HOLDA) while the core processor is in reset. Refer to Section 14.6.1, “HOLD/HOLDA Protocol” on page 14-23 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 requests are 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.

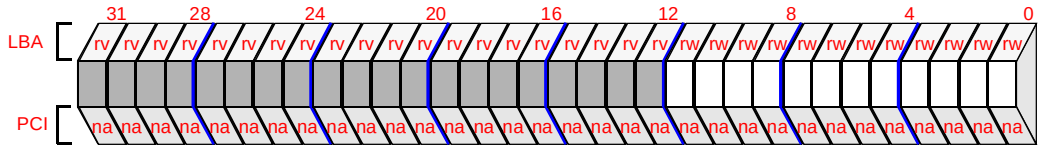
Table 18-6 shows the bit definitions for the local bus arbitration latency counter register.

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 18-6. Local Bus Arbitration Latency Count Register – LBALCR



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 80960VH 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 80960VH 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 Internal Arbitration Units

The 80960VH contains an internal arbitration unit that controls access to the internal PCI buses within the device. The Primary Internal PCI Arbitration Unit arbitrates for the following internal units:

- Primary ATU
- DMA Channel 0
- DMA Channel 1

The internal PCI arbitration unit uses a fixed round-robin arbitration scheme with each device on a bus having equal priority.

18.3.1 Internal Master Latency Timer

The PCI interface of the 80960VH contains a Master Latency Timer (MLT) for use by the internal resources when they are acting as PCI bus masters. The ATU and the DMA channels 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.

The 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). The bus master must relinquish the bus when an external device or one of the internal resources requests the bus.

Timers 19

This chapter describes the i960® VH processor's 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

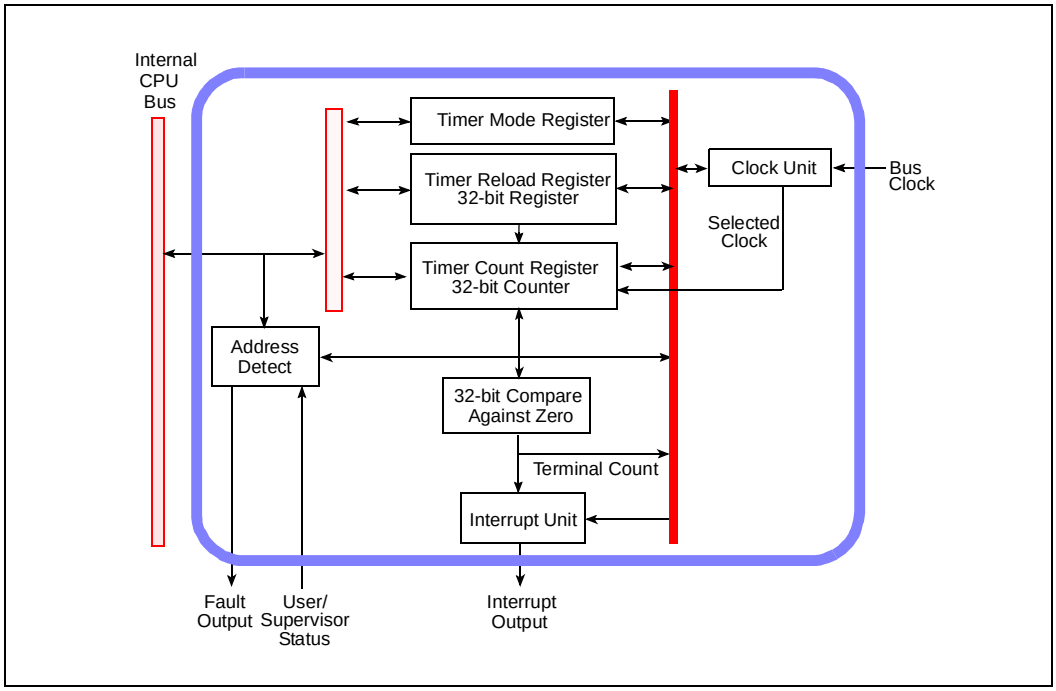


Table 19-1. Timer Performance Ranges

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 Register – TMR0:1” on page 19-2.
- Timer Count Register - contains the timer’s current count. See Section 19.1.2, “Timer Count Register – TCR0:1” on page 19-5.
- Timer Reload Register - contains the timer’s reload count. See Section 19.1.3, “Timer Reload Register – TRR0:1” on page 19-6.

Table 19-2. Timer Registers

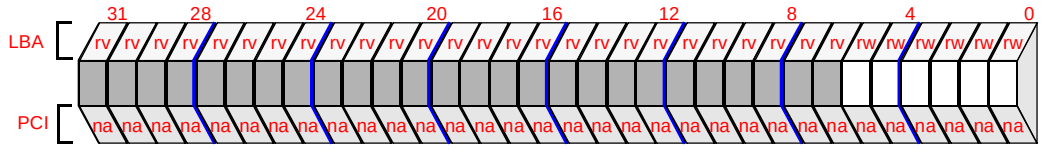
Timer Unit	Register Acronym	Register Name
Timer 0	TMR0	Timer Mode Register 0
	TCR0	Timer Count Register 0
	TRR0	Timer Reload Register 0
Timer 1	TMR1	Timer Mode Register 1
	TCR1	Timer Count Register 1
	TRR1	Timer Reload Register 1

For register memory locations, see Table C-3 “Timer Registers” on page C-3.

19.1.1 Timer Mode Register – TMR0:1

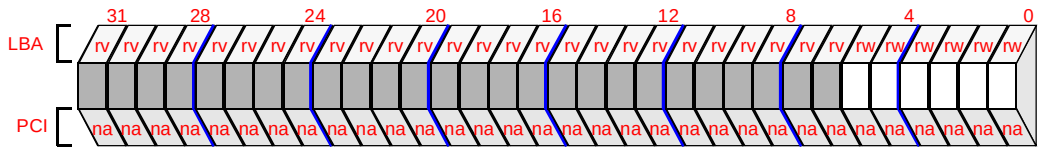
The Timer Mode Register (TMRx) lets the user program the mode of operation and determine the current status of the timer. TMRx bits are described in the subsections following Table 19-3 and are summarized in Table 19-7.

Table 19-3. Timer Mode Register – TMRx (Sheet 1 of 2)



LBA: CH 0-0308H CH 1-0318H **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**
31:06	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

Table 19-3. Timer Mode Register – TMRx (Sheet 2 of 2)



LBA: CH 0-0308H CH 1-0318H **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	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 12.2, “i960® VH Processor Initialization” on page 12-2.

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. For example, the timer runs while the processor is in Halt mode. 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 12.2, “i960® VH Processor Initialization” on page 12-2.

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. For example, the timer runs while the processor is in Halt mode. 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 12.2, “i960® VH Processor Initialization” on page 12-2.

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

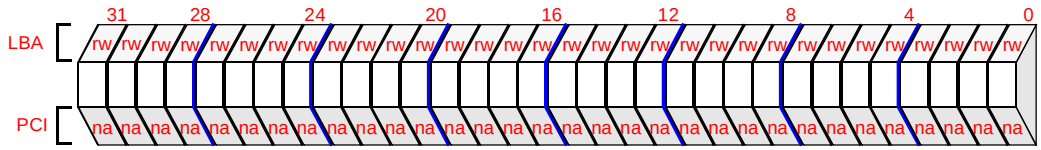
Bit 5 TMRx.csel1	Bit 4 TMRx.csel0	Timer Clock (TCLOCK)
0	0	Timer Clock = Bus Clock
0	1	Timer Clock = Bus Clock / 2
1	0	Timer Clock = Bus Clock / 4
1	1	Timer Clock = Bus Clock / 8

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
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 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" on page 19-9.

User software can read or write TCRx whether the timer is running or stopped. Bit 3 of TMRx determines user read/write control (Section 19.1.1.4). The TCRx value is undefined after hardware or software reset.

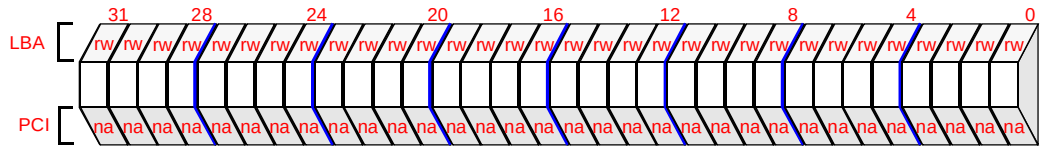
19.1.3 Timer Reload Register – TRR0:1

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" on page 19-9 for more information.)

User software can access TRRx whether the timer is running or stopped. Bit 3 of TMRx determines read/write control (Section 19.1.1.4, "Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup)" on page 19-4). TRRx value is undefined after hardware or software reset.

Table 19-6. Timer Reload Register – TRRx



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" on page 19-8 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.

NOTE: 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

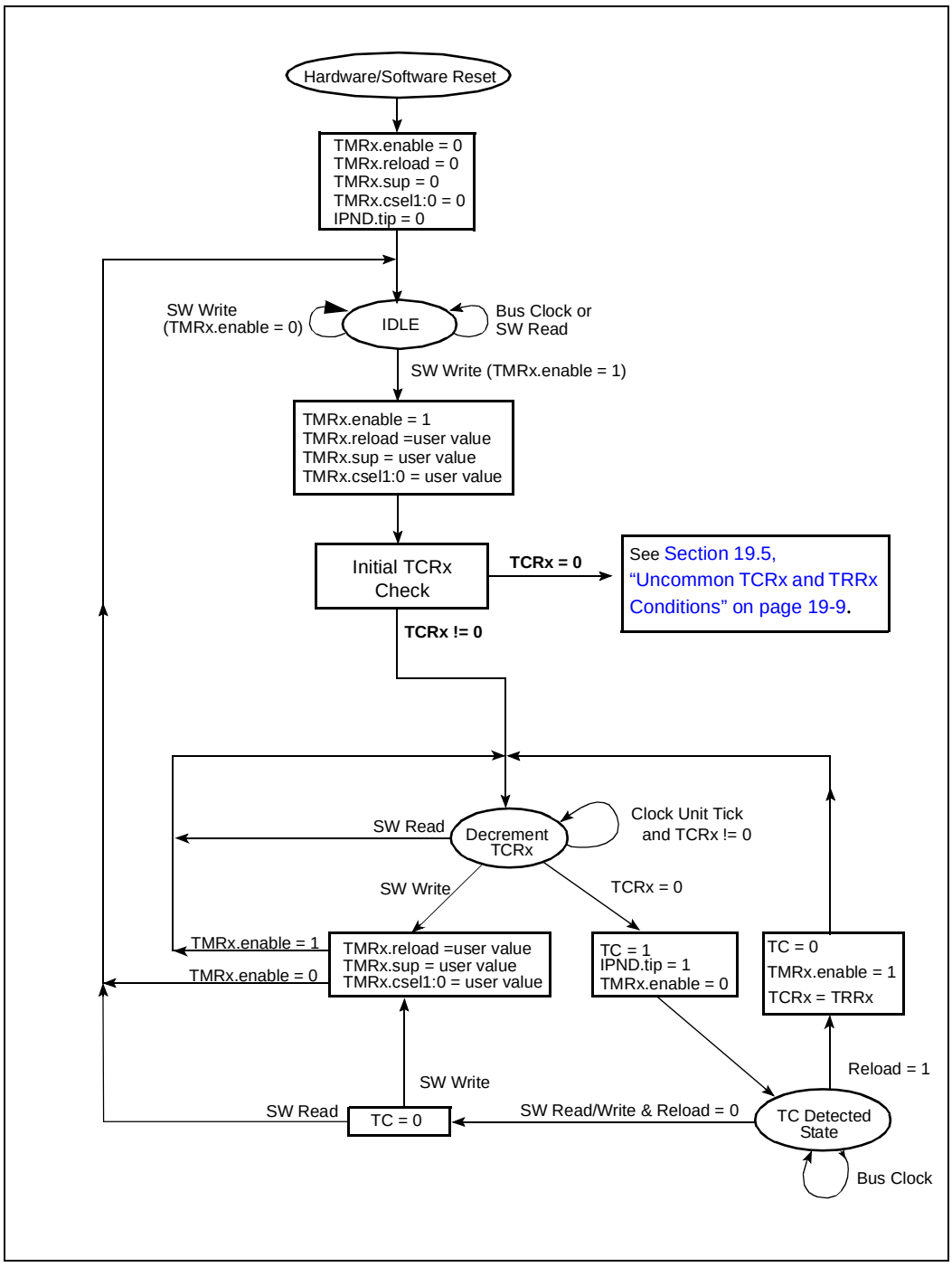
TRRx	TCRx	Bit 2 (TMRx.reload)	Bit 1 (TMRx.enable)	Action
X	0	0	1	TMRx.tc and TINTx set, TMR.enable cleared
0	0	1	1	Timer and auto reload enabled, TINTx not generated and timer enable remains set.
0	N	1	1	Timer and auto reload enabled. TINT.x set when TCRx=0. The timer remains enabled but further TINTx's 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” on page 19-9.

Figure 19-2. Timer Unit State Diagram



DMA Controller 20

This chapter describes the integrated Direct Memory Access (DMA) Controller, including the operation modes, setup, external interface, registers and interrupts.

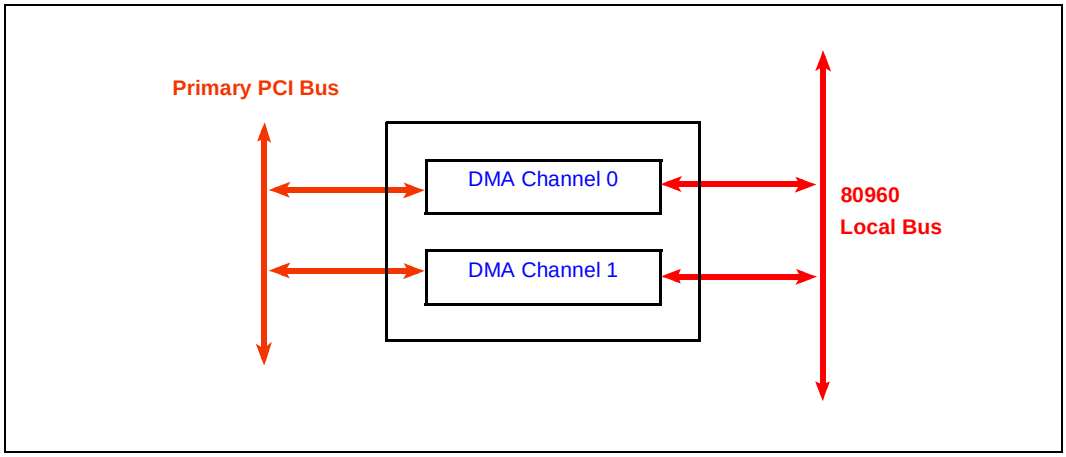
20.1 Overview

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:

- Two Independent Channels
- Memory Controller Interface
- 32-bit addressing range on the 80960 local bus
- 32-bit addressing range on the primary PCI interface
- Independent PCI interfaces to the primary PCI bus.
- 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

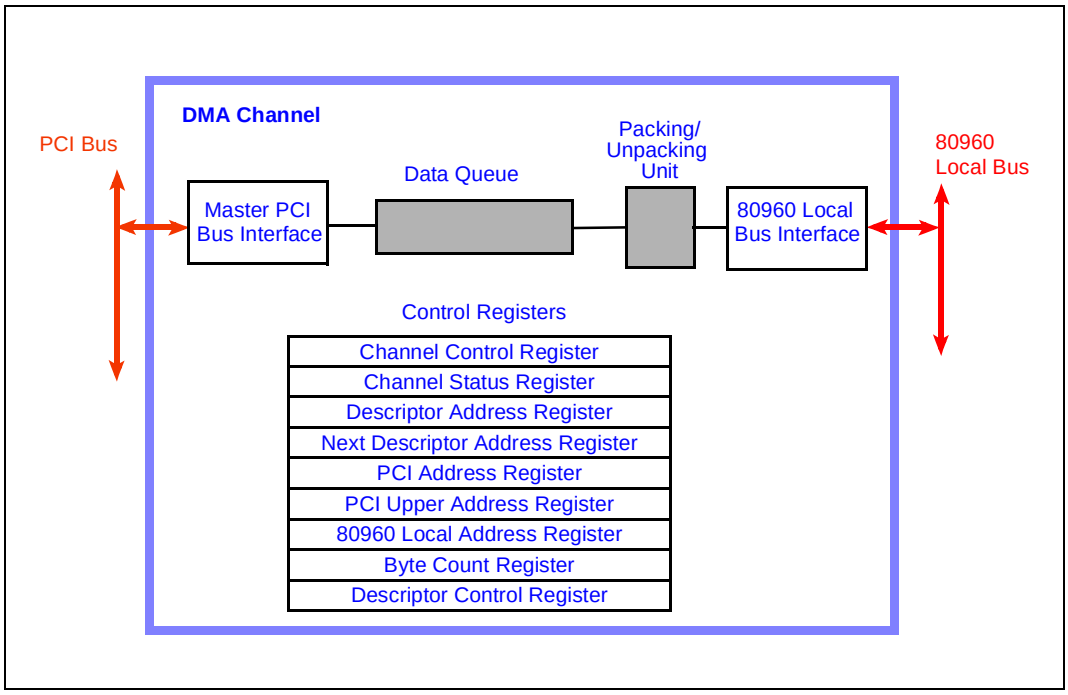
The DMA Controller provides two channels of high throughput PCI-to-memory transfers:

- Channels 0 and 1 transfer data blocks between the primary PCI bus and 80960 local memory. Channel 0 also supports demand-mode transfers.

Channel 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. 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” on page 20-3.

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” on page 20-20. 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

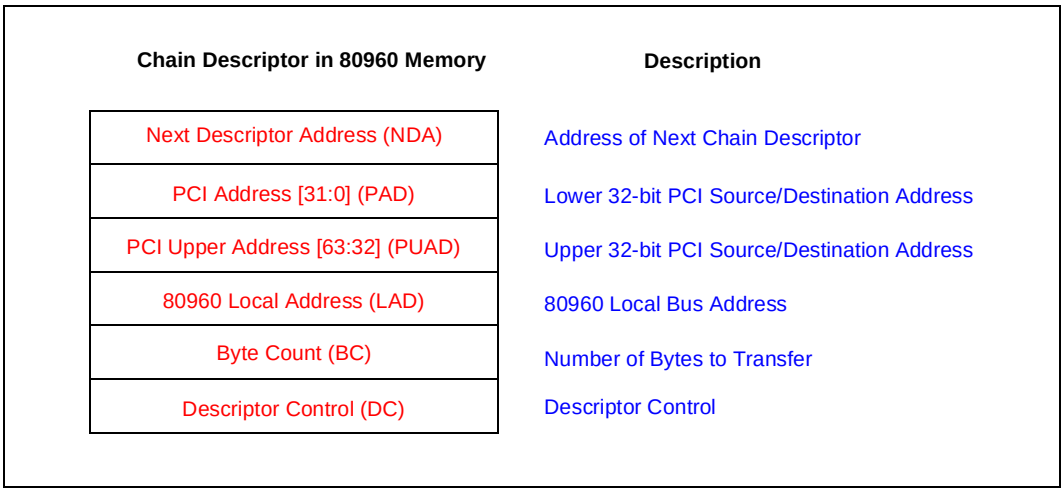
Register	Abbreviation	Description
Channel Control Register	CCR	Channel Control Word
Channel Status Register	CSR	Channel Status Word
Descriptor Address Register	DAR	Address of Current Chain Descriptor
Next Descriptor Address Register	NDAR	Address of Next Chain Descriptor
PCI Address Register	PADR	Lower 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.

Figure 20-3. DMA Chain Descriptor



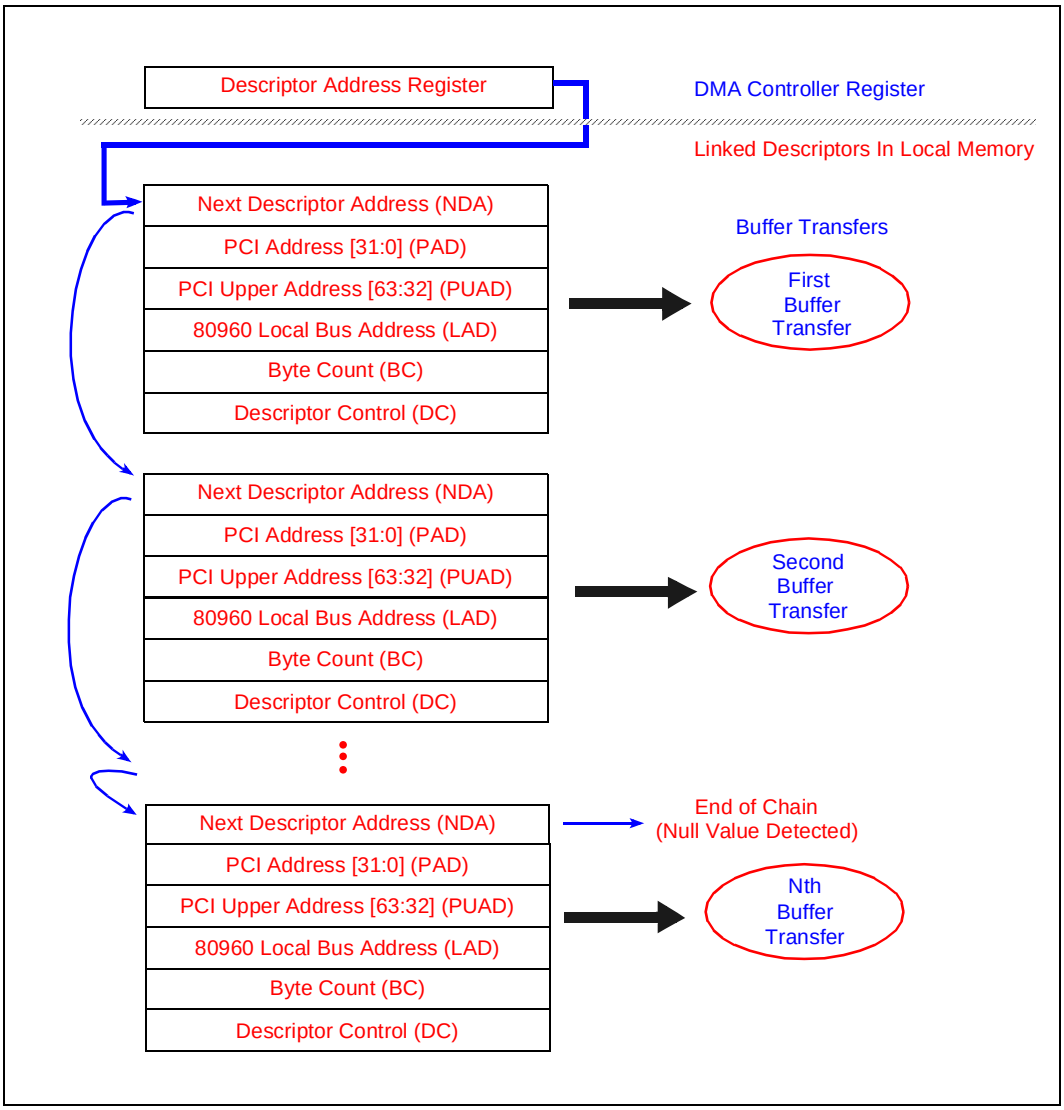
Each chain descriptor word is analogous to control register values. Bit definitions for chain descriptor words are the same as for the DMA control registers.

- 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 ignores 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. Dual Address Cycles are not supported on the 80960VH processor, therefore this field must be set to zero (0).
- 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” on page 20-30.

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.

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.

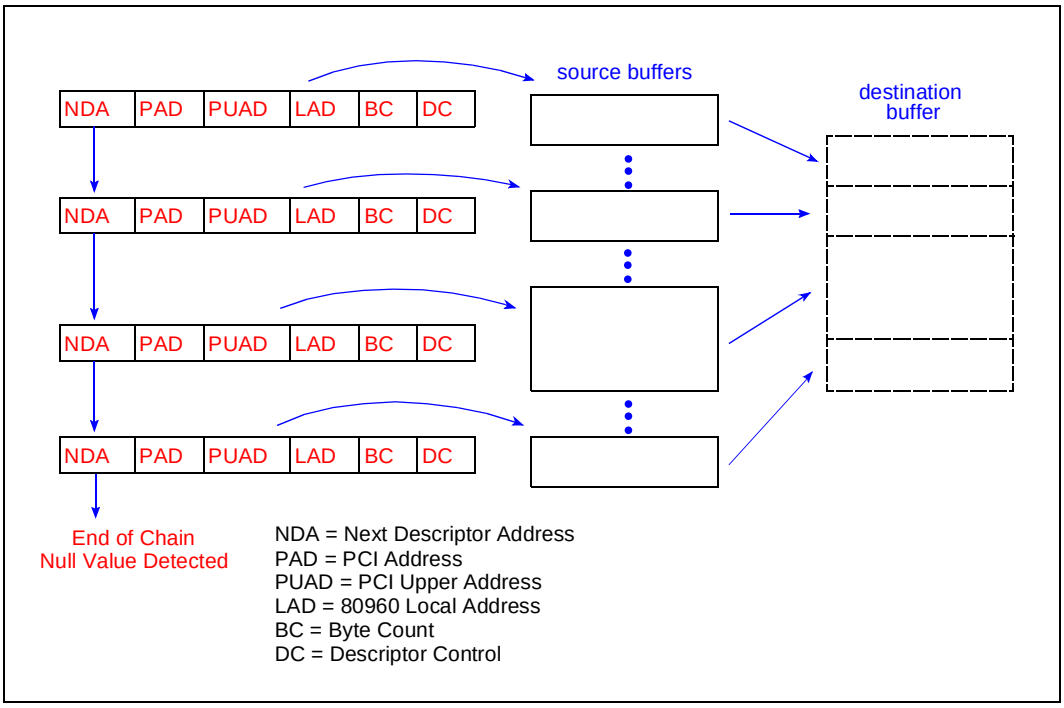
Once a DMA transfer is active, it may be temporarily suspended by clearing the CCR Channel Enable bit. Note that this does not abort the DMA transfer; the channel resumes the DMA transfer when the Channel Enable bit is set.

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, so 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

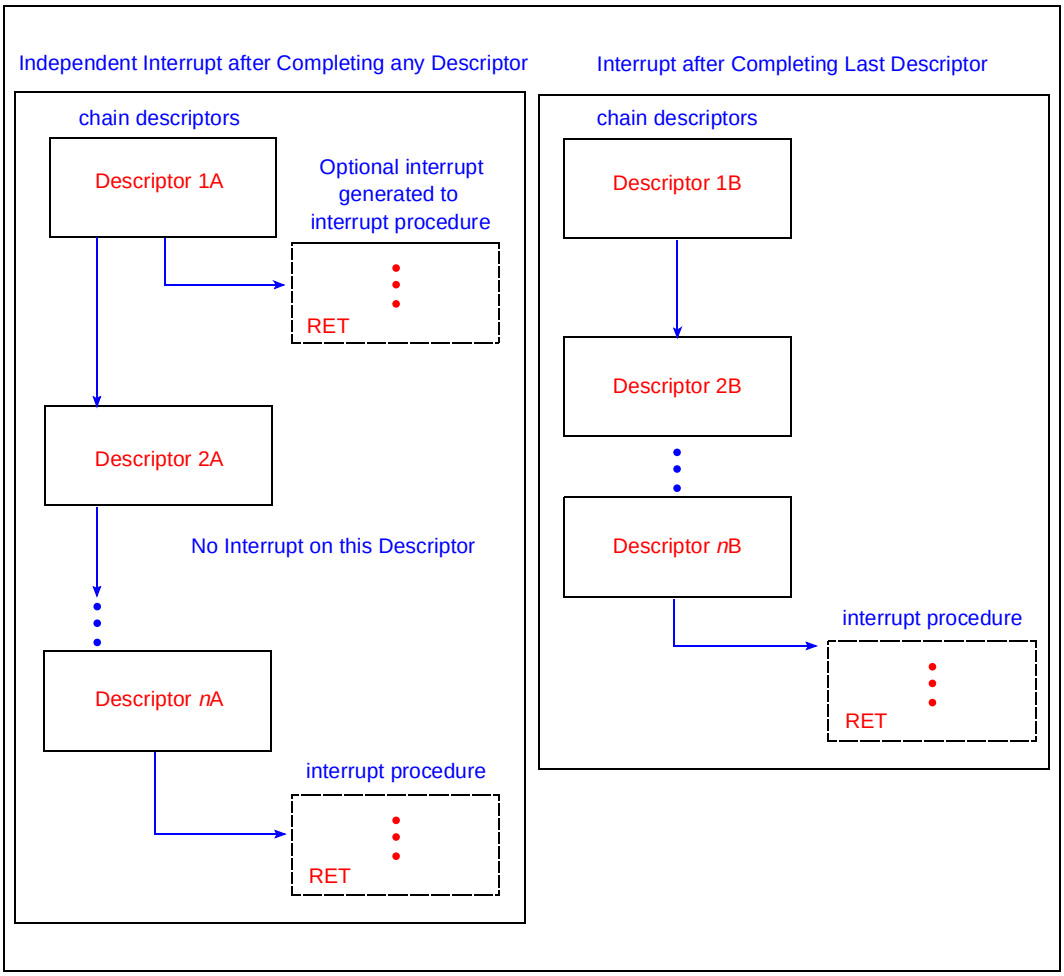
Chained DMA transfers can be synchronized to a program executing on the i960 core processor through the use of processor interrupts. The channel generates an interrupt to the i960 core processor under certain conditions. They are:

- 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.

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, "Descriptor Control Register - DCRx" on page 20-28) 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.

Figure 20-7. DMA - Aligned Write to Device, Wait States, Device Always Requesting

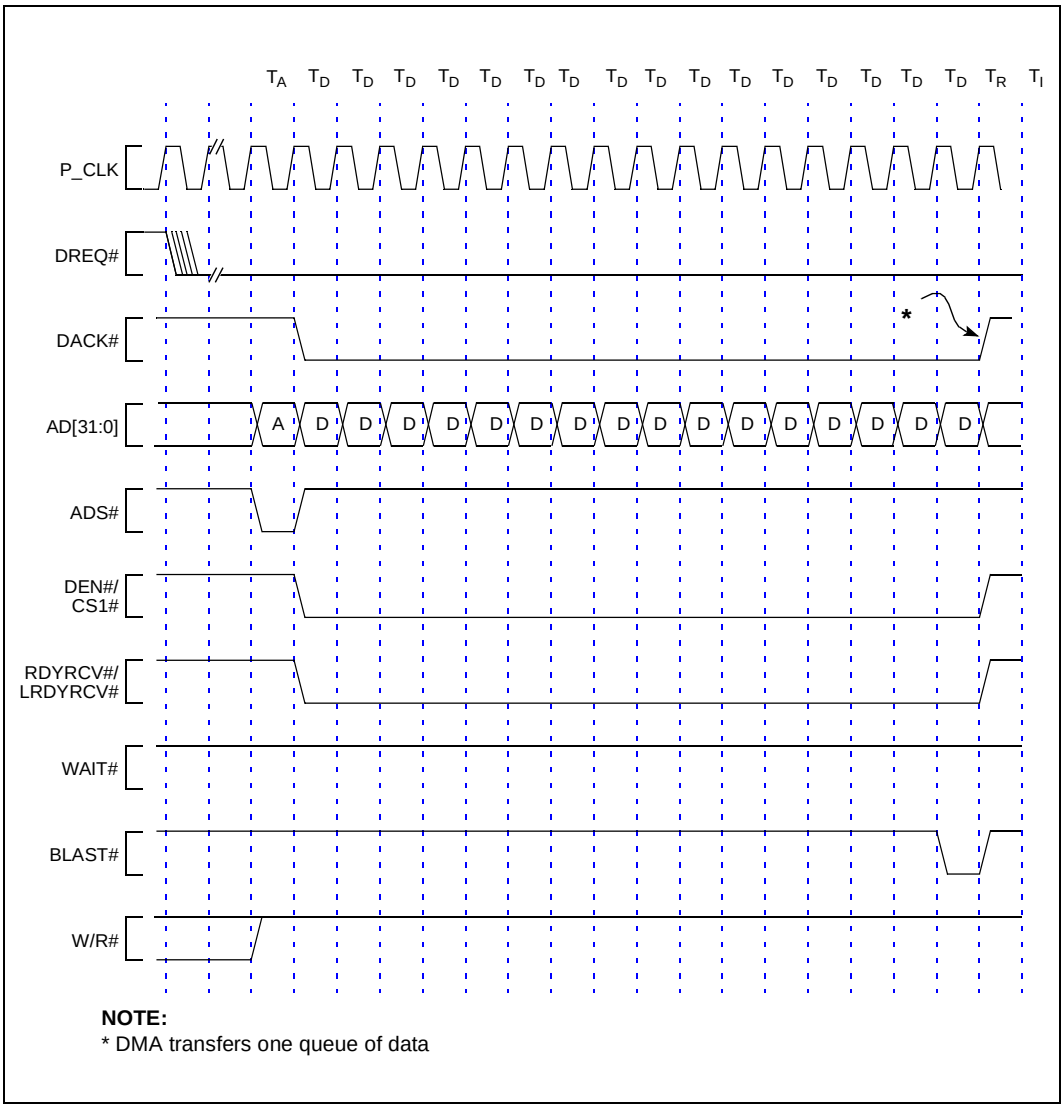
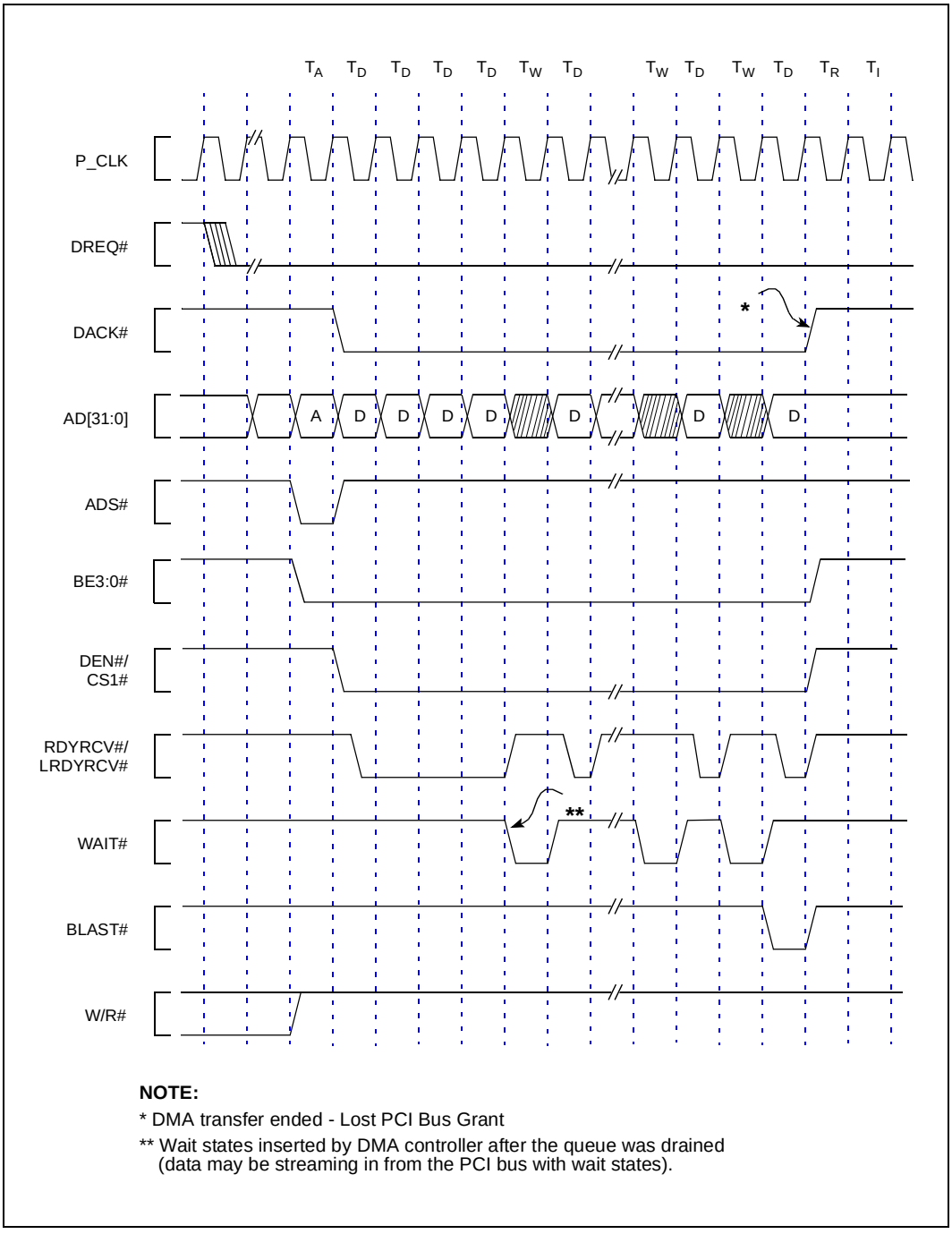


Figure 20-8. DMA - Aligned Write to Device, DMA Inserting Wait States, Device Always Requesting

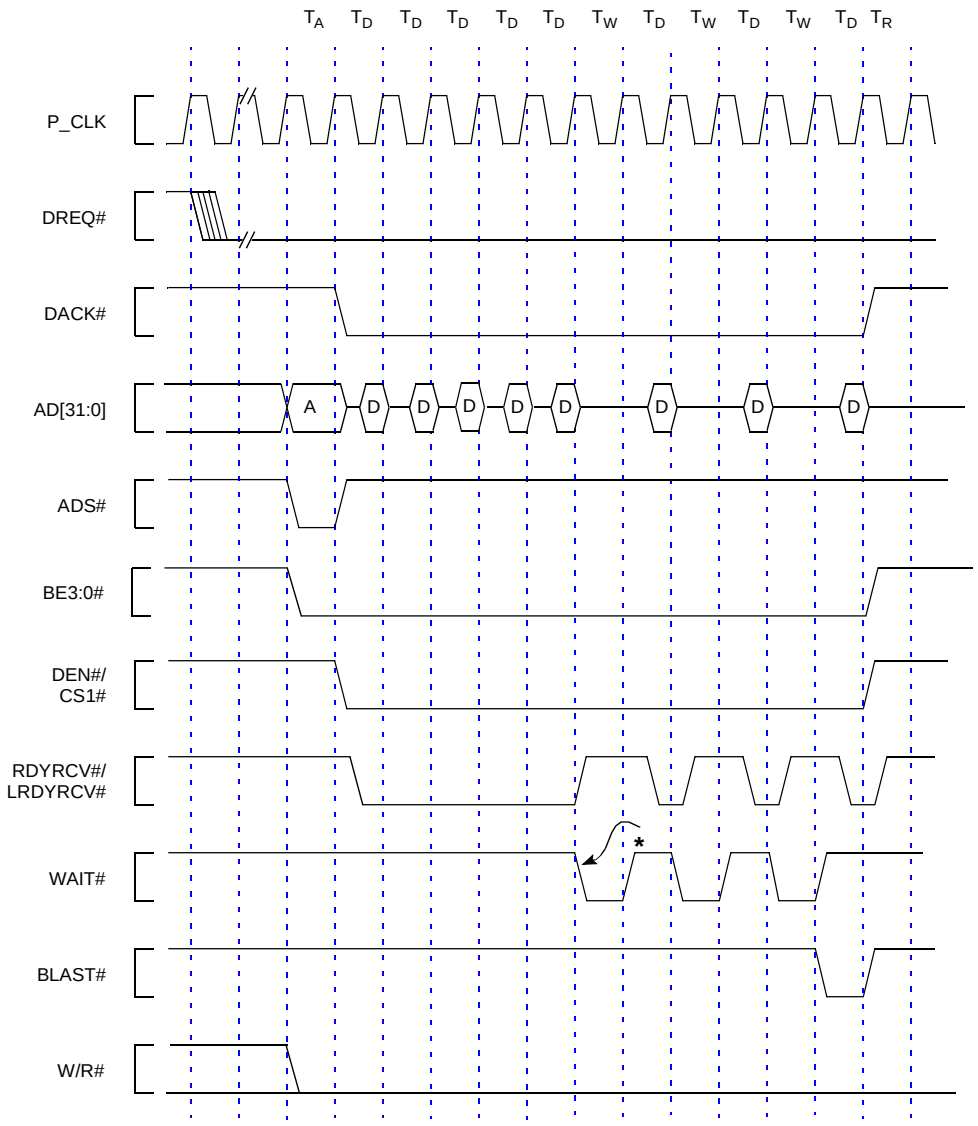


NOTE:

* 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).

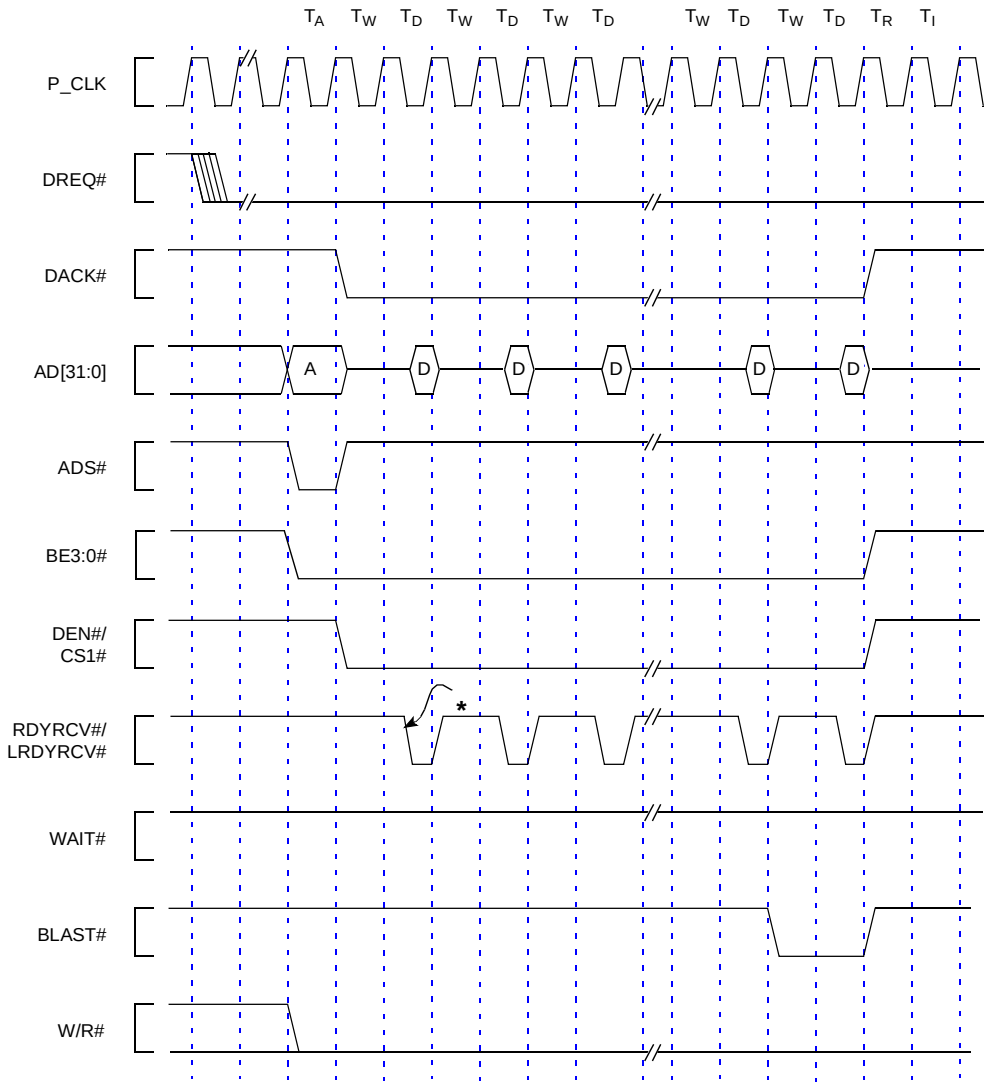
Figure 20-9. DMA - Aligned Read from Device, DMA Inserting Wait States, Device Always Requesting



NOTE:

* Wait states inserted by the DMA controller after the queue filled (data may be streaming on the PCI bus with wait states).

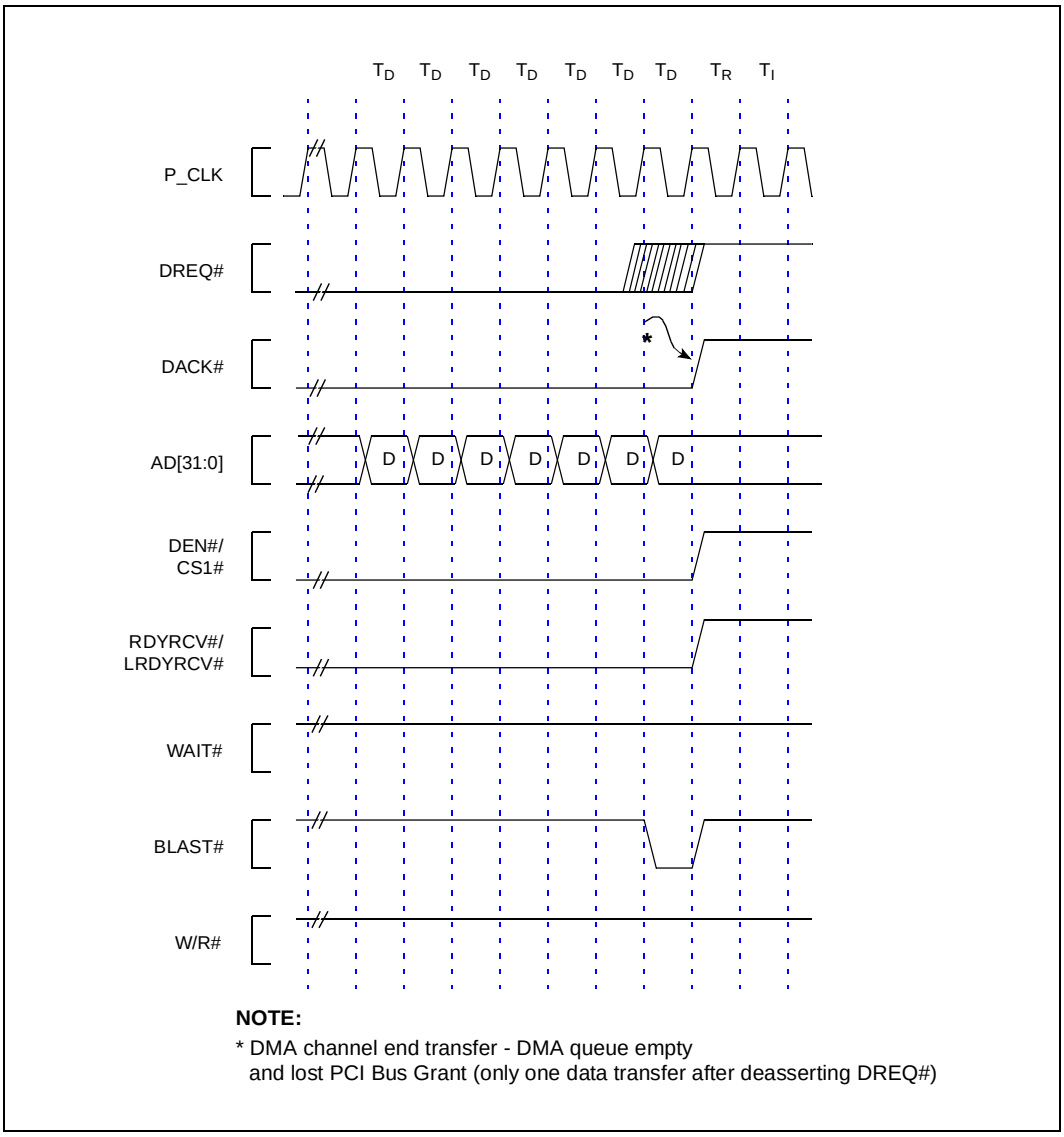
Figure 20-10. DMA - Aligned Read from Device, Device Inserting Wait States, Device Always Requesting



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#

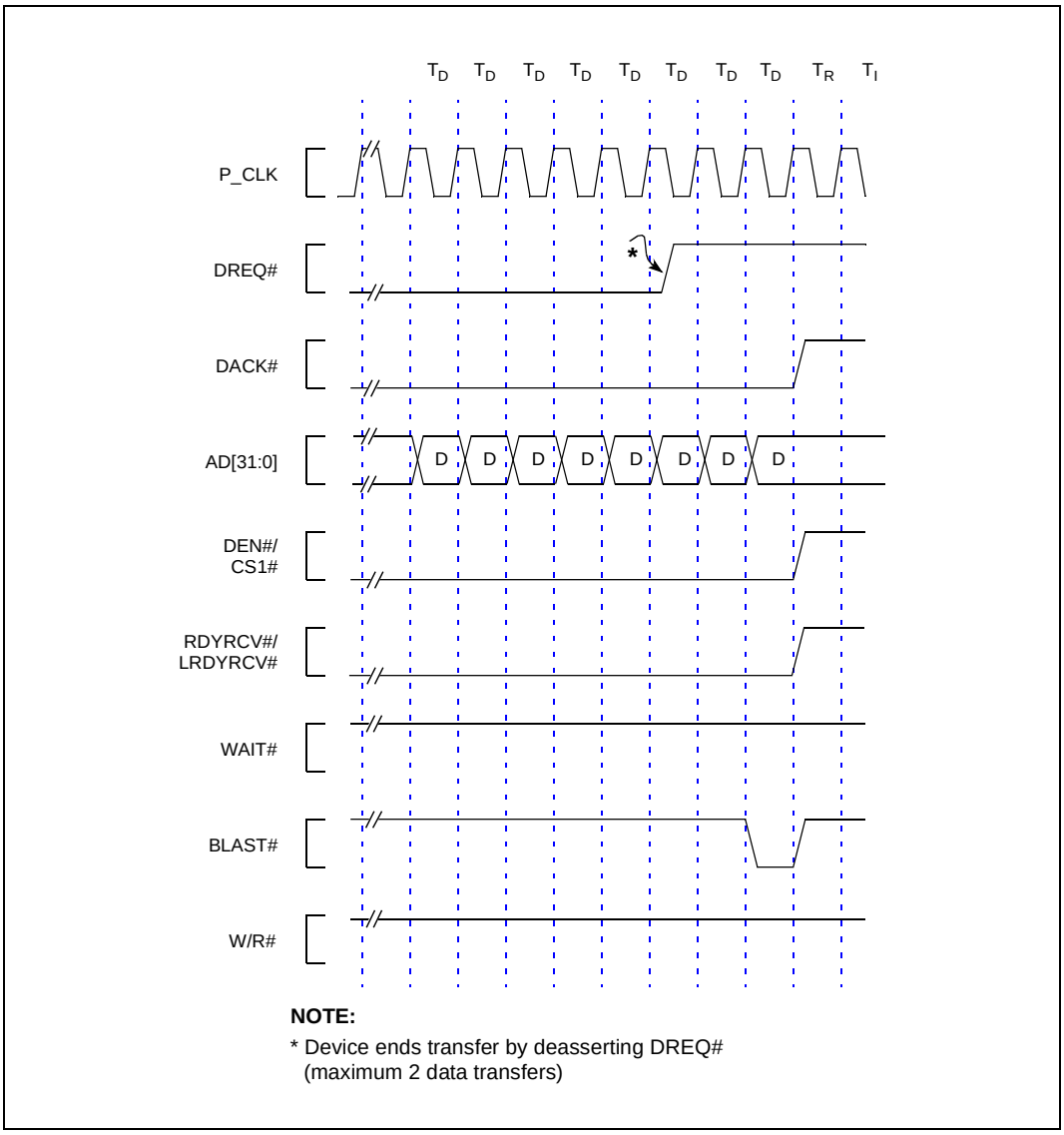
Figure 20-11. DMA - Aligned Write to Device, Zero Wait States, Device ends Transfer



NOTE:

* DMA channel end transfer - DMA queue empty and lost PCI Bus Grant (only one data transfer after deasserting DREQ#)

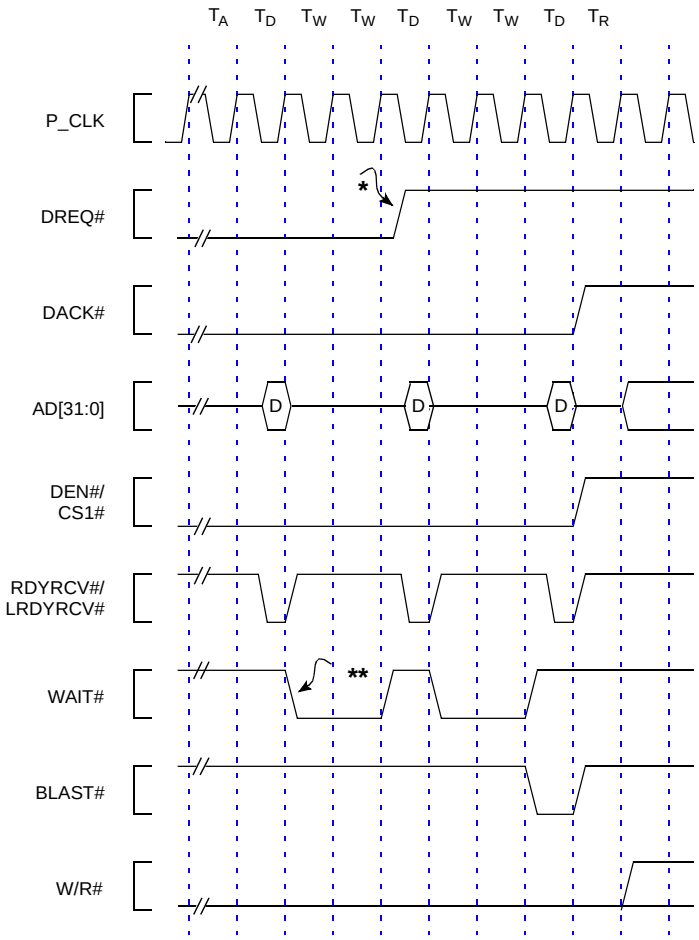
Figure 20-12. DMA - Aligned Write to Device, Zero Wait States, Device ends Transfer



NOTE:

* Device ends transfer by deasserting DREQ# (maximum 2 data transfers)

Figure 20-13. DMA - READ from Device, Wait States, Device ends Transfer

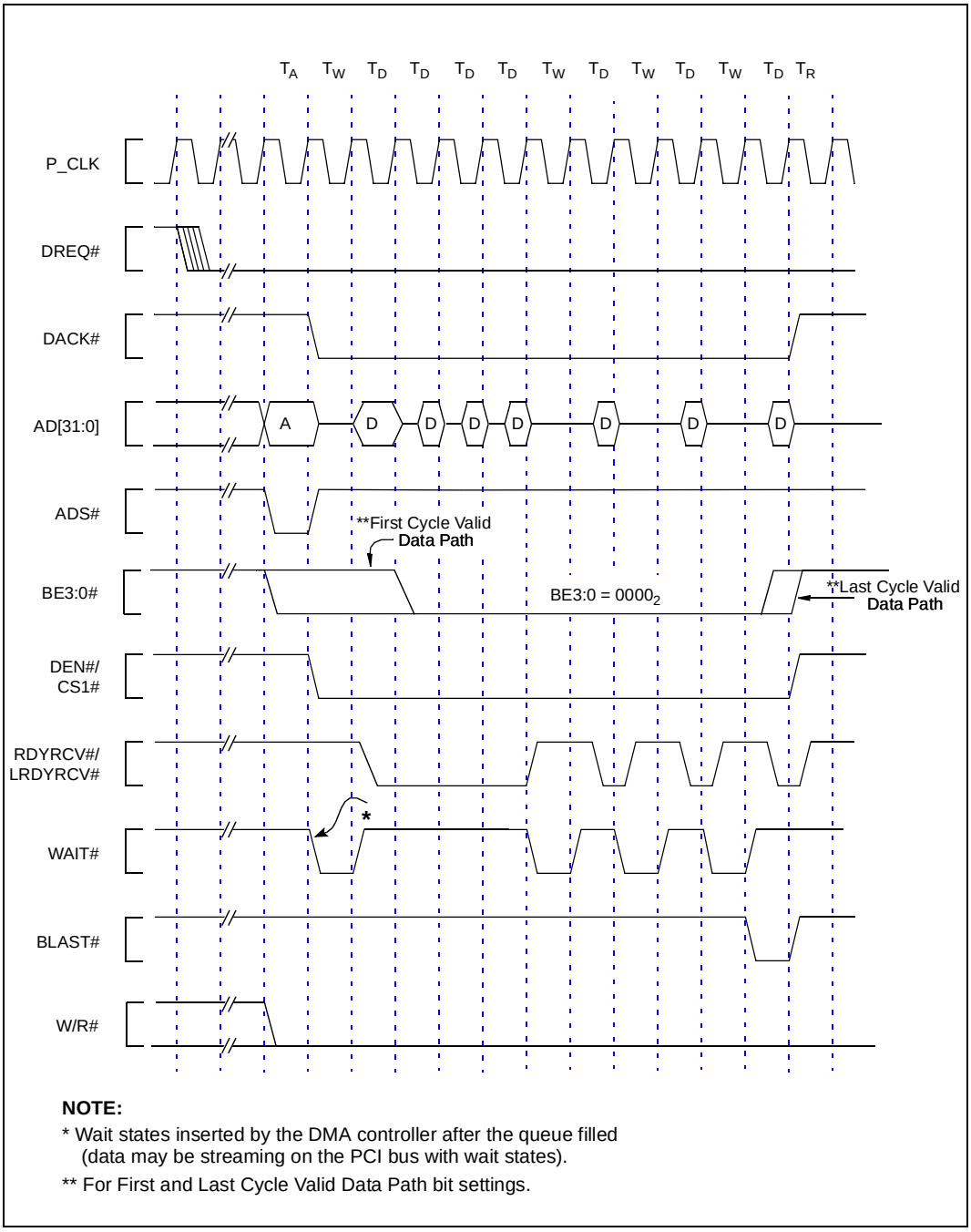


NOTE:

* 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 20-14. DMA - Unaligned Read from 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).

** For First and Last Cycle Valid Data Path bit settings.

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, then 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, then 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:
   a. For Channels 0 and 1: Primary 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.

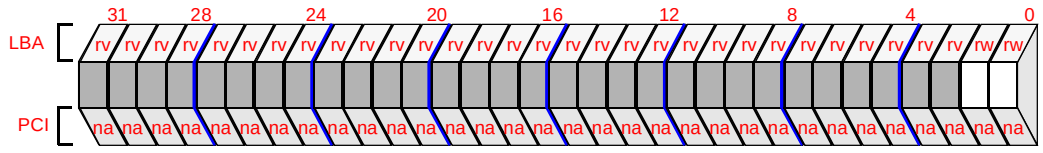
Table 20-2. DMA Controller Register Summary

Section	Register Name - Acronym	Page	Size (Bits)	DMA Channel	80960 Local Bus Address	PCI Config Addr Offset
20.7.1	Channel Control Register - CCRx	20-21	32	0 1	0000 1400H 0000 1440H	NA
20.7.2	Channel Status Register - CSRx	20-22	32	0 1	0000 1404H 0000 1444H	NA
20.7.3	Descriptor Address Register - DARx	20-24	32	0 1	0000 140CH 0000 144CH	NA
20.7.4	Next Descriptor Address Register - NDARx	20-24	32	0 1	0000 1410H 0000 1450H	NA
20.7.5	PCI Address Register - PADRx	20-25	32	0 1	0000 1414H 0000 1454H	NA
20.7.6	PCI Upper Address Register - PUADRx	20-26	32	0 1	0000 1418H 0000 1458H	NA
20.7.7	80960 Local Address Register - LADRx	20-26	32	0 1	0000 141CH 0000 145CH	NA
20.7.8	Byte Count Register - BCRx	20-27	32	0 1	0000 1420H 0000 1460H	NA
20.7.9	Descriptor Control Register - DCRx	20-28	32	0 1	0000 1424H 0000 1464H	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.

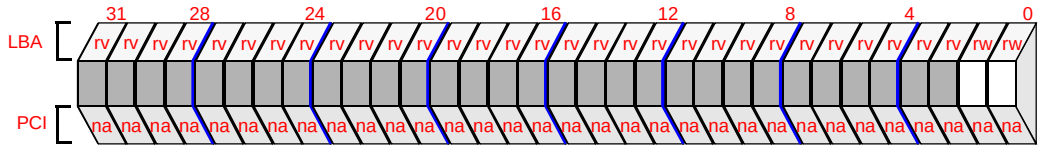
Table 20-3. Channel Control Register - CCRx (Sheet 1 of 2)



LBA: CH.0-1400H CH.1-1440H **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.

Table 20-3. Channel Control Register - CCRx (Sheet 2 of 2)



LBA: CH.0-1400H CH.1-1440H **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

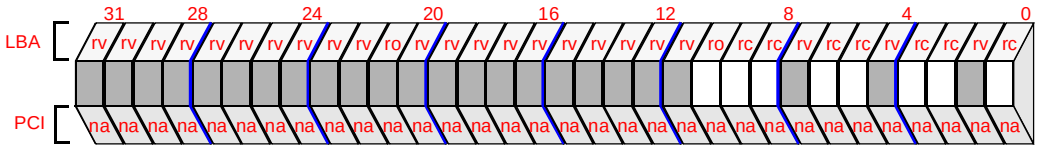
Bit	Default	Description
01	0_2	Chain Resume - when set, causes the channel to resume chaining by re-reading the current descriptor located at the address in the Descriptor Address Register when the channel is idle (Channel Active bit in the CSR is clear) or when the channel completes a DMA transfer. 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. 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” on page 20-29 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.

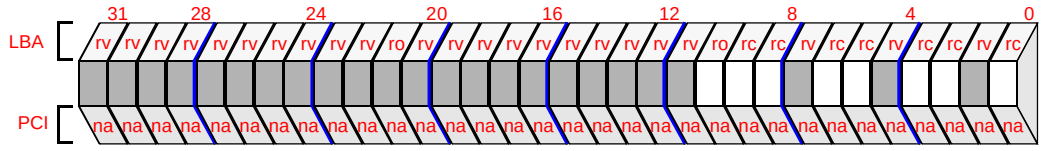
Table 20-4. Channel Status Register - CSRx (Sheet 1 of 2)



LBA: CH.0-1404H CH.1-1444H **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	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.
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.

Table 20-4. Channel Status Register - CSRx (Sheet 2 of 2)



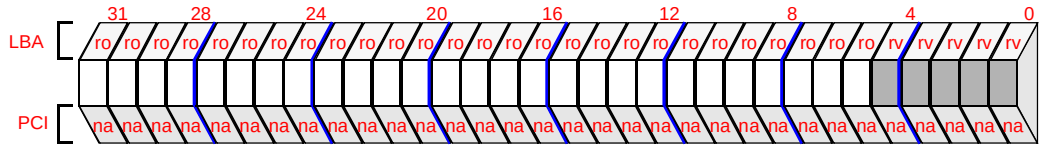
LBA: CH.0-1404H CH.1-1444H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
00	0_2	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: CH.0-140CH CH.1-144CH **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	Current Descriptor Address - 80960 local bus memory address of the current chain descriptor that was read by the channel.
04:00	00H	Reserved.

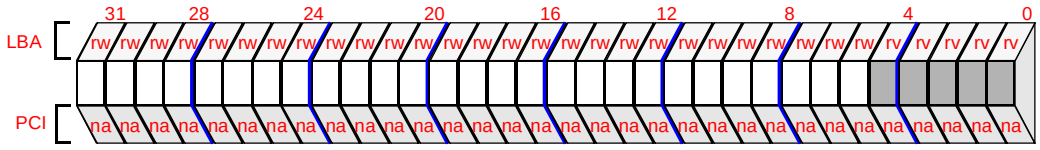
20.7.4 Next Descriptor Address Register - NDARx

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



LBA: CH.0-1410H CH.1-1450H **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.

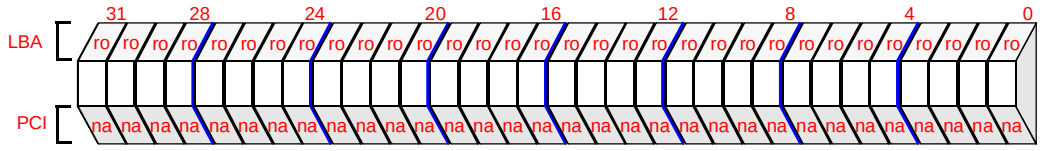
20.7.5 PCI Address Register - PADRx

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. DAC is not supported on the i960 VH processor. 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 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



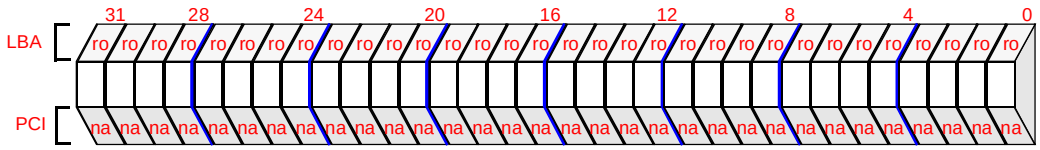
LBA: CH.0-1414H CH.1-1454H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset

Bit	Default	Description
31:00	0000 0000H	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. Dual Address Cycle is not supported on the i960 VH processor. The PCI Upper Address [63:32] (PUAD) in the DMA Chain Descriptor must always be set to zero (0).

Table 20-8. PCI Upper Address Register - PUADRx



LBA: CH.0-1418H CH.1-1458H **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

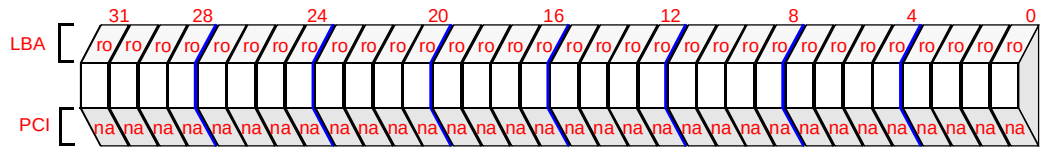
Bit	Default	Description
31:00	0000 0000H	PCI Upper Address - is the PCI source/destination upper address. Not implemented and a reserved register. The PCI upper address [63:32] (PUAD) in the DMA Chain Descriptor must always be set to zero (0). Refer to bit 5 of the Descriptor Control Register - DCRx in Section 20.7.9.

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



LBA: CH.0-141CH CH.1-145CH **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	80960 local bus address - the 80960 local bus source/destination address.

20.7.8 Byte Count Register - BCRx

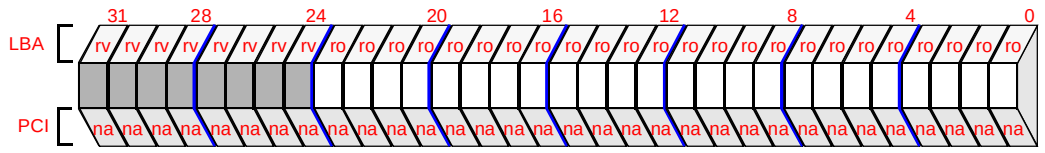
The Byte Count Register contains the number of bytes to transfer for a DMA transfer. This is a read-only register that is loaded from the Byte Count word in a chain descriptor. It allows for a maximum DMA transfer of 16 Mbytes. A value of zero is a valid byte count and results in no data words being 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



LBA: CH.0-1420H CH.1-1460H **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

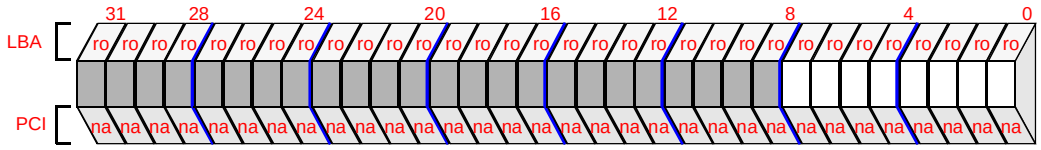
Bit	Default	Description
31:24	00H	Reserved
23:00	00 0000H	Byte Count - is the number of bytes to transfer for a DMA transfer.

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: CH.0-1424H CH.1-1464H **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	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, “Demand Mode DMA”) 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. Not implemented and a reserved bit field. This bit must always be set to zero (0) to disable the Dual Address Cycle.
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; for example, 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 (Sheet 1 of 2)

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

Table 20-12. PCI Commands (Sheet 2 of 2)

C/BE3:0#	PCI Command Type	Description
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 (Sheet 1 of 2)

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

Table 20-13. DMA Interrupt Summary (Sheet 2 of 2)

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
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.

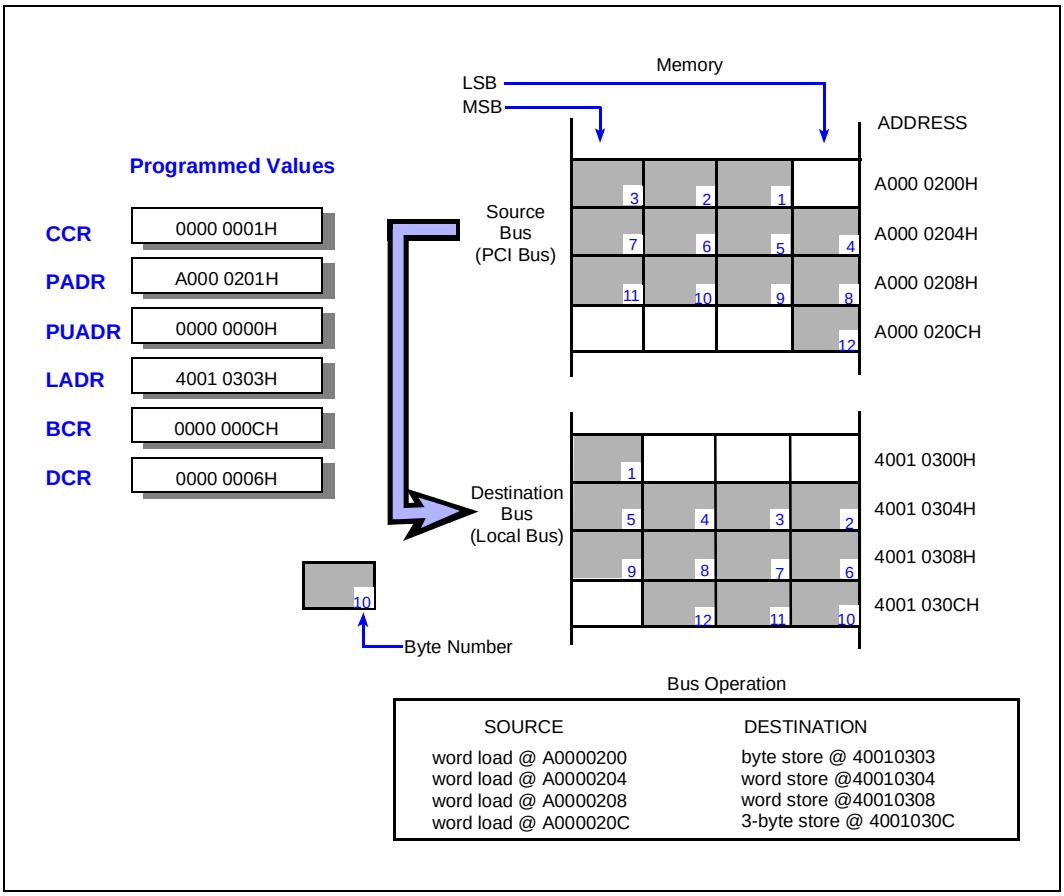
When a Memory Fault Error or Bus Fault Error occurs while reading the Chain Descriptor or Next Descriptor Address, the channel sets the appropriate CSR error flag, loads the CCRs (if possible), and stops.

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.

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.

Figure 20-16. Software Example for Channel Initialization

```
CCR0 = 0x0000 0000 ; Disable channel
Call setup_channel
```

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.

Figure 20-17. Software Example for Channel Suspend

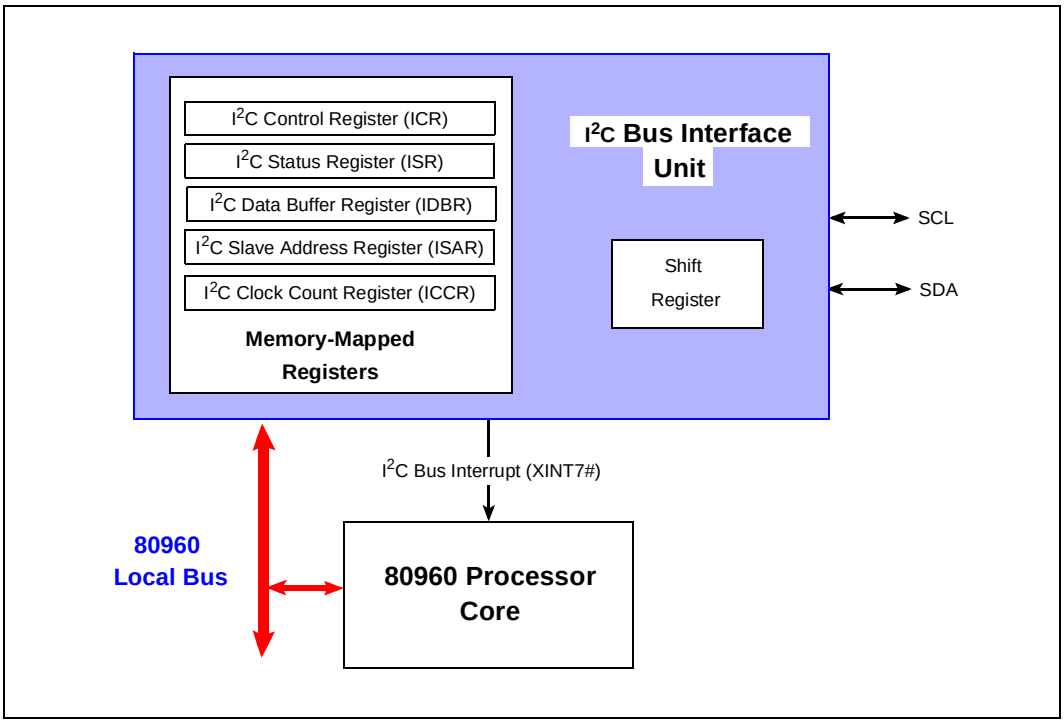
```
CCR0 = 0x0000 0000; Suspend Channel 0

     Channel suspended.....

CCR0 = 0x0000 0001; Resume Channel 0
```

I²C Bus Interface Unit 21

This chapter describes the I²C (Inter-Integrated Circuit) bus interface unit of the i960® VH processor, including the operation modes and setup. Throughout this manual, this peripheral is referred to as the I²C unit.

Figure 21-1 shows a block diagram of the I²C unit and its interface to the 80960 local bus.

Figure 21-1. I²C Unit Block Diagram



21.1 Overview

The I²C bus allows the 80960VH to interface to other I²C 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 80960VH subsystem to an external device.

Data transfers to and from the I²C bus via a buffered interface. Control and status information are relayed through a set of 80960 memory-mapped registers. An interrupt mechanism notifies the 80960VH of I²C activity. Refer to any of the following sources for details on I²C bus operation:

- I²C *Peripheral for Microcontrollers* – Philips Semiconductor
- I²C *Bus and How to Use It (Including Specifications)* – Philips Semiconductor
- I²C *Peripherals for Microcontrollers (Including Fast Mode)* – Signetics

The I²C unit allows the i960 VH processor to serve as a master or slave device residing on the I²C bus. The I²C 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 I²C bus
- An 8-bit buffer for passing data to and from the 80960VH
- A shift register for parallel/serial data conversions
- A set of control and status registers
- A dedicated interrupt to inform the 80960VH of activity on the I²C bus

21.2 Theory of Operation

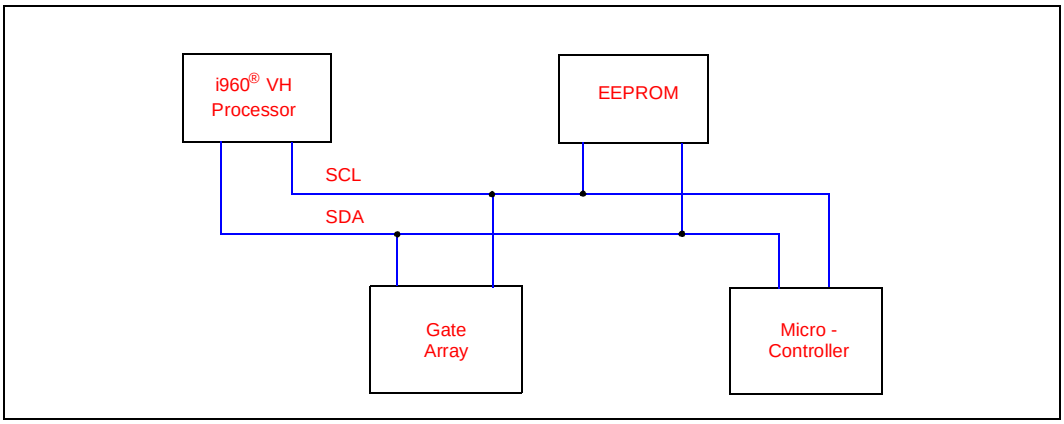
The I²C bus defines a complete serial protocol for passing information between agents on the I²C bus using only a two pin interface. The interface consists of a Serial Data/Address (SDA) line and a Serial Clock Line (SCL). Each device on the I²C bus is recognized by a unique 7-bit address and can operate as a transmitter or as a receiver. In addition to transmitter and receiver, the I²C bus uses the concept of master and slave. Table 21-1 defines terms used in this chapter.

Table 21-1. I²C Bus Definitions

I²C Device	Definition
Transmitter	Sends data to the I²C bus.
Receiver	Receives data from the I²C 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	A 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 I²C bus operation, consider the case of an 80960VH acting as a master on the bus (see Figure 21-2). The 80960VH, as a master, addresses an EEPROM as a slave to receive data. The 80960VH is a master-transmitter and the EEPROM is a slave-receiver. When the 80960VH reads data, the 80960VH 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^2C Bus Configuration Example

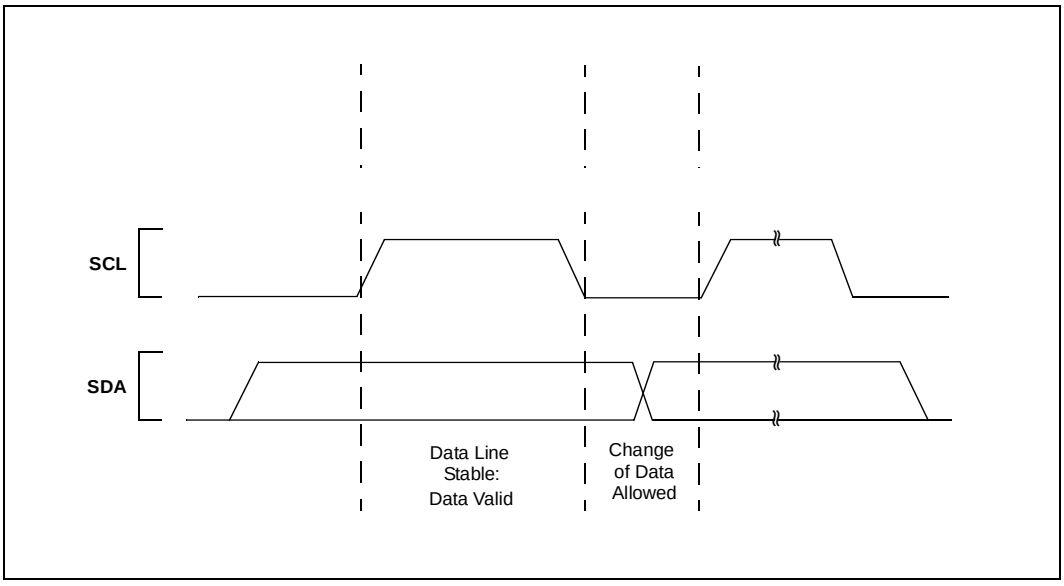


The I^2C bus allows for a multi-master system, which means more than one device can initiate data transfers at the same time. To support this feature, the I^2C bus arbitration relies on the wired-AND connection of all I^2C interfaces to the I^2C bus. Two masters can drive the bus simultaneously provided 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^2C 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, etc. 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" on page 21-7.

The I^2C 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^2C *Peripheral for Microcontrollers* by Philips Semiconductor for details. I^2C unit does not support I^2C 10-bit addressing or CBUS.

Figure 21-3. Bit Transfer on the I2C Bus



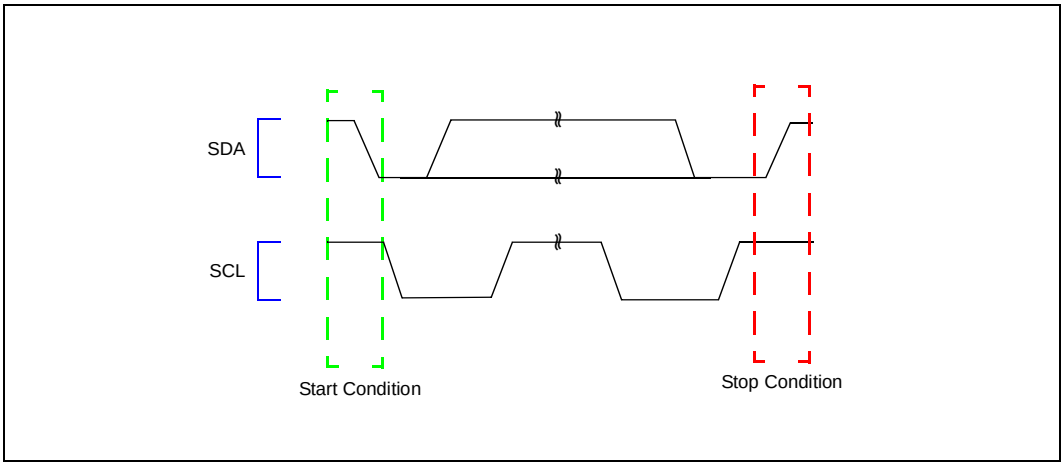
21.3 Start and Stop Bus States

The 80960VH uses the START and STOP bits (bits 1:0) in the ICR (Table 21-6) to:

- Initiate a START condition on the I2C bus.
- Enable data chaining (repeated START).
- Initiate a STOP condition on the I2C 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 “I2C Data Buffer Register – IDBR” on page 21-21) before setting the START ICR bit (see Figure 21-4). The START and the IDBR contents are transmitted on the I^2C bus when the ICR transfer byte bit is set. The I^2C 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^2C unit when arbitration is lost. While initiating a START and the arbitration is lost, the I^2C unit may re-attempt the START when the bus becomes free - see Section 21.6.2, “SDA Arbitration” on page 21-8. See Section 21.6, “Arbitration” on page 21-7 for details on how the I^2C 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 80960VH 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^2C 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^2C unit after the transmission of a START, STOP or repeated START.

After each byte transfer (including the Ack/Nack bit) the I^2C unit holds the SCL line low (inserting wait states) until the transfer byte bit in the ICR is set. This action notifies the I^2C 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 80960VH 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 80960VH’s I^2C clock (SCL) is programmed via the I^2C Clock Count Register (ICCR). The following subsections describe how the SCL works and is programmed.

21.4.1 SCL Clock Generation

The 80960VH's I²C unit is required to generate the I²C 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 * 80960VH 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 I²C unit performs the necessary clock synchronization. The ICCR provides a simple method for determining I²C clock frequencies. Table 21-2 details sample programming values for the ICCR.

Table 21-2. ICCR Programming Values

ICCR Value			80960VH Local Bus Frequency	SCL Transition Period	I²C 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 I²C Data Buffer Register (IDBR) and the I²C Slave Address Register (ISAR). The IDBR (see Table 21-9 "I2C Data Buffer Register – IDBR" on page 21-21) contains data or a slave address and R/W# bit (Figure 21-5). The ISAR contains the 80960VH's programmable slave address. Data coming into the I²C 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 I²C unit passes this onto the serial bus when the transfer byte bit in the ICR is set. See Section 21.10.1, "I2C Control Register - ICR" on page 21-15.

When the I²C 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 I²C 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 I²C bus and the acknowledge cycle is complete.
4. When the I²C bus is ready to transfer the next byte before the processor has written the IDBR (and a STOP condition is not in place), the I²C unit inserts wait states until the processor writes a new value into the IDBR and sets the ICR transfer byte bit.

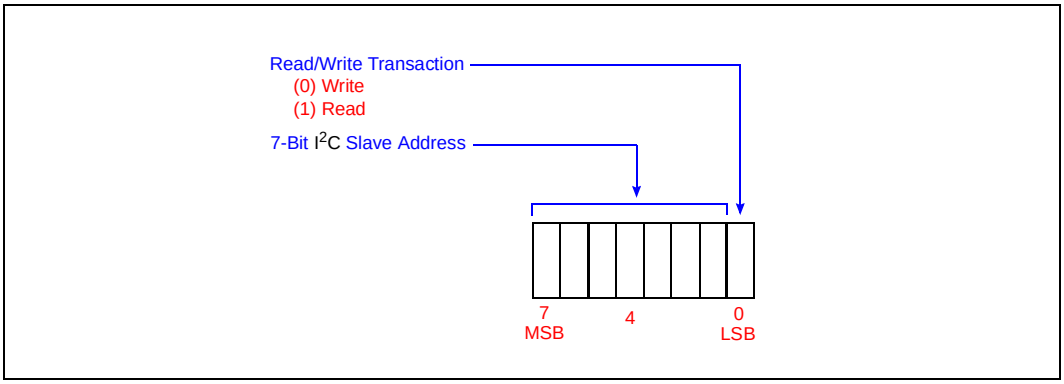
When the I²C unit is in receive mode (master or slave):

1. The processor reads the IDBR data over the 80960 local bus after the IDBR receive full interrupt is signaled.
2. The I2C unit transfers data from the shift register to the IDBR after the Ack cycle completes.
3. The I2C unit inserts wait states until the IDBR is read. Refer to Section 21.7, “I2C Acknowledge” on page 21-10 for acknowledge pulse information in receiver mode.
4. After the processor reads the IDBR, the I2C 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 I2C 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 I2C unit remains in master-transmit mode and the addressed slave device stays in slave-receive mode. When the transaction is a read, the I2C 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 I2C unit aborts the transaction by automatically sending a STOP and setting the ISR bus error bit.

When the I2C unit is enabled and idle (no bus activity), it stays in slave-receive mode and monitors the I2C bus for a START signal. Upon detecting a START pulse, the I2C unit reads the first seven bits and compares them to those in the I2C Slave Address Register (ISAR) and the general call address (00H). When the bits match those of the ISAR register, the I2C unit reads the eighth bit (R/W# bit) and transmits an Ack pulse. The I2C 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” on page 21-14 for actions when a general call address is detected.

21.6 Arbitration

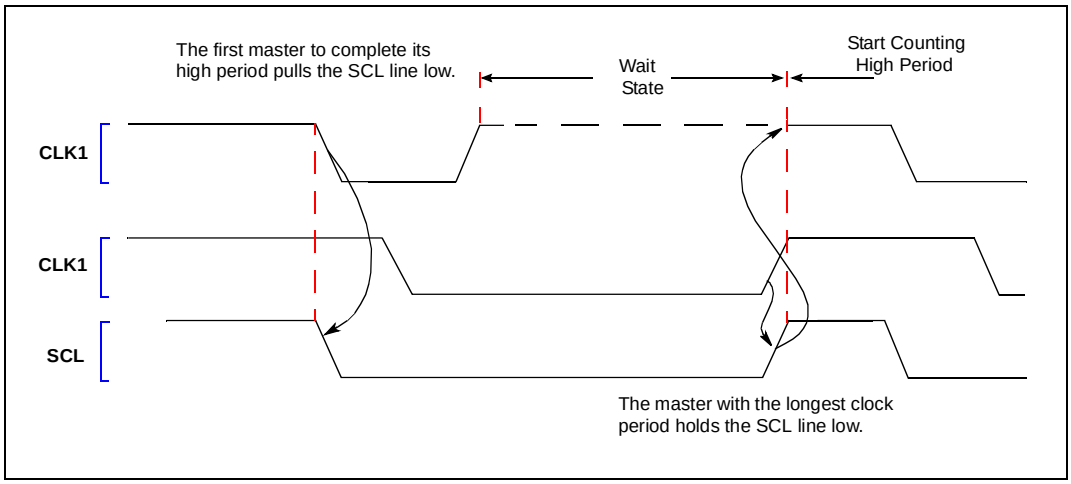
Arbitration on the I2C bus is required due to the multi-master capabilities of the I2C bus. Arbitration is used when two or more masters simultaneously generate a START condition within the minimum I2C hold time of the START condition. The following sections describe the arbitration on the SCL and SDA lines.

21.6.1 SCL Arbitration

Each master on the I2C 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 I2C 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.

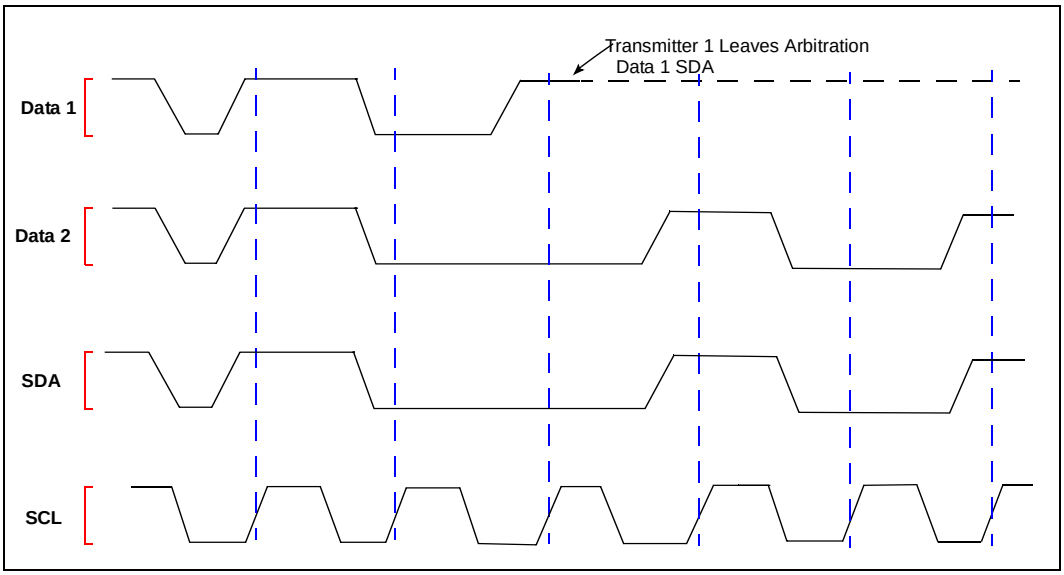
Figure 21-6. Clock Synchronization During the Arbitration Procedure



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 I2C 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 outputs the first high data bit loses arbitration and shuts its data drivers off. When the I2C 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 I2C unit loses arbitration during transmission of the seven address bits and the 80960VH is not being addressed as a slave device, the I2C unit resends the address when the I2C 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 80960VH as a slave device, the I2C unit switches to slave-receive mode and the original data in the I2C 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 I2C unit to write to its own slave address. This can cause the I2C 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 I2C bus. To prevent errors, the I2C 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 I2C 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 I²C Acknowledge

Every I²C 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).

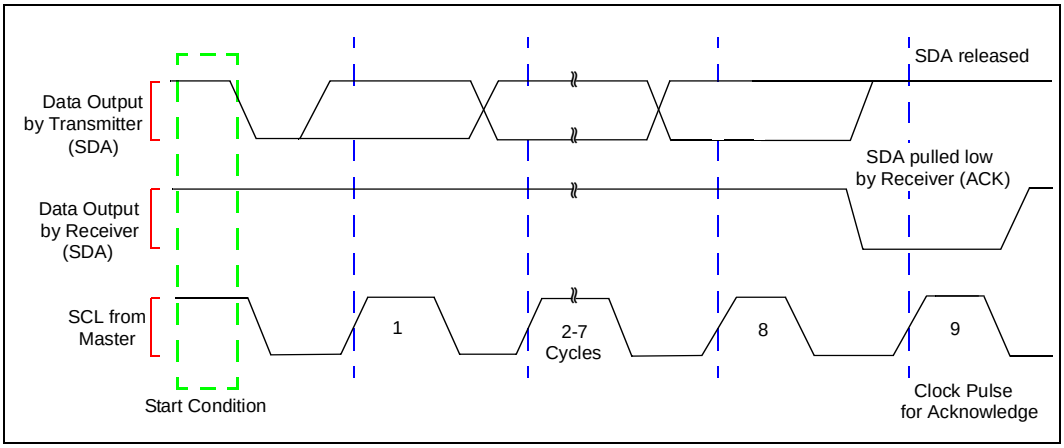
In master-transmit mode, when the target slave receiver device cannot generate the acknowledge pulse, the SDA line remains high. This lack of acknowledge (Nack) causes the I²C unit to set the bus error detected bit in the ISR and generate the associated interrupt (when enabled). The I²C unit aborts the transaction by generating a STOP automatically.

In master-receive mode, the I²C unit signals the slave-transmitter to stop sending data by using the negative acknowledge (Nack). The Ack/Nack bit value driven by the I²C 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 I²C bus protocol). When the transmit bit is set in the ICR, the I²C 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 I²C 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 I²C 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.

Figure 21-8. Acknowledge on the I²C Bus



21.8 I2C Master and Slave Operations

The I2C 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. • I2C unit sends the data. • I2C unit is responsible for clocking. • Slave device must be in slave-receive mode.
Master - Receive	• Used for a read operation on the bus. • I2C unit receives the data. • I2C unit is responsible for clocking. • Slave device must be in slave-transmit mode.
Slave - Transmit	• Used for a write operation on the bus. • I2C unit sends the data. • Master device must be in master-receive mode.
Slave - Receive (default)	• Used for a read operation on the bus. • I2C unit receives the data. • Master device must be in master-transmit mode.

The I2C unit enable bit (6) in the ICR must be set and the reset bit (14) cleared before the I2C unit may act as a master or slave device. When the I2C 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 80960VH.

The I2C unit transfers in 1-byte increments. A data transfer on the I2C 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 I2C bus, the I2C 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 I2C unit has the option of being one of two master modes:

- Master-Transmit — The 80960VH writes data
- Master-Receive — The 80960VH reads data

The 80960VH 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 I2C unit through the memory-mapped registers. When the 80960VH needs to read data, the I2C 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 I2C 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

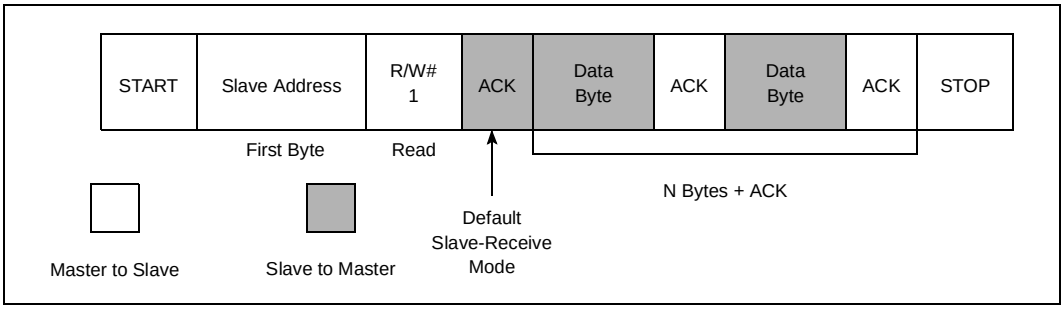


Figure 21-10. Master-Receiver Read from Slave-Transmitter / Repeated Start / Master-Transmitter Write to Slave-Receiver

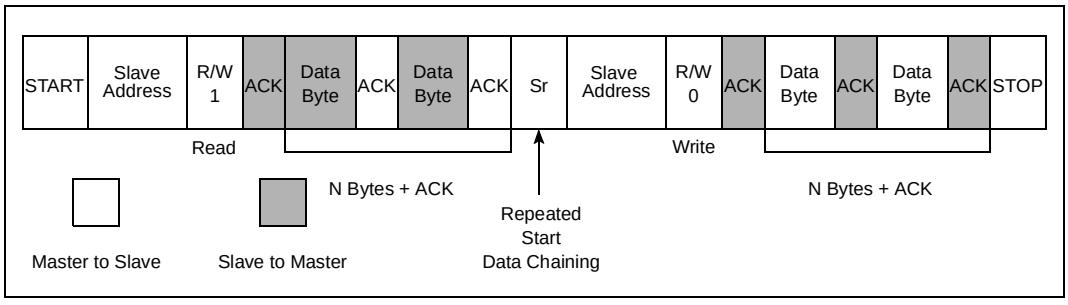
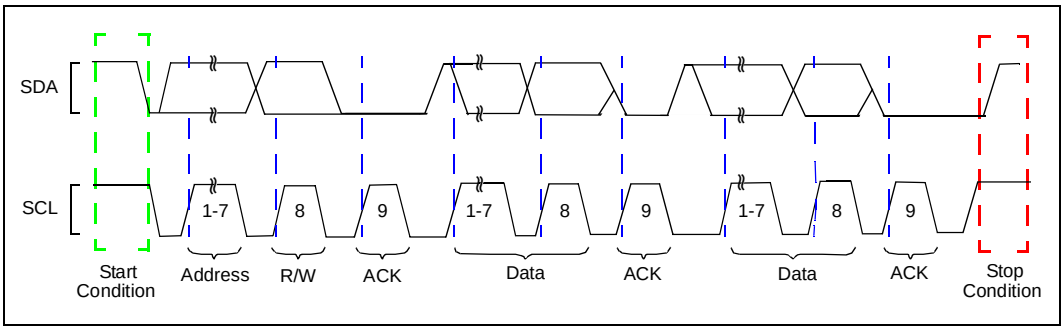


Figure 21-11. A Complete Data Transfer



The 80960VH initiates a master transaction by writing to the ICR register. Table 21-4 “General Call Address Second Byte Definitions” on page 21-15 describes the I²C unit responsibilities as a master device.

21.8.2 Slave Operations

Figure 21-12 through Figure 21-14 are examples of I²C transactions. These show the relationships between master and slave devices.

Figure 21-12. Master-Transmitter Write to Slave-Receiver

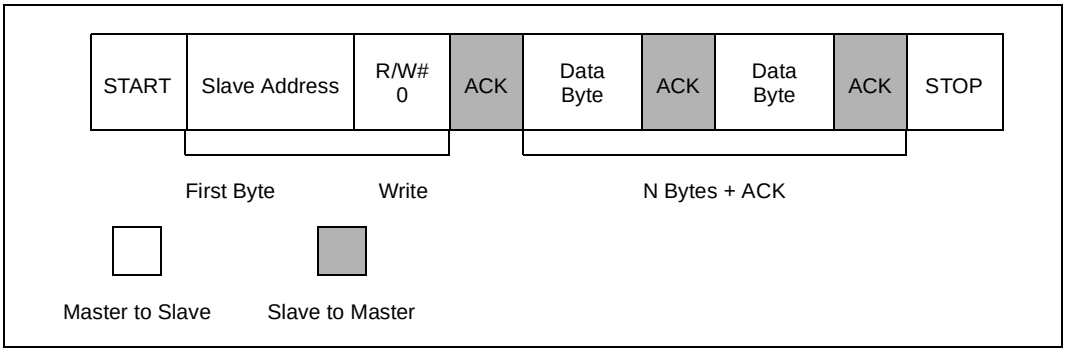


Figure 21-13. Master-Receiver Read to Slave-Transmitter

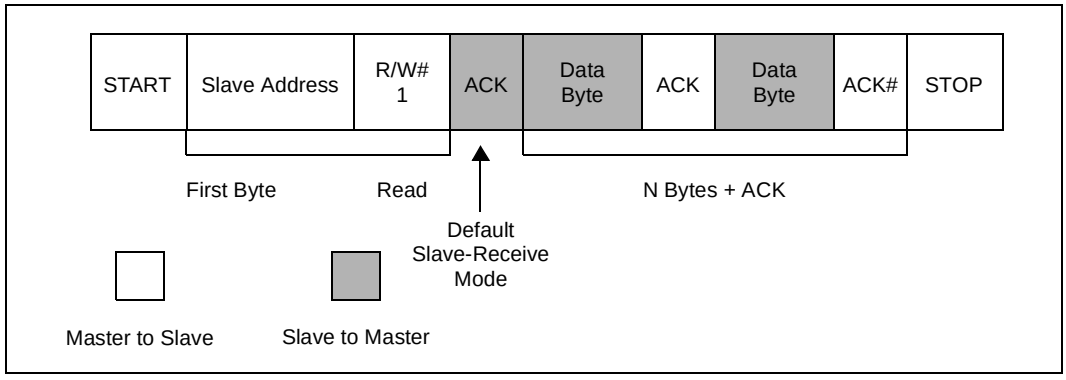
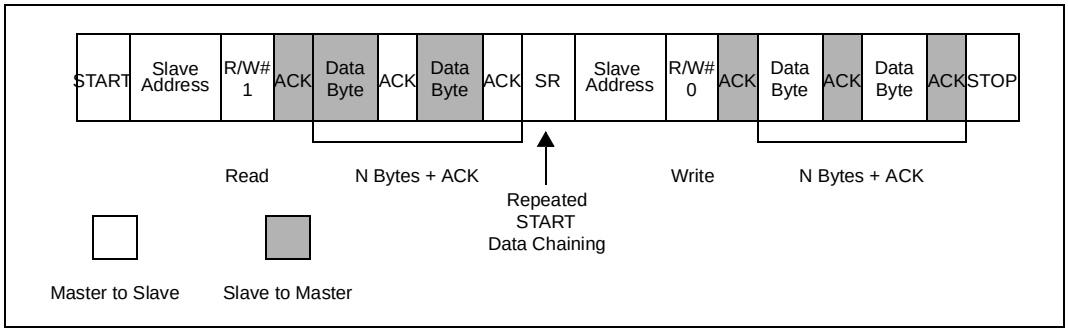


Figure 21-14. Master-Receiver Read to Slave-Transmitter, Repeated START, Master-Transmitter Write to Slave-Receiver



21.8.3 General Call Address

The I2C unit supports both sending and receiving general call address transfers on the I2C bus. When sending a general call message from the I2C unit, software must set the general call disable bit in the ICR to keep the I2C unit from responding as a slave. Failure to do this causes the I2C 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” on page 21-15 shows the valid values and definitions when B = 0.

When the 80960VH is acting as a slave, and the I2C unit receives a general call address and the ICR general call disable bit is clear the I2C unit:

- Sets the ISR general call address detected bit.
- Sets the ISR slave address detected bit.
- Signals an interrupt (when enabled) to the 80960VH.

When the I2C unit receives a general call address and the ICR general call disable bit is set, the I2C unit will ignore the general call address.

Figure 21-15. General Call Address

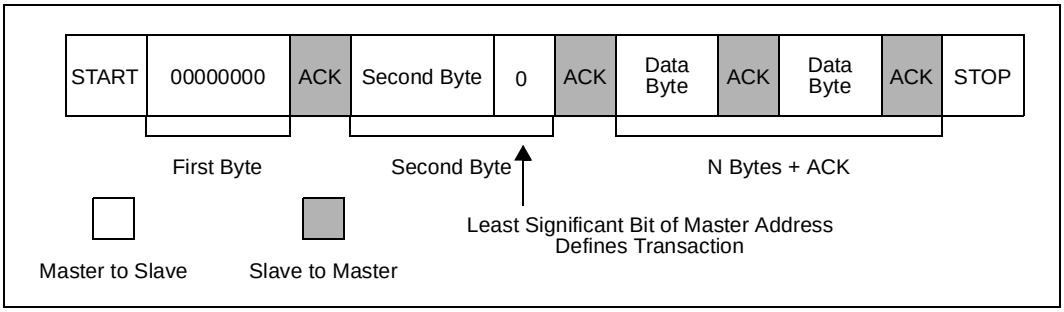


Table 21-4. General Call Address Second Byte Definitions

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 I²C Bus Unit and Reset

The I²C unit is reset by the local bus reset signal that is active when P_RST# is asserted or when reset local bus bit in Reset/Retry Control Register (RRCR) is set. Software is responsible for ensuring the I²C unit is not busy (unit busy is clear) before asserting reset. Software is also responsible for ensuring the I²C bus is idle when the unit is enabled after reset. When directed to reset, the I²C 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 80960VH I²C unit resets, the associated I²C MMRs remain intact. When resetting the I²C 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 I²C Registers

Table 21-5 identifies all I²C unit registers. Subsections identify all registers and define bit settings.

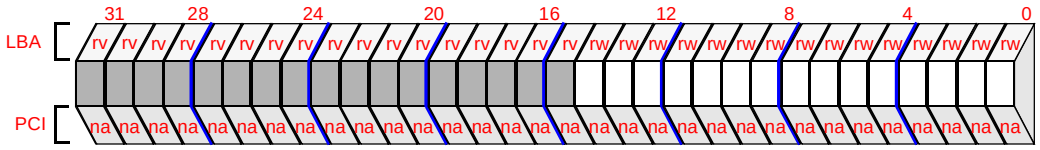
Table 21-5. I²C Register Summary

Section	Register Name, Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Config Addr Offset
21.10.1	I2C Control Register - ICR	21-15	32	0000 1680H	NA
21.10.2	I2C Status Register- ISR	21-18	32	0000 1684H	NA
21.10.3	I2C Slave Address Register – ISAR	21-20	32	0000 1688H	NA
21.10.4	I2C Data Buffer Register – IDBR	21-21	32	0000 168CH	NA
21.10.5	I2C Clock Count Register – ICCR	21-21	32	0000 1690H	NA

21.10.1 I²C Control Register - ICR

The 80960VH uses the bits in the I²C Control Register (ICR) to control the I²C unit.

Table 21-6. I²C Control Register – ICR (Sheet 1 of 3)

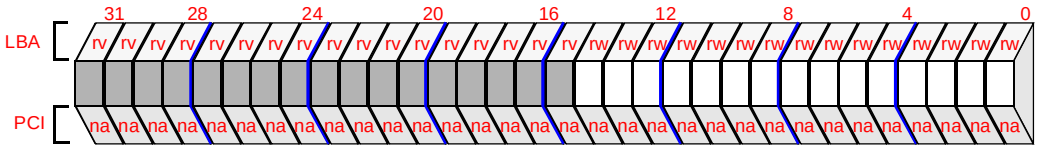


LBA: 1680H
PCI: NA

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
31:15	0000 0H	Reserved
14	0_2	**Unit Reset**: 1 = Reset the 80960VH I²C unit only. 0 = No reset.
13	0_2	**Slave Address Detected Interrupt Enable**: 1 = Enables the I²C unit to signal an interrupt to the 80960VH 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 I²C 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 I²C 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 I²C unit to signal an interrupt for the following I²C bus errors: • As a master transmitter, no Ack was detected after a byte was sent. • As a slave receiver, the I²C unit generated a Nack pulse. **Note:** Software must guarantee that misplaced START and STOP conditions do not occur. See Section 14.6, “Bus Arbitration” on page 14-23. 0 = Disable interrupt.
09	0_2	**IDBR Receive Full Interrupt Enable**: 1 = Enables the I²C unit to signal an interrupt to the 80960VH when the IDBR has received a data byte from the I²C bus. 0 = Disable interrupt.
08	0_2	**IDBR Transmit Empty Interrupt Enable**: 1 = Enables the I²C unit to signal an interrupt to the 80960VH after transmitting a byte onto the I²C bus. 0 = Disable interrupt.
07	0_2	**General Call Interrupt Disable**: 1 = Disables I²C unit response to general call messages as a slave. 0 = Enables the I²C unit to respond to general call messages. This bit must be set when sending a master mode general call message from the I²C unit.

Table 21-6. I²C Control Register – ICR (Sheet 2 of 3)

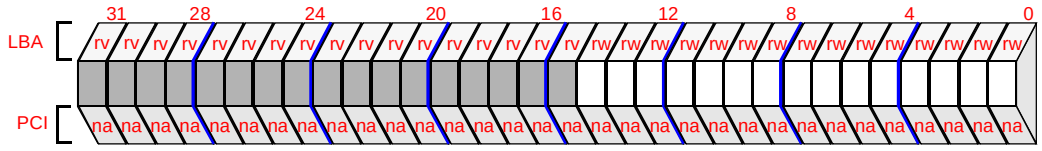


LBA: 1680H
PCI: NA

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
06	0_2	**I²C Unit Enable**: 1 = Enables the I²C 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 I²C bus is idle before setting this bit.
05	0_2	**SCL Enable**: 1 = Enables the I²C clock output for master mode operation. The ICCR (see Section 21.10.5, on page 21-21) must be programmed with a valid value before setting this bit. 0 = Disables the I²C unit from driving the SCL line.
04	0_2	**Master Abort**: used by the I²C unit when in master mode to generate a STOP without transmitting another data byte. 1 = The I²C unit sends STOP without data transmission. 0 = The I²C 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 80960VH does not send a repeated START, setting this bit sends the STOP. Once again, the transfer byte bit (03) must remain clear.
03	0_2	**Transfer Byte**: used to send/receive a byte on the I²C bus. 1 = send/receive a byte. 0 = cleared by I²C unit when the byte is sent/received. The 80960VH 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 I²C 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 I²C unit when in master or slave receive mode. 1 = The I²C unit sends a negative Ack (Nack) after receiving a data byte. 0 = The I²C unit sends an Ack pulse after receiving a data byte. The I²C 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 I²C 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, on page 21-4. for more details on the STOP state. 1 = Send a STOP 0 = Do not send a STOP

Table 21-6. I²C Control Register – ICR (Sheet 3 of 3)



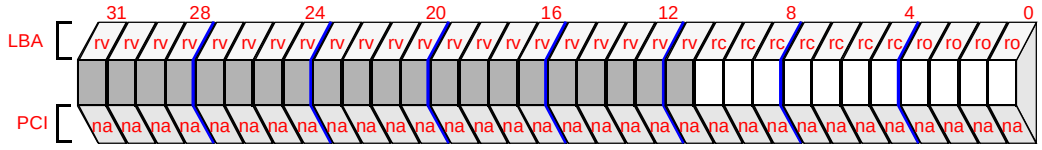
LBA: 1680H **PCI:** NA		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
00	0_2	**START**: used to initiate a START condition to the I²C unit when in master mode. See Section 21.3, on page 21-4. for more details on the START state. 1 = Send a START 0 = Do not send a START

21.10.2 I²C Status Register- ISR

I²C 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 I²C Interrupt Status Register (ISR). Software uses the ISR bits to check the status of the I²C unit and bus. ISR bits (bits 5-9) are updated after the Ack/Nack bit has completed on the I²C bus. The ISR is also used to clear interrupts signaled from the I²C unit. They are:

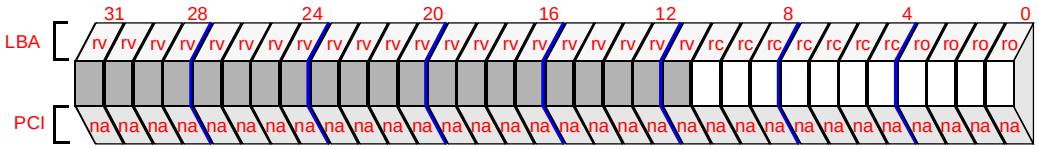
- IDBR receive full
- IDBR transmit empty
- slave address detected
- bus error detected
- STOP condition detect
- arbitration lost

Table 21-7. I²C Status Register – ISR (Sheet 1 of 3)



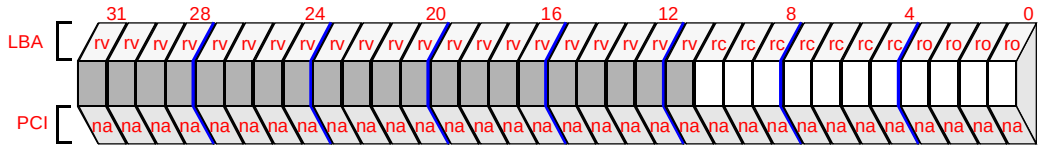
LBA: 1684H **PCI:** NA		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:11	0000 00H	Reserved

Table 21-7. I²C Status Register – ISR (Sheet 2 of 3)



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**
10	0_2	**Bus Error Detected**: 1 = The I²C 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 I²C unit generates a Nack pulse. **Note:** When an error occurs, I²C bus transactions continue. Software must guarantee that misplaced START and STOP conditions do not occur. See Section 21.6, "Arbitration" on page 21-7. 0 = no error detected.
09	0_2	**Slave Address Detected**: 1 = I²C 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.
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 80960VH'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).

Table 21-7. I²C Status Register – ISR (Sheet 3 of 3)



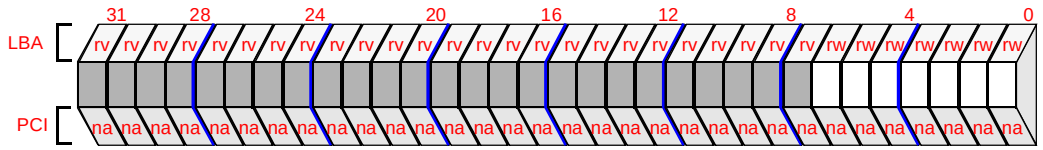
LBA: 1684H **PCI:** NA	**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset

Bit	Default	Description
02	0_2	Unit Busy: 1 = Set when the 80960VH's I²C unit is busy. This is defined as the time between the first START and STOP. 0 = I²C unit not busy.
01	0_2	**Ack/Nack Status**: 1 = The I²C unit received a Nack. 0 = The I²C 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 I²C unit is in receive mode. 0 = The I²C 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 I²C Slave Address Register – ISAR

The I²C Slave Address Register (see Table 21-8) defines the I²C unit's 7-bit slave address to which the 80960VH responds when in slave-receive mode. This register is written by the 80960VH before enabling I²C operations. The register is fully programmable (no address is assigned to the I²C unit) so it can be set to a value other than those of hard-wired I²C slave peripherals that might exist in the system. The ISAR is not affected by the 80960VH being reset. The ISAR register default value is 00H.

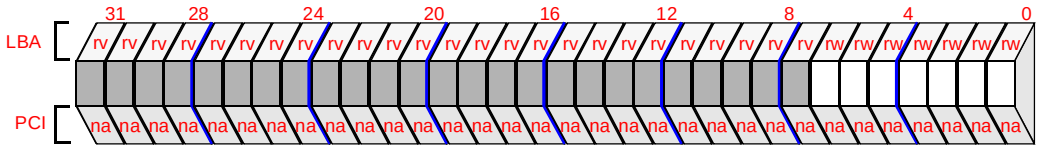
Table 21-8. I²C Slave Address Register – ISAR (Sheet 1 of 2)



LBA: 1688H **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
31:07	0000 000H	Reserved

Table 21-8. I²C Slave Address Register – ISAR (Sheet 2 of 2)

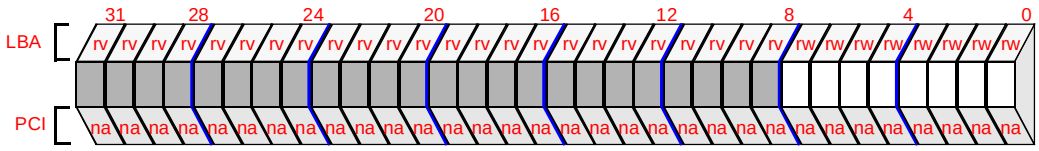


LBA: 1688H **PCI:** NA		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
06:00	0H	**I²C Slave Address**: The 7-bit address to which the I²C unit responds when in slave-receive mode.

21.10.4 I²C 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. I²C Data Buffer Register – IDBR



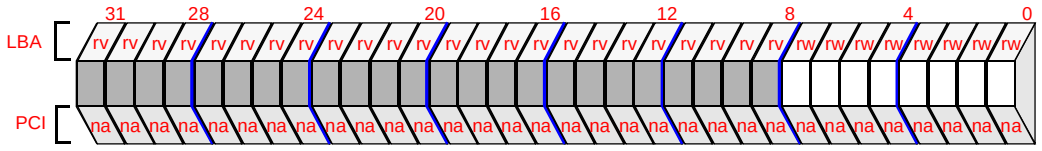
LBA: 168CH **PCI:** NA		**Legend:** NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 local bus address PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:08	0000 00H	Reserved
07:00	00H	**I²C Data Buffer**: Buffer for I²C bus send/receive data.

21.10.5 I²C Clock Count Register – ICCR

The I²C Clock Count Register (ICCR) defines the multiplier used to generate the I²C 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 I²C clock output used during I²C master operations.

Changing this register while the SCL enable bit is set results in undefined behavior.

Table 21-10. I²C Clock Count Register – ICCR



LBA: 1690H **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	**I²C Clock Count**: 8 bit count value used to generate an I²C clock from the 80960VH local bus clock.

Test Features 22

This chapter describes the i960® VH 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.

22.1 On-Circuit Emulation (ONCE)

On-circuit emulation aids board-level testing. This feature allows a mounted 80960VH 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 80960VH system.

Note: Do not use ONCE mode with boundary-scan (JTAG). See Section 22.1.2, “ONCE Mode and Boundary-Scan (JTAG) are Incompatible” on page 22-2.

22.1.1 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 *80960VH Processor Data Sheet* for specific timing of the ONCE# pin and the characteristics of the on-circuit emulation mode.

22.1.2 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 80960VH signals while JTAG is active.

Since the 80960VH 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.

22.1.3 How to use the Data Enable (DEN#) Signal with an In-Circuit Emulator

When using an ICE in an 80960VH 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 80960VH'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, then the transceivers are enabled. This may result in bus contention.

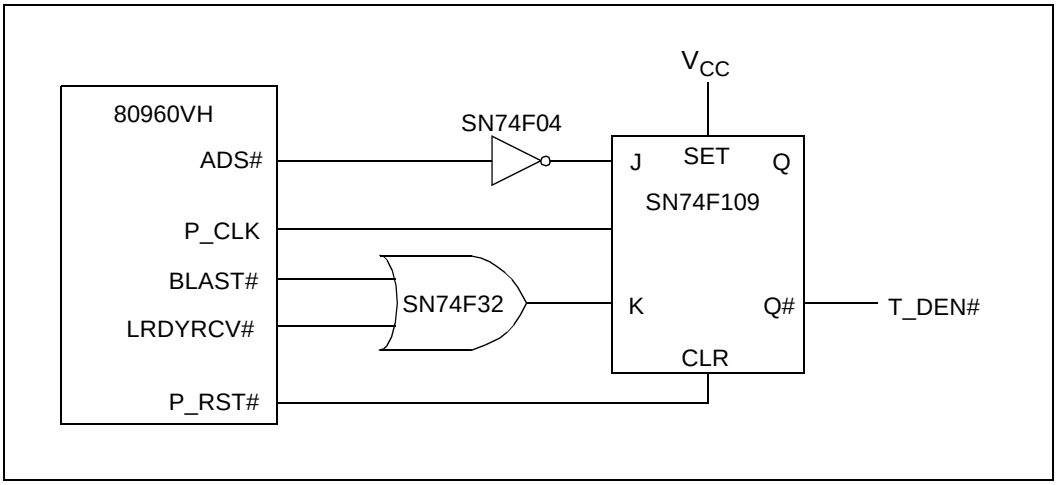
The use of DEN# in 80960Jx designs was possible because the 80960Jx was on a "POD". The POD was cabled to the target board where it plugged in to a socket. The POD masks out DEN# during ICE microcode transactions. The 80960VH's package does not allow the use of a POD; consequently, the ICE signals connect directly to the target system and the DEN# signal cannot be masked.

22.1.3.1 DEN# Alternatives

To use an ICE with your 80960VH design, alternatives to DEN# are:

- Ground the OE# pin of the transceiver
- Recreate a DEN# signal with the circuit shown below

Figure 22-1. DEN# Alternatives



The circuit asserts T_DEN# (Q#) at the start of the first data cycle when ADS# asserts and BLAST# and LRDYRCV# deasserts. T_DEN# deasserts at the end of the last data cycle when ADS# deasserts and BLAST# and LRDYRCV# assert. During RESET, T_DEN# deasserts.

Equivalent components may be used in place of the components shown.

22.2 Boundary-Scan (JTAG)

The 80960VH Internal Bus 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).

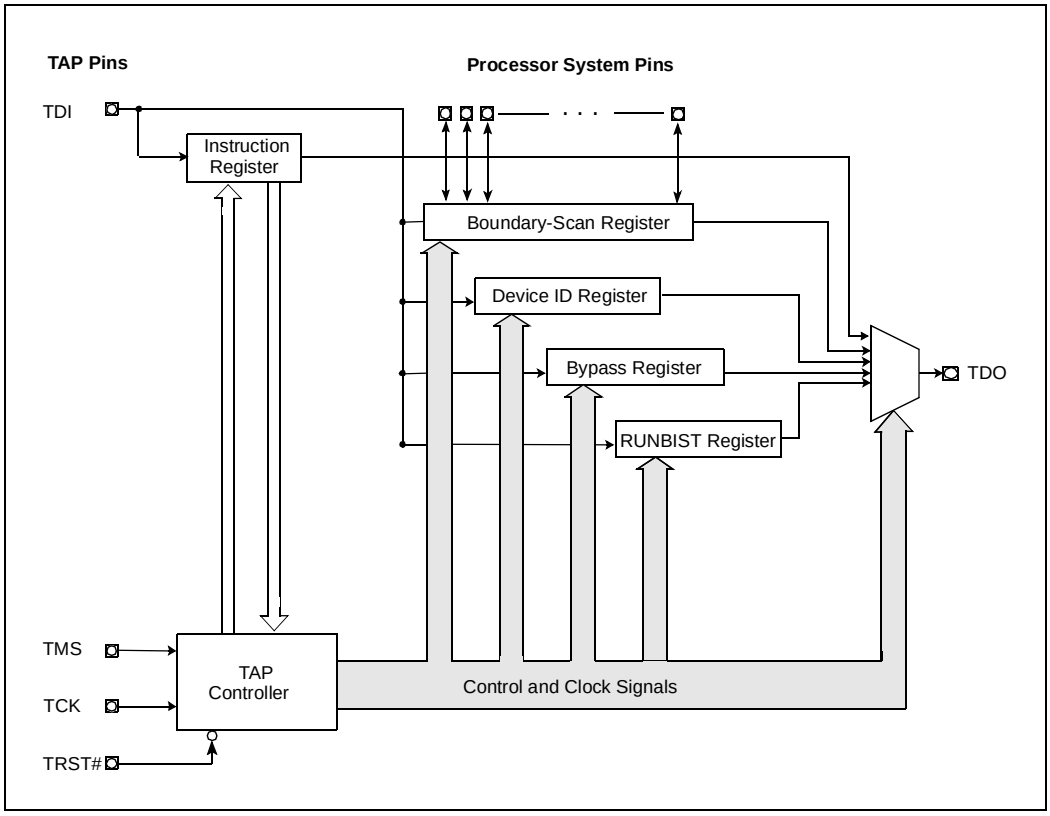
Note: Do not use ONCE mode with boundary-scan (JTAG). See Section 22.1.2, “ONCE Mode and Boundary-Scan (JTAG) are Incompatible” on page 22-2.

22.2.1 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 22-2 illustrates how these pieces fit together to form the JTAG unit.

Figure 22-2. Test Access Port Block Diagram



22.2.2 TAP Pins

The 80960VH'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 22-1. The TAP pins provide access to the instruction register and the test data registers.

Table 22-1. TAP Controller Pin Definitions (Sheet 1 of 2)

Pin	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.

Table 22-1. TAP Controller Pin Definitions (Sheet 2 of 2)

Pin	Type	Definition
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 to put the TAP controller in a known state. Failure to do so may cause improper processor operation.

22.2.3 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 terminate.

The instruction determines the test to be performed, the test data register to be accessed, or both (Table 22-2). 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 shifts, and becomes active on the falling edge of TCK. See Figure 22-5 for an example of loading the instruction register.

22.2.3.1 Boundary-Scan Instruction Set

The 80960VH supports three mandatory boundary-scan instructions (**bypass**, **sample**/**preload** and **extest)** plus four additional public instructions (**idcode**, **clamp**, **highz** and **runbist)**. Table 22-2 lists the 80960VH’s boundary-scan instruction codes. Those codes listed as “not used” or “private” should not be used.

Table 22-2. Boundary-Scan Instruction Set (Sheet 1 of 2)

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**

Table 22-2. Boundary-Scan Instruction Set (Sheet 2 of 2)

Instruction Code	Instruction Name	Instruction Code	Instruction Name
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 22-3. 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® VH 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 deasserted 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 22-3. 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 do not change while the **clamp** instruction is selected.

22.2.4 TAP Test Data Registers

The 80960VH 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 [a?] register is selected, data is transferred from TDI to TDO without inversion. The following sections describe each of the test data registers. See Figure 22-6 for an example of loading the data register.

22.2.4.1 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 *80960VH Processor Data Sheet*. The format of the register is discussed in Section 12.5, "Device Identification on Reset" on page 12-20. 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.

22.2.4.2 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.

22.2.4.3 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 12.3.1, "Self Test Function (STEST, FAIL#)" on page 12-7 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.

22.2.4.4 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 22-4 shows the bit order of the 80960VH 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 80960VH'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 22-4. i960® VH Processor Boundary Scan Register Bit Order (Sheet 1 of 6)

Bit	Signal	Input/Output
0	LRDYRCV#/STEST	I/O
1	RDYRCV#	I
2	control	enable cell
3	NMI#	I
4	HOLDA	O
5	HOLD	I
6	LCDINIT#	I
7	LOCK#/ONCE#	I/O
8	D/C#/RST_MODE#	I/O
9	FAIL#	O
10	WIDTH/HLTD0	I/O
11	WIDTH/HLTD1/RETRY	I/O
12	LRST#	O
13	XINT0#	I

Table 22-4. i960® VH Processor Boundary Scan Register Bit Order (Sheet 2 of 6)

Bit	Signal	Input/Output
14	XINT1#	I
15	XINT2#	I
16	XINT3#	I
17	XINT4#	I
18	XINT5#	I
19	control	enable cell
20	control	enable cell
21	control	enable cell
22	XINT6#	I
23	XINT7#	I
24	RAS0#	O
25	RAS1#	O
26	RAS2#	O
27	RAS3#	O
28	CAS0#	O
29	CAS1#	O
30	CAS2#	O
31	CAS3#	O
32	CAS4#	O
33	CAS5#	O
34	CAS6#	O
35	CAS7#	O
36	MWE0#	O
37	MWE1#	O
38	MWE2#	O
39	MWE3#	O
40	DWE0#	O
41	DWE1#	O
42	CE0#	O
43	CE1#	O
44	LEAF0#	O
45	LEAF1#	O
46	DALE0	O
47	DALE1	O
48	control	enable cell
49	MA0	O
50	MA1	O
51	MA2	O

Table 22-4. i960® VH Processor Boundary Scan Register Bit Order (Sheet 3 of 6)

Bit	Signal	Input/Output
52	MA3	O
53	MA4	O
54	MA5	O
55	MA6	O
56	MA7	O
57	MA8	O
58	MA9	O
59	MA10	O
60	MA11	O
61	DP0	I/O
62	DP1	I/O
63	control	enable cell
64	DP2	I/O
65	DP3	I/O
66	CLKMODE0#	I
67	CLKMODE1#	I
68	DREQ#	I
69	DACK#	I/O
70	WAIT#	O
71	control	enable cell
72	control	enable cell
73	P_AD0	I/O
74	P_AD1	I/O
75	P_AD2	I/O
76	P_AD3	I/O
77	P_AD4	I/O
78	P_AD5	I/O
79	P_AD6	I/O
80	P_AD7	I/O
81	P_C/BE0#	I/O
82	P_AD8	I/O
83	P_AD9	I/O
84	P_AD10	I/O
85	P_AD11	I/O
86	P_AD12	I/O
87	control	enable cell
88	P_AD13	I/O
89	P_AD14	I/O

Table 22-4. i960® VH Processor Boundary Scan Register Bit Order (Sheet 4 of 6)

Bit	Signal	Input/Output
90	P_AD15	I/O
91	P_C/BE1#	I/O
92	P_PAR	I/O
93	control	enable cell
94	control	enable cell
95	control	enable cell
96	P_SERR#	I/O
97	P_PERR#	I/O
98	P_LOCK#	I
99	P_STOP#	I/O
100	P_DEVSEL#	I/O
101	control	enable cell
102	control	enable cell
103	P_TRDY#	I/O
104	P_IRDY#	I/O
105	P_FRAME#	I/O
106	P_C/BE2#	I/O
107	control	enable cell
108	control	enable cell
109	P_AD16	I/O
110	P_AD17	I/O
111	P_AD18	I/O
112	P_AD19	I/O
113	P_AD20	I/O
114	P_AD21	I/O
115	P_AD22	I/O
116	P_AD23	I/O
117	P_IDSEL	I
118	P_C/BE3#	I/O
119	control	enable cell
120	P_AD24	I/O
121	P_AD25	I/O
122	P_AD26	I/O
123	P_AD27	I/O
124	P_AD28	I/O
125	control	enable cell
126	P_AD29	I/O
127	P_AD30	I/O

Table 22-4. i960® VH Processor Boundary Scan Register Bit Order (Sheet 5 of 6)

Bit	Signal	Input/Output
128	P_AD31	I/O
129	P_REQ#	O
130	P_GNT#	I
131	control	enable cell
132	control	enable cell
133	control	enable cell
134	control	enable cell
135	control	enable cell
136	P_RST#	I
137	P_INTD#	O
138	P_INTC#	O
139	P_INTB#	O
140	P_INTA#	O
141	SCL	I/O
142	SDA	I/O
143	control	enable cell
144	control	enable cell
145	BLAST#	I/O
146	ADS#	O
147	W/R#	O
148	DT/R#	O
149	DEN#	I/O
150	ALE	O
151	BE0#	O
152	BE1#	O
153	control	enable cell
154	control	enable cell
155	BE2#	O
156	BE3#	O
157	AD31	I/O
158	AD30	I/O
159	AD29	I/O
160	AD28	I/O
161	AD27	I/O
162	AD26	I/O
163	AD25	I/O
164	AD24	I/O
165	AD23	I/O

Table 22-4. i960® VH Processor Boundary Scan Register Bit Order (Sheet 6 of 6)

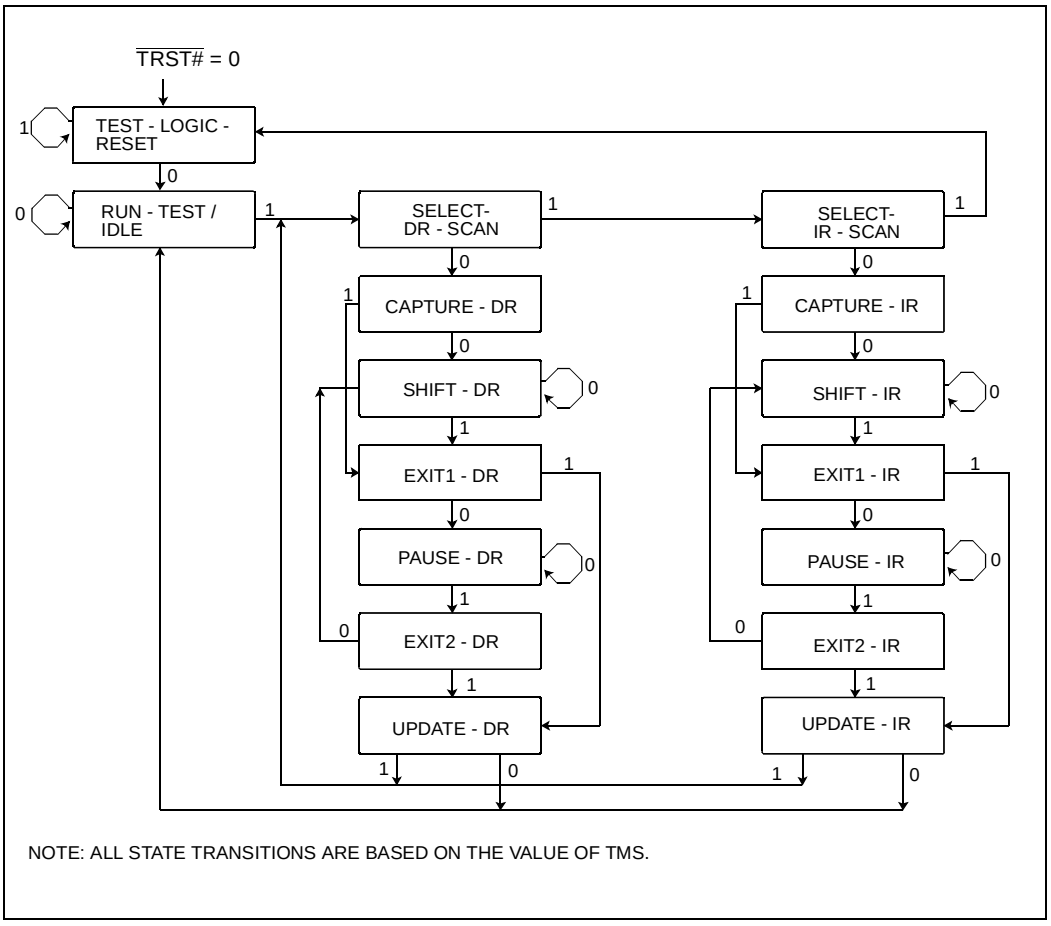
Bit	Signal	Input/Output
166	AD22	I/O
167	AD21	I/O
168	AD20	I/O
169	AD19	I/O
170	P_CLK	I
171	AD18	I/O
172	control	enable cell
173	AD17	I/O
174	AD16	I/O
175	AD15	I/O
176	AD14	I/O
177	AD13	I/O
178	AD12	I/O
179	AD11	I/O
180	AD10	I/O
181	AD9	I/O
182	AD8	I/O
183	AD7	I/O
184	AD6	I/O
185	AD5	I/O
186	control	enable cell
187	AD4	I/O
188	AD3	I/O
189	AD2	I/O
190	AD1	I/O
191	AD0	I/O

22.2.5 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 22-3 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 22-3. TAP Controller State Diagram



22.2.5.1 Test Logic Reset State

In this state, test logic is disabled to allow normal operation of the 80960VH. 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.

22.2.5.2 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.

22.2.5.3 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 22.2.5.10, “Select-IR Scan State” on page 22-16. The instruction register does not change while the TAP controller is in this state.

22.2.5.4 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.

22.2.5.5 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.

22.2.5.6 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.

22.2.5.7 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.

22.2.5.8 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.

22.2.5.9 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.

22.2.5.10 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.

22.2.5.11 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.

22.2.5.12 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.

22.2.5.13 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.

22.2.5.14 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.

22.2.5.15 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.

22.2.5.16 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.

22.2.6 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 22-4 for an illustration of the waveforms involved in this example. The steps are:

1. Load the **sample**/**preload** instruction into the instruction register:
   a. Use TMS to select the Shift-IR state. While in the Shift-IR state, shift in the new instruction, least significant byte first.
   b. Use the Shift-IR state four times to read the least-significant through most-significant instruction bits into the instruction register (one does not care what old instruction is being shifted out of the TDO pin).
   c. Enter the Update-IR state to make the instruction take effect.
2. Capture pin data and shift the data out through the TDO pin:
   a. Use TMS to select the Select-DR-Scan state.
   b. Transition the TAP controller to the Capture-DR state to latch pin data in the boundary-scan register cells.
   c. 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.
   d. 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.
   e. 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. At 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 22-4. Example Showing Typical JTAG Operations

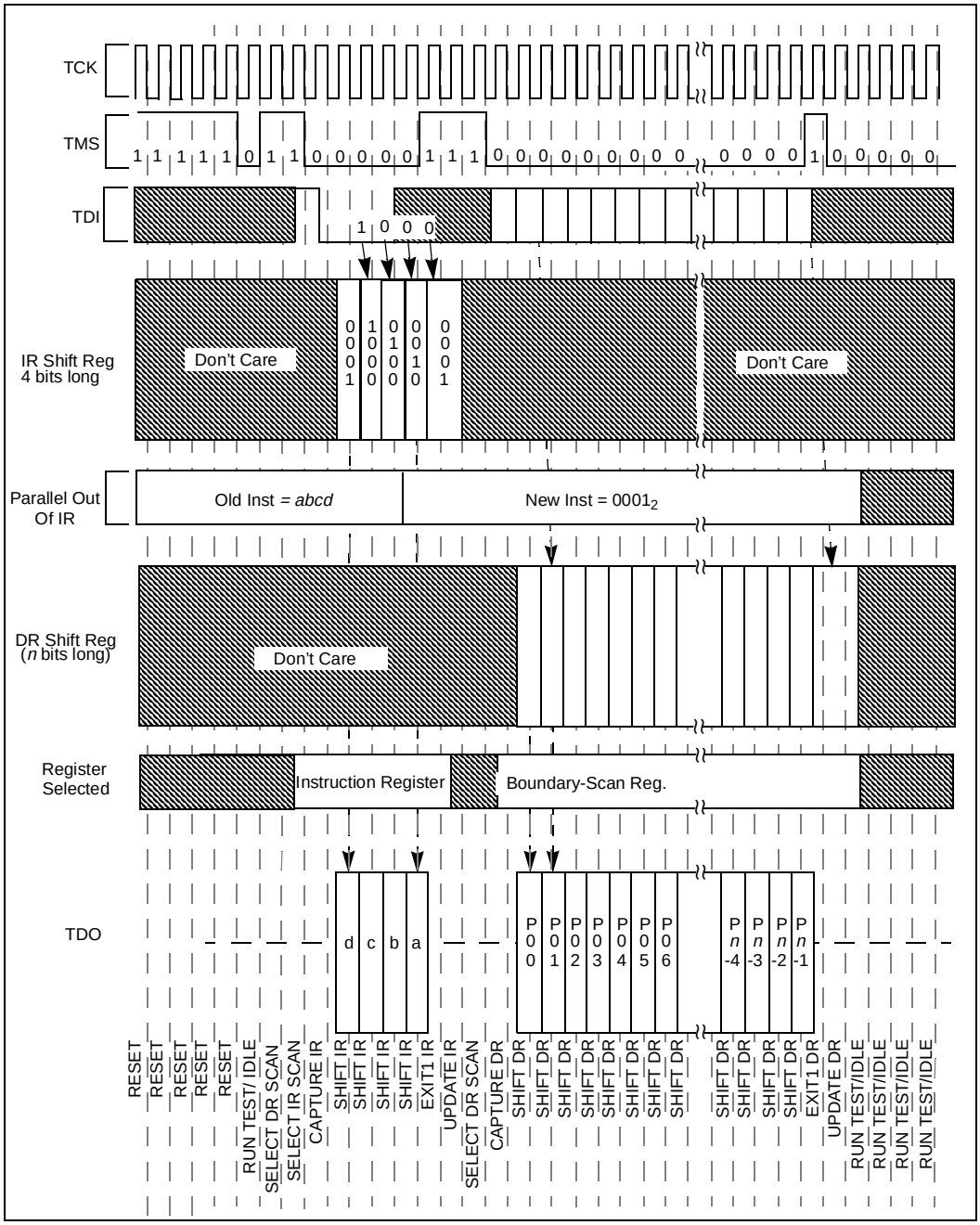


Figure 22-5. Timing Diagram Illustrating the Loading of Instruction Register

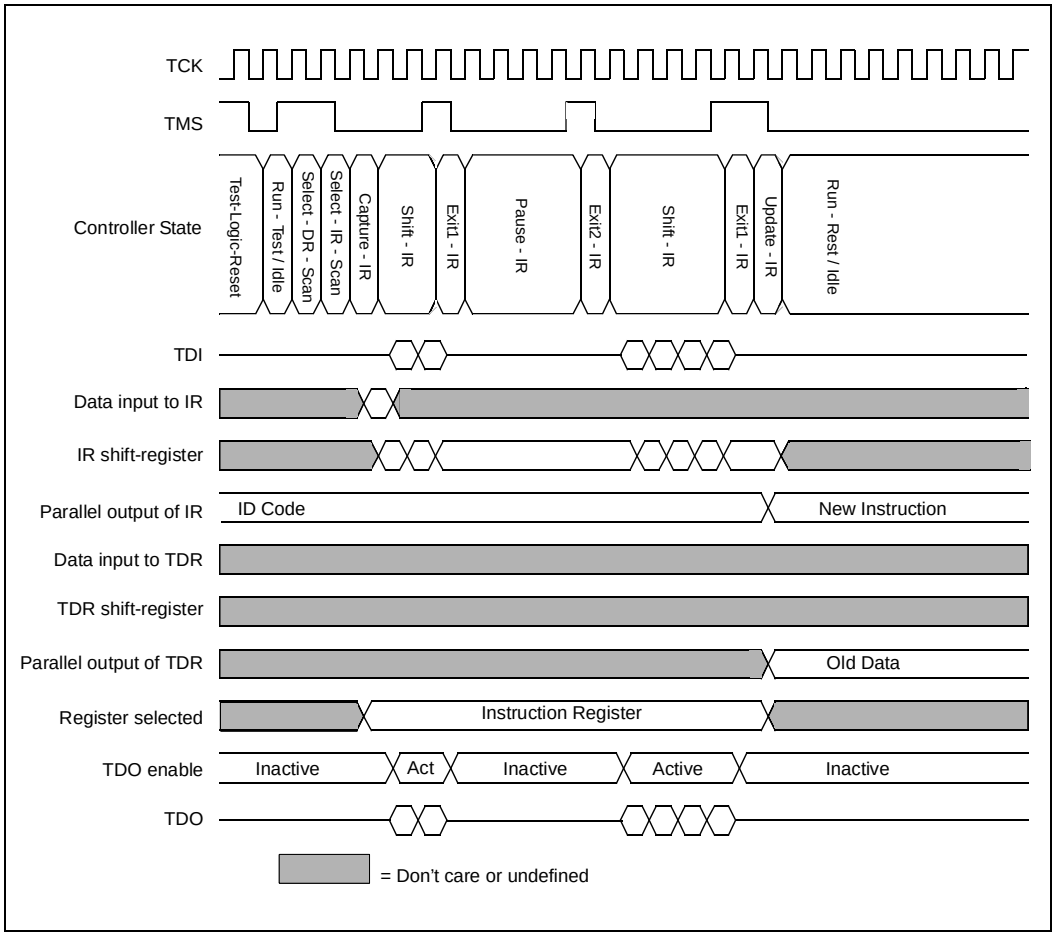
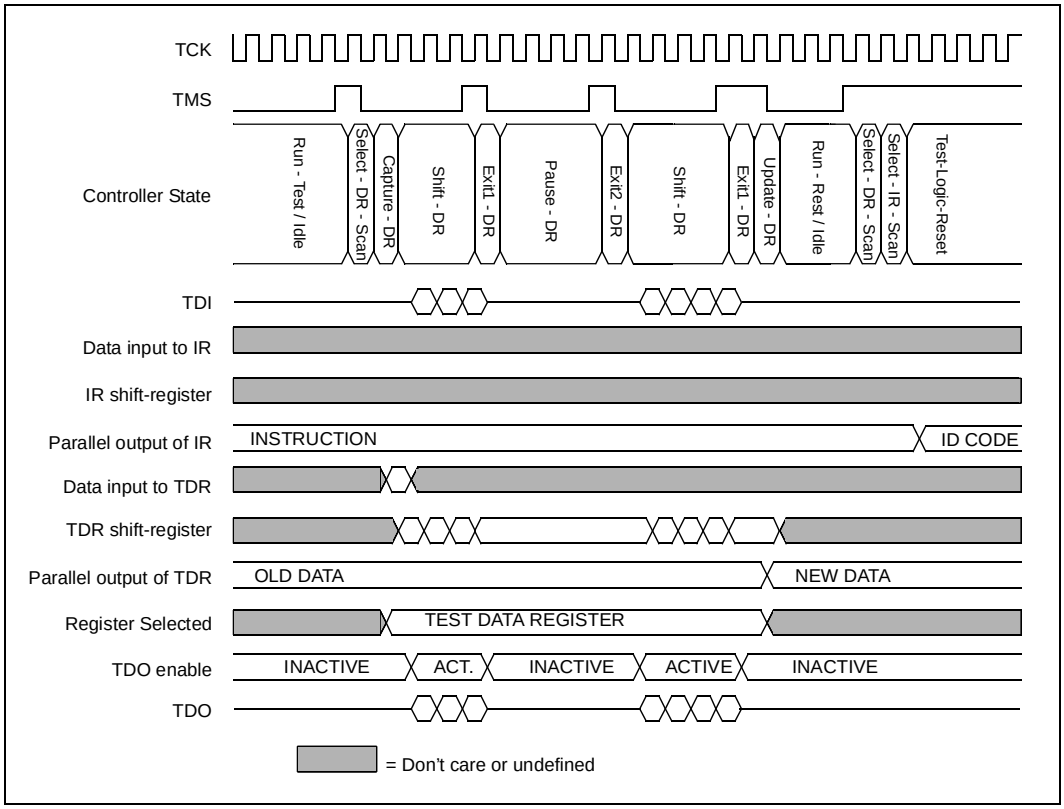


Figure 22-6. Timing Diagram Illustrating the Loading of Data Register



Machine-level Instruction Formats A

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

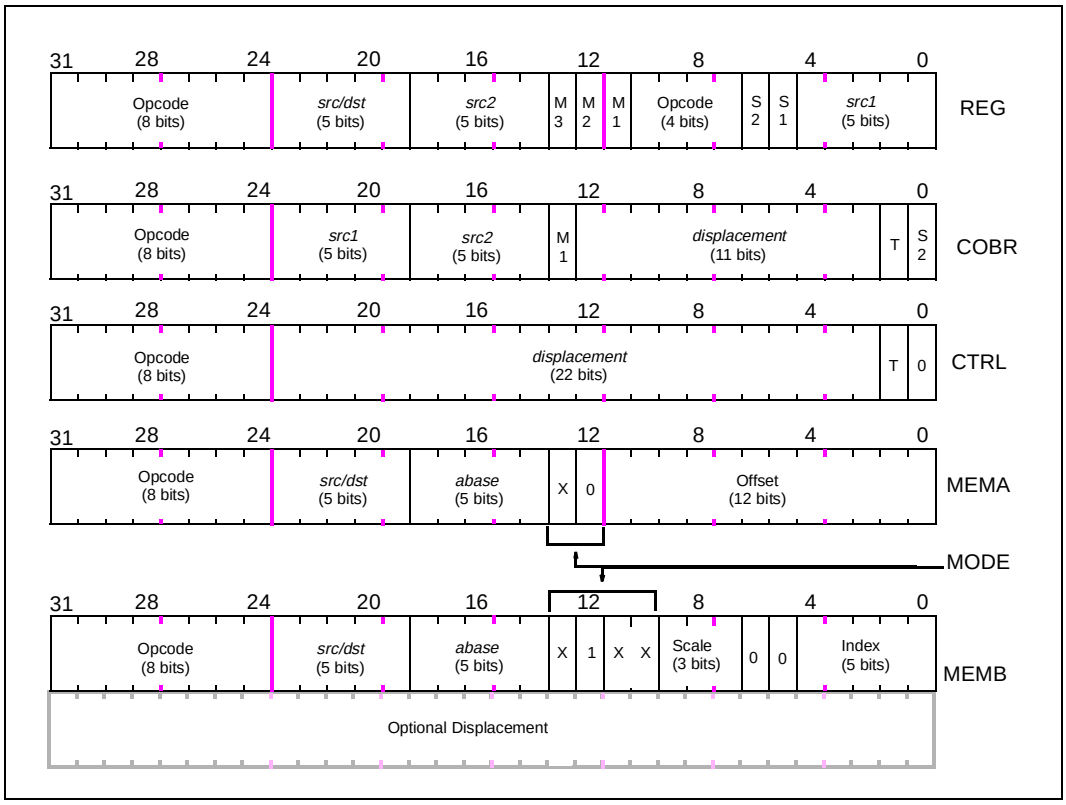


Table A-1. Instruction Field Descriptions

Instruction Field	Description
Opcode	The opcode of the instruction. Opcode encodings are defined in Section 6.1.8, "Opcode and Instruction Format" on page 6-4.
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 global 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, then the *src/dst* operand is a global or local register that is encoded as shown in Table A-3. If M3 is set, then 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
- register indirect
- register indirect with displacement
- index with displacement

- IP 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.

Opcodes and Execution Times B

B.1 Instruction Reference by Opcode

This section lists the instruction encoding for each i960® VH 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	—	x	reserved

Table B-2. REG Format Instruction Encodings (Sheet 1 of 6)

Opcode	**Mnemonic**	**Cycles to Execute**	*Opcode (11 - 4)*	***src/dst***	***src2***	*Mode*			*Opcode (3-0)*	*Special Flags*		***src1***
			31 24	2319	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*

NOTE: Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 2 of 6)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
			31 24	2319	18 .14	13	12	11	10 7	6	5	4 0
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*

NOTE: Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 3 of 6)

Opcode	**Mnemonic**	**Cycles to Execute**	*Opcode (11 - 4)*	***src/dst***	***src2***	*Mode*			*Opcode (3-0)*	*Special Flags*		***src1***
			31 24	2319	18 .14	13	*12*	*11*	*10 7*	*6*	5	*4 0*
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*
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*

NOTE: Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 4 of 6)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
			31 24	2319	18 .14	13	12	11	10 7	6	5	4 0
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*
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*

NOTE: Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 5 of 6)

Opcode	**Mnemonic**	**Cycles to Execute**	*Opcode (11 - 4)*	***src/dst***	***src2***	*Mode*			*Opcode (3-0)*	*Special Flags*		***src1***
			31 24	2319	18 .14	13	*12*	*11*	*10 7*	*6*	5	*4 0*
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*
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*

NOTE: Execution time based on function performed by instruction.

Table B-2. REG Format Instruction Encodings (Sheet 6 of 6)

Opcode	Mnemonic	Cycles to Execute	Opcode (11 - 4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
			31 24	2319	18 .14	13	12	11	10 7	6	5	4 0
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*

NOTE: Execution time based on function performed by instruction.

Table B-3. COBR Format Instruction Encodings (Sheet 1 of 2)

Opcode	Mnemonic	Cycles to Execute	Opcode	src1	src2	M	Displacement	T	S2
			3124	23 19	1814	13	122	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

Table B-3. COBR Format Instruction Encodings (Sheet 2 of 2)

Opcode	Mnemonic	Cycles to Execute	Opcode	*src1*	*src2*	M	Displacement	T	S2
			3124	*23 19*	*1814*	*13*	*122*	*1*	*0*
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*

NOTE: 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 (Sheet 1 of 2)

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*

NOTE: 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-4. CTRL Format Instruction Encodings (Sheet 2 of 2)

Opcode	Mnemonic	Cycles to Execute	Opcode	*displacement*	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*

NOTE: 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

Table B-7. Cycle Counts for dcctl Operations

Operation	Cycles to Execute
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

3124	23.19	1814	1312	110
Opcode	***src/dst***	**ABASE**	**Mode**	**Offset**

3124	23.19	1814	13121110	97	65	40
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									

Table B-9. MEM Format Instruction Encodings

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
80	**ldob**	
82	**stob**	
84	**bx**	4-7
85	**balx**	5-8
86	**callx**	9-12
88	**ldos**	
8A	**stos**	
8C	**lda**	
90	**ld**	
92	**st**	
98	**ldl**	

Opcode	Mnemonic	Cycles to Execute
9A	**stl**	
A0	**ldt**	
A2	**stt**	
B0	**ldq**	
B2	**stq**	
C0	**ldib**	
C2	**stib**	
C8	**ldis**	
CA	**stis**	

NOTE: 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

Memory-Mapped Registers C

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® VH 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 are 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. Seen Section 10.2.7.2, "Hardware Breakpoints" on page 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. An **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
13.5.1	Default Logical Memory Configuration Register – DLMCON	13-7	FF00 8100H
	Reserved		FF00 8104H
13.5.1	Logical Memory Address Registers – LMADR0:1-0	13-6	FF00 8108H
13.5.1	Logical Memory Mask Registers – LMMR0:1-0	13-7	FF00 810CH
13.5.1	Logical Memory Address Registers – LMADR0:1-1	13-6	FF00 8110H
13.5.1	Logical Memory Mask Registers – LMMR0:1-1	13-7	FF00 8114H
	Reserved		FF00 8118H to FF00 83FFH
10.2.7.6	Instruction Breakpoint Registers – IPBx	10-8	FF00 8400H to FF00 8404H
	Reserved		FF00 8408H to FF00 841FH
10.2.7.5	Data Address Breakpoint Registers – DABx	10-7	FF00 8420H to FF00 8424H
	Reserved		FF00 8428H to FF00 843FH
10.2.7.4	Breakpoint Control Register – BPCON	10-6	FF00 8440H
	Reserved		FF00 8444H to FF00 84FFH
8.4.4	Interrupt Mask – IMSK and Interrupt Pending Registers – IPND	8-27	FF00 8500H
8.4.4	Interrupt Mask – IMSK and Interrupt Pending Registers – IPND	8-27	FF00 8504H
	Reserved		FF00 8508H to FF00 850FH
8.4.2	Interrupt Control Register – ICON	8-24	FF00 8510H
	Reserved		FF00 8514H to FF00 851FH
8.4.3	Interrupt Mapping Registers – IMAP0-IMAP2	8-25	FF00 8520H
8.4.3	Interrupt Mapping Registers – IMAP0-IMAP2	8-25	FF00 8524H
8.4.3	Interrupt Mapping Registers – IMAP0-IMAP2	8-25	FF00 8528H
	Reserved		FF00 852CH through FF00 85FFH
13.2	Physical Memory Control Register 0 – PMCON0_1	13-3	FF00 8600H

Table C-2. Supervisor Space Register Addresses (Sheet 2 of 2)

Section	Register Name - Acronym	Page	80960 Local Bus Address
	Reserved		FF00 8604H
13.2	Physical Memory Control Register 1 – PMCON2_3	13-3	FF00 8608H
	Reserved		FF00 860CH
13.2	Physical Memory Control Register 2 – PMCON4_5	13-3	FF00 8610H
	Reserved		FF00 8614H
13.2	Physical Memory Control Register 3 – PMCON6_7	13-3	FF00 8618H
	Reserved		FF00 861CH
13.2	Physical Memory Control Register 4 – PMCON8_9	13-3	FF00 8620H
	Reserved		FF00 8624H
13.2	Physical Memory Control Register 5 – PMCON10_11	13-3	FF00 8628H
	Reserved		FF00 862CH
13.2	Physical Memory Control Register 6 – PMCON12_13	13-3	FF00 8630H
	Reserved		FF00 8634H
12.4.1 13.2	PMCON14_15 Register Bit Description in IBR Physical Memory Control Register 7 – PMCON14_15	12-15 13-3	FF00 8638H
	Reserved		FF00 863CH through FF00 86F8H
13.3.1	Bus Control Register – BCON	13-5	FF00 86FCH
12.4.2	Process Control Block – PRCB	12-15	FF00 8700H
8.1.5	Interrupt Stack And Interrupt Record	8-5	FF00 8704H
7.6	User and Supervisor Stacks	7-16	FF00 8708H
	Reserved		FF00 870CH
12.5	Processor Device ID Register - PDIDR	12-20	FF00 8710H
12.5	i960® Core Processor Device ID Register - DEVICEID	12-20	FF00 8710H
	Reserved		FF00 8714H through FFFF FFFFH

Table C-3. Timer Registers (Sheet 1 of 2)

Section	Register Name	Page	80960 Local Bus Address
	Reserved		FF00 0000H to FF00 02FFH
19.1.3	Timer Reload Register – TRR0:1-0	19-6	FF00 0300H
19.1.2	Timer Count Register – TCR0:1-0	19-5	FF00 0304H
19.1.1	Timer Mode Register – TMR0:1-0	19-2	FF00 0308H

Table C-3. Timer Registers (Sheet 2 of 2)

Section	Register Name	Page	80960 Local Bus Address
	Reserved		FF00 030CH
19.1.3	Timer Reload Register – TRR0:1-1	19-6	FF00 0310H
19.1.2	Timer Count Register – TCR0:1-1	19-5	FF00 0314H
19.1.1	Timer Mode Register – TMR0:1-1	19-2	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 80960VH. Table C-5 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 Address Translation Unit, and I^2C Bus Interface Unit.

Portions of the 80960 processor core address space are already reserved by the 80960 processor core. 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 and the i960 core processor. Addresses 0000 1000H through 0000 17FFH are allocated to the PMMR interface.

C.4 Accessing The Peripheral Memory-Mapped Registers

The PMMR interface is a slave device connected to the 80960VH local bus. This interface accepts data transactions which appear on the local bus from the ATU and the 80960 processor core.

The PMMR interface allows these devices to perform read, write, or read-modify-write transactions. The specific actions taken when modifying any value in the PMMR space is independently defined within each chapter which describes the functionality of the register.

Note: The PMMR interface does not support multi-word burst accesses from any local bus master.

All PMMR transactions shall be allowed from the 80960 processor core operating in either user mode or supervisor mode. In addition, the PMMR shall not provide any access fault to the 80960 processor core.

The following PMMR registers have read/write access from the internal bus:

- Vendor ID Register
- Device ID Register
- Revision ID Register
- Class Code Register
- Header Type Register
- Subsystem ID Register
- 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 shall return a value of zero for registers declared as “reserved”. For PCI Configuration Write transactions, the PMMR shall discard the data. For all other types of access, reading or writing a register declared as “reserved” is undefined.

C.5 Architecturally Reserved Memory Space

The i960 VH processor provides 4 Gbytes of address space. Portions of this address space is architecturally reserved and refrained from use by the customers. Figure C-1 shows the reserved address space.

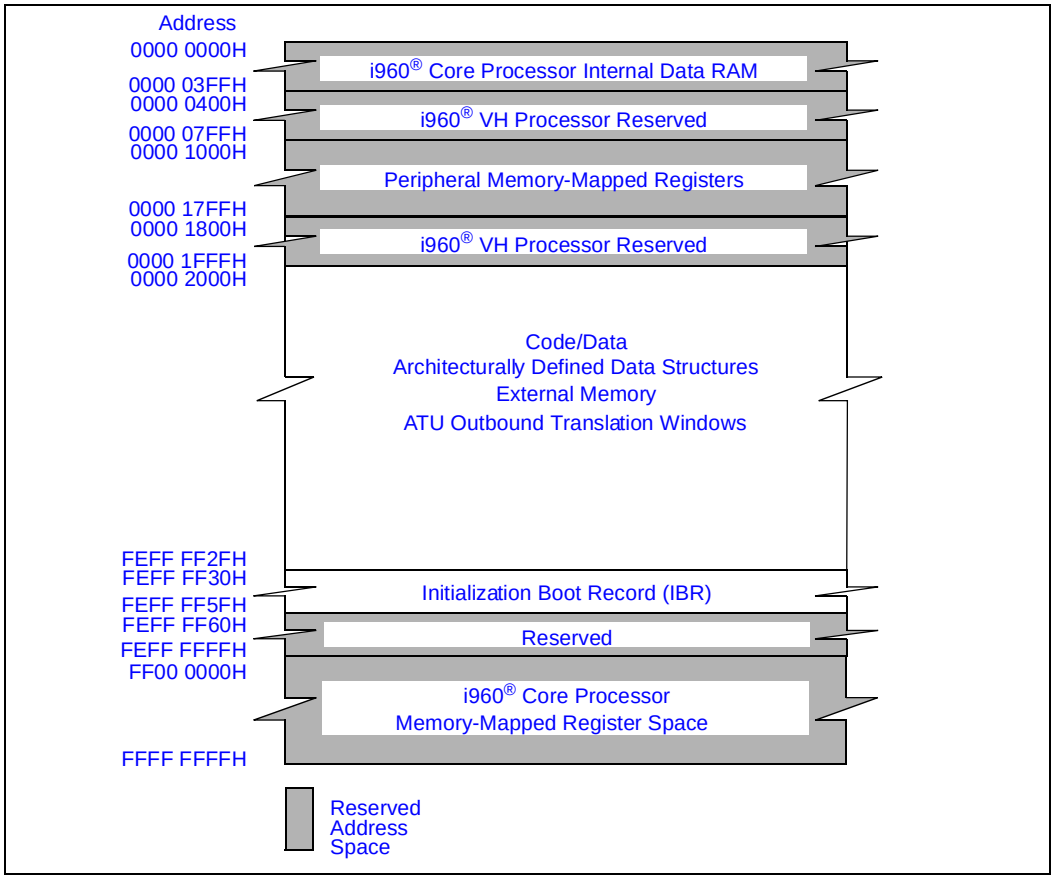
Addresses FF00 0000H through FFFF FFFFH are reserved for implementation-specific functions. This address range is termed “reserved” because future i960 architecture implementations may use these addresses for special functions such as mapped registers or data structures. Therefore, to ensure complete object level compatibility, portable code must not access or depend on values in this region.

Addresses 0000 0000H through 0000 03FFH are reserved for the internal data RAM of the 80960 processor core. This internal data RAM contains interrupt vectors plus RAM available to the application software for variable allocation or data structures. Loads and stores directed to these addresses access internal memory; instruction fetches from these addresses are not allowed for the 80960VH.

Addresses 0000 0400H through 0000 1FFFH are reserved for i960® VH processor use and should not be used by the system designer.

Addresses 0000 1000H through 0000 17FFH are allocated to the PMMR interface. These registers are reserved for i960 VH processor use and should not be written by the system designer.

Figure C-1. i960® VH processor Address Space



C.6 Peripheral Memory-Mapped Register Address Space

The PMMR address space is divided to support the integrated peripherals on the 80960VH. Table C-5 shows all of the i960® VH processor integrated peripheral memory-mapped registers and their internal bus addresses.

Table C-4. 80960 Internal Addresses Assigned to Integrated Peripherals (Sheet 1 of 2)

Integrated Peripheral	80960 Address Block
Reserved	0000 1000H through 0000 11FFH
Address Translation Unit	0000 1200H through 0000 12FFH
Core and Peripheral Control Unit	See ATU Extended Configuration Space

Table C-4. 80960 Internal Addresses Assigned to Integrated Peripherals (Sheet 2 of 2)

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
Reserved	0000 1780H through 0000 17FFH

The registers for the Address Translation Unit are accessible via PCI configuration transactions. The DMA Controllers, Memory Controller, I^2C Bus Interface Unit, Messaging Unit, Internal Arbitration Unit, and the PCI and Peripheral Interrupt Controller must have the address translation logic configured to translate PCI addresses into the 80960VH address space to access the memory-mapped registers.

Table C-5 shows all 80960VH integrated peripheral memory-mapped registers and their 80960 local bus addresses.

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 1 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
		Reserved			0000 1000H through 0000 11FFH	
Address Translation Unit	16.7.1	ATU Vendor ID Register - ATUVID	16-21	16	0000 1200H	00H
	16.7.2	ATU Device ID Register - ATUDID	16-22	16	0000 1202H	02H
	16.7.3	Primary ATU Command Register - PATUCMD	16-22	16	0000 1204H	04H
	16.7.4	Primary ATU Status Register - PATUSR	16-23	16	0000 1206H	06H
	16.7.5	ATU Revision ID Register - ATURID	16-24	8	0000 1208H	08H
	16.7.6	ATU Class Code Register - ATUCCR	16-25	24	0000 1209H	09H
	16.7.7	ATU Cacheline Size Register - ATUCLSR	16-25	8	0000 120CH	0CH
	16.7.8	ATU Latency Timer Register - ATULT	16-26	8	0000 120DH	0DH
	16.7.9	ATU Header Type Register - ATUHTR	16-26	8	0000 120EH	0EH
	16.7.10	ATU BIST Register - ATUBISTR	16-27	8	0000 120FH	0FH

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 2 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
Address Translation Unit	16.7.11	Primary Inbound ATU Base Address Register - PIABAR	16-28	32	0000 1210H	10H
		Reserved			0000 1214H through 0000 122BH	14H through 2BH
	16.7.13	ATU Subsystem Vendor ID Register - ASVIR	16-30	16	0000 122CH	2CH
	16.7.14	ATU Subsystem ID Register - ASIR	16-31	16	0000 122EH	2EH
	16.7.15	Expansion ROM Base Address Register - ERBAR	16-31	32	0000 1230H	30H
		Reserved			0000 1234H through 0000 123BH	34H through 3BH
	16.7.16	ATU Interrupt Line Register - ATUILR	16-32	8	0000 123CH	3CH
	16.7.17	ATU Interrupt Pin Register - ATUIPR	16-33	8	0000 123DH	3DH
	16.7.18	ATU Minimum Grant Register - ATUMGNT	16-34	8	0000 123EH	3EH
	16.7.19	ATU Maximum Latency Register - ATUMLAT	16-34	8	0000 123FH	3FH
	16.7.20	Primary Inbound ATU Limit Register - PIALR	16-35	32	0000 1240H	40H
	16.7.21	Primary Inbound ATU Translate Value Register - PIATVR	16-36	32	0000 1244H	44H
		Reserved			0000 1248H through 0000 1250H	48H through 50H
	16.7.22	Primary Outbound Memory Window Value Register - POMWVR	16-36	32	0000 1254H	54H
		Reserved			0000 1258H	58H
	16.7.23	Primary Outbound I/O Window Value Register - POIOWVR	16-37	32	0000 125CH	5CH

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 3 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
Address Translation Unit		Reserved			0000 1260H through 0000 1270H	60H through 70H
	16.7.24	Expansion ROM Limit Register - ERLR	16-38	32	0000 1274H	74H
	16.7.25	Expansion ROM Translate Value Register - ERTVR	16-38	32	0000 1278H	78H
		Reserved			0000 127CH through 0000 1287H	7CH through 87H
	16.7.26	ATU Configuration Register - ATUCR	16-39	32	0000 1288H	88H
		Reserved			0000 128CH	8CH
	16.7.27	Primary ATU Interrupt Status Register - PATUISR	16-40	32	0000 1290H	90H
		Reserved			0000 1294H through 0000 12A0H	94H through A0H
	16.7.28	Primary Outbound Configuration Cycle Address Register - POCCAR	16-41	32	0000 12A4H	A4H
		Reserved			0000 12A8H	A8H
	16.7.29	Primary Outbound Configuration Cycle Data Port - POCCDP	16-42	32	0000 12ACH	ACH
		Reserved			0000 12B0H through 0000 12C0H	B0H through C0H
	16.7.30	Reset/Retry Control Register - RRCR	16-42	32	0000 12C4H	C4H
	16.7.31	PCI Interrupt Routing Select Register PIRSR	16-42	32	0000 12C8H	C8H
	16.7.32	Core Select Register - CSR	16-43	32	0000 12CCH	CCH

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 4 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
Address Translation Unit		Reserved			0000 12D0H through 0000 12FFH	D0H through FFH
Messaging Unit		Reserved			0000 1300H through 0000 130CH	Available through Primary ATU Inbound Translation Window or must translate PCI address to the 80960 Memory-Mapped Address
	17.4.1	Inbound Message Registers - IMRx - 0	17-5	32	0000 1310H	
	17.4.1	Inbound Message Registers - IMRx - 1	17-5	32	0000 1314H	
	17.4.2	Outbound Message Registers - OMRx - 0	17-6	32	0000 1318H	
	17.4.2	Outbound Message Registers - OMRx - 1	17-6	32	0000 131CH	
	17.4.3	Inbound Doorbell Register - IDR	17-6	32	0000 1320H	
	17.4.4	Inbound Interrupt Status Register - IISR	17-7	32	0000 1324H	
	17.4.5	Inbound Interrupt Mask Register - IIMR	17-8	32	0000 1328H	
	17.4.6	Outbound Doorbell Register - ODR	17-9	32	0000 132CH	
	17.4.7	Outbound Interrupt Status Register - OISR	17-10	32	0000 1330H	
	17.4.8	Outbound Interrupt Mask Register - OIMR	17-11	32	0000 1334H	
		Reserved			0000 1338H through 0000 13FFH	
DMA Controller	20.7.1	Channel Control Register - CCRx - 0	20-21	32	0000 1400H	Must Translate PCI address to the 80960 Memory-Mapped Address
	20.7.2	Channel Status Register - CSRx - 0	20-22	32	0000 1404H	
		Reserved			0000 1408H	
	20.7.3	Descriptor Address Register - DARx - 0	20-24	32	0000 140CH	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 5 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
DMA Controller	20.7.4	Next Descriptor Address Register - NDARx - 0	20-24	32	0000 1410H	Must Translate PCI address to the 80960 Memory-Mapped Address
	20.7.5	PCI Address Register - PADRx - 0	20-25	32	0000 1414H	
	20.7.6	PCI Upper Address Register - PUADRx - 0	20-26	32	0000 1418H	
	20.7.7	80960 Local Address Register - LADRx - 0	20-26	32	0000 141CH	
	20.7.8	Byte Count Register - BCRx - 0	20-27	32	0000 1420H	
	20.7.9	Descriptor Control Register - DCRx - 0	20-28	32	0000 1424H	
		Reserved			0000 1428H through 0000 143FH	
	20.7.1	Channel Control Register - CCRx - 1	20-21	32	0000 1440H	
	20.7.2	Channel Status Register - CSRx - 1	20-22	32	0000 1444H	
		Reserved			0000 1448H	
	20.7.3	Descriptor Address Register - DARx - 1	20-24	32	0000 144CH	
	20.7.4	Next Descriptor Address Register - NDARx - 1	20-24	32	0000 1450H	
	20.7.5	PCI Address Register - PADRx - 1	20-25	32	0000 1454H	
	20.7.6	PCI Upper Address Register - PUADRx - 1	20-26	32	0000 1458H	
	20.7.7	80960 Local Address Register - LADRx - 1	20-26	32	0000 145CH	
	20.7.8	Byte Count Register - BCRx - 1	20-27	32	0000 1460H	
	20.7.9	Descriptor Control Register - DCRx - 1	20-28	32	0000 1464H	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 6 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
DMA Controller		Reserved			0000 1468H through 0000 14FFH	Must Translate PCI address to the 80960 Memory-Mapped Address
Memory Controller	15.5.1	Memory Bank Control Register - MBCR	15-6	32	0000 1500H	
	15.5.2	Memory Bank Base Address Registers - MBBAR0:1-0	15-8	32	0000 1504H	
	15.5.3.1	Memory Bank Read Wait State Registers - MBRWS0:1-0	15-10	32	0000 1508H	
	15.5.3.2	Memory Bank Write Wait State Registers - MBWWS0:1-0	15-11	32	0000 150CH	
Memory Controller	15.5.2	Memory Bank Base Address Registers - MBBAR0:1-0	15-8	32	0000 1510H	
	15.5.3.1	Memory Bank Read Wait State Registers - MBRWS0:1-1	15-10	32	0000 1514H	
	15.5.3.2	Memory Bank Write Wait State Registers - MBWWS0:1-1	15-11	32	0000 1518H	
	15.6.4	DRAM Bank Control Register — DBCR	15-22	32	0000 151CH	
	15.6.5	DRAM Base Address Register — DBAR	15-23	32	0000 1520H	
	15.6.6	DRAM Read Wait State Register — DRWS	15-24	32	0000 1524H	
	15.6.7	DRAM Write Wait State Register — DWWS	15-25	32	0000 1528H	
	15.6.8	DRAM Refresh Interval Register — DRIR	15-27	32	0000 152CH	
	15.7.1	DRAM Parity Enable Register — DPER	15-29	32	0000 1530H	
	15.7.2	Bus Monitor Enable Register — BMER	15-30	32	0000 1534H	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 7 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
Memory Controller	15.7.3	Memory Error Address Register — MEAR	15-31	32	0000 1538H	Must Translate PCI address to the 80960 Memory-Mapped Address
	15.7.4	Local Processor Interrupt Status Register — LPISR	15-32	32	0000 153CH	
		Reserved			0000 1504H through 0000 15FFH	
Local Bus Arbitration Unit	18.2.1	Local Bus Arbitration Control Register - LBACR	18-4	32	0000 1600H	
	18.2.6	Local Bus Arbitration Latency Counter Register – LBALCR	18-6	32	0000 1604H	
		Reserved			0000 1608H through 0000 167FH	
I^2C Bus Interface Unit	21.10.1	I2C Control Register - ICR	21-15	32	0000 1680H	
	21.10.2	I2C Status Register- ISR	21-18	32	0000 1684H	
	21.10.3	I2C Slave Address Register – ISAR	21-20	32	0000 1688H	
	21.10.4	I2C Data Buffer Register – IDBR	21-21	32	0000 168CH	
	21.10.5	I2C Clock Count Register – ICCR	21-21	32	0000 1690H	
		Reserved			0000 1694H through 0000 16FFH	
PCI And Peripheral Interrupt Controller	8.4.7	NMI Interrupt Status Register – NISR	8-30	32	0000 1700H	
	8.4.6	XINT7 Interrupt Status Register – X7ISR	8-29	32	0000 1704H	
	8.4.5	XINT6 Interrupt Status Register – X6ISR	8-29	32	0000 1708H	

Table C-5. Peripheral Memory-Mapped Register Locations (Sheet 8 of 8)

80960VH Peripheral	Section	Register Name - Acronym	Page	Size (Bits)	80960 Local Bus Address	PCI Conf Address Offset
PCI And Peripheral Interrupt Controller	8.4.1	PCI Interrupt Routing Select Register (PIRSR)	8-23	32	See ATU Configu-ration Space (0000 12C8H)	Must Translate PCI address to the 80960 Memory-Mapped Address
	12.5	Processor Device ID Register - PDIDR	12-21	32	0000 1710H	
		Reserved			0000 1714H through 0000 17FFH	

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z