PA-RISC 1.1 Architecture and Instruction Set Reference Manual

HEWLETT PACKARD

HP Part Number: 09740-90039
Printed in U.S.A. February 1994

Third Edition

Notice

Printing History

The printing date will change when a new edition is printed. The manual part number will change when extensive changes are made.

First Edition . . . . November 1990
Second Edition. September 1992
Third Edition . . . . February 1994

Contents

Figures

Tables

Preface

This manual is the Third Edition of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and it supersedes the Second Edition (published in September 1992) and the First Edition (published in November 1990). The Third Edition includes complete specifications for all the architectural enhancements defined since the Second Edition was published, in addition to all the material presented in the First and Second Editions.

From an unprivileged software perspective, this revised PA-RISC 1.1 specification is forward and backward compatible with the original PA-RISC 1.1 specification and forward compatible with the PA-RISC 1.0 specification:

- All unprivileged software written to the PA-RISC 1.0 specification or the First or Second Editions of the PA-RISC 1.1 specification will run unchanged on processors conforming to the Third Edition of the PA-RISC 1.1 specification.
- With operating system support, almost all software written to the Third Edition of the PA-RISC 1.1 specification will run unchanged on processors conforming to the First or Second Editions of the PA-RISC 1.1 specification. The only exception to this rule is that software which relies on the new PSW E-bit to access little endian data must not be executed on earlier processors.

The architectural enhancements included in the Third Edition enable higher performance and greater functionality, especially in the I/O arena:

- Mixed Endian

  An optional E-bit in the Processor Status Word enables memory references to data and instructions to have either big or little endian byte ordering. Previously, only big endian byte ordering was provided.

  The mixed endian capability enables the PA-RISC architecture to be compatible with systems which offer little endian as well as systems which provide big endian byte orderings.

- Cache Coherent I/O

  Two instructions (LOAD COHERENCE INDEX and SYNCHRONIZE DMA) have been added to enable cache coherent memory references by I/O modules. Previously, responsibility for cache coherence between the processor and I/O modules lay with software, which had to use a sequence of flush and purge operations to ensure coherence.

  While software cache coherence for I/O is still attractive in uniprocessor systems because of the lower hardware cost, hardware cache coherence for I/O has a relatively low incremental cost in multiprocessor systems.

- Uncacheable Memory

  An optional U (Uncacheable) bit has been added to each data TLB entry which controls cache move-in for the corresponding page. When the U-bit is set, new lines must not be moved in to the data cache, although existing lines may remain resident in the cache. This forces all references to non-resident lines to cause transactions to memory and enables better support of industry standard I/O busses where byte and word transactions to memory are sometimes required to communicate with I/O devices.

- Wider Protection Identifiers

  The maximum size of Protection Identifiers (PIDs) and Access Identifiers (Access IDs) has been increased to 18 bits (the minimum remains 15 bits) to better support larger multi-user systems with a very large number of processes.

- A Spatial Locality Cache Control Hint for Load and Store Instructions

  A Spatial Locality (SL) cache control hint has been added to load and store instructions. The hint is a recommendation to the processor to fetch the addressed cache line from memory, but not to displace any existing cache data, because there is good spatial locality, but poor temporal locality.

  For example, this hint might be used by software to sequentially initialize a series of small data items which will not be accessed again for a while.

- Floating-point Graphics Clip Tests

  A queue of condition bits, changes to the FLOATING-POINT COMPARE instruction, and new FLOATING-POINT TEST variants have been added to the floating-point unit to provide higher performance when doing graphics clip tests.

- Performance Monitor Coprocessor

  An optional performance monitor coprocessor has been defined to provide hardware assistance for performance analysis. Two instructions (PERFORMANCE MONITOR ENABLE and PERFORMANCE MONITOR DISABLE) have been defined to control the measurement of sections of code. Details of what is measured and how the measurement results are accessed by software are implementation dependent.

- Debug SFU

  An optional debug special function unit has been defined for Level 0 processors. The SFU consists of a set of registers and instructions which allow unprivileged software to set instruction and data breakpoints on ranges of addresses. New interruptions and PSW bits provide simple mechanisms for privileged software to manage the breakpoint traps.

Change bars have been added to the text referring to any of these architectural enhancements to assist readers familiar with the Second Edition of the PA-RISC 1.1 specification.

In addition, all known errors in the Second Edition of the PA-RISC 1.1 specification have been corrected and the text has been clarified in many places. These changes are not marked with change bars.

Conventions

Fonts

In this manual, fonts are used as follows:

Italic	is used for instruction fields and arguments. For example: "The completer, *cmplt*, encoded in the *u* and *m* fields of the instruction, ...". *Italic* is also used for references to other parts of this and other manuals. For example: "As described in *Chapter 4, Flow Control and Interruptions*, ...".
Bold	is used for emphasis and the first time a word is defined. For example: "Implementations must provide seven registers called **shadow registers** ...".
UPPER CASE	is used for instruction names, instruction mnemonics, short (three characters or less) register and register field names, and acronyms. For example: "The PL field in the IIAOQ register ...".
Underbar (_)	characters join words in register, variable, and function names. For example: "The boolean variable cond_satisfied in the Operation section ...".

Numbers

The standard notation in this document for addresses and data is hexadecimal (base 16). Memory addresses and fields within instructions are written in hexadecimal. Where numbers could be confused with decimal notation, hexadecimal numbers are preceded with 0x. For example, 0x2C is equivalent to decimal 44.

1 Overview

Introduction

PA-RISC is an extension of the architecture principles of the Reduced Instruction Set Computer (RISC). The simple design provides exceptional performance and is ideal for use in a broad family of cost-effective, compatible systems. Some typical applications include: commercial data processing, computation-intensive scientific and engineering applications, and real-time control.

Computer architectures developed in the 1960s and 1970s have evolved towards increasing system complexity. These architectures, loosely called Complex Instruction Set Computers (CISCs), have large instruction sets containing many specialized instructions. CISCs typically use microcoded control programs (i.e., microcode) to provide support for complex functions and high-level languages.

Extensive research into patterns of computer usage reveals that general-purpose computers spend up to 80% of their time executing simple instructions such as load, store, and branch. The more complex instructions are used infrequently. Unfortunately, while these complex instructions add functionality, they also add overhead for additional instruction decoding, microcode, and longer cycle times. The extra overhead reduces the performance of the simple, often-executed instructions and negates any advantages of implementing complex instructions. These findings led to the concept of the Reduced Instruction Set Computer.

PA-RISC processors implement a controlled evolution of processor architecture which is carefully designed to preserve a customer's software investment. Forward compatibility of object code is guaranteed. This allows software written for one processor to execute on any other processor without modification. The instruction set is designed to be an excellent target for optimizing compilers and is optimized for simple, often used instructions that execute in one CPU cycle. Implementation of more complex functions is assigned to system software or to assist processors such as the floating-point coprocessor. The instruction set is also very regular; all instructions are fixed-length (32-bits) and major opcodes and register fields always appear in the same locations.

The Input/Output (I/O) system is memory-mapped and accessed through load and store instructions for simplicity, flexibility, and speed. It is optimized for I/O intensive commercial data processing environments as well as for real-time control applications.

Addressing capabilities are far more powerful than those found in typical 32-bit systems. The use of 48-bit, 56-bit, or 64-bit virtual addresses is supported with full compatibility over the entire family of systems. Also supported are multiple virtual address spaces and very large data structures (up to 4 Gbytes). A powerful protection mechanism supports secure and structured operating systems.

PA-RISC is designed to support both high-performance and fault-tolerant multiprocessor systems and is an ideal platform for AI applications. The architecture can take immediate advantage of evolving hardware and software technologies with the high performance of advanced optimizing compilers.

System Features

The RISC features provided by PA-RISC include:

- Direct hardware implementation of instruction set — The instruction set can be hardwired to speed instruction execution. No microcode is needed for single cycle execution. Conventional machines require several cycles to perform even simple instructions.
- Fixed instruction size — All instructions are one word (32-bits) in length. This simplifies the instruction fetch mechanism since the location of instruction boundaries is not a function of the instruction type.
- Small number of addressing modes — The instruction set uses short displacement, long displacement and indexed modes to access memory.
- Reduced memory access — Only load and store instructions access memory. There are no computational instructions that access memory; load/store instructions operate between memory and a register. Control hardware is simplified and the machine cycle time is minimized.
- Ease of pipelining — The instructions were designed to be easily divisible into parts. This and the fixed size of the instructions allow the instructions to be easily piped.

PA-RISC provides a flexible and expandable architecture that maximizes performance from any given semiconductor technology. PA-RISC includes extensions to RISC concepts that help achieve given levels of performance at significantly lower cost than other systems.

The major extensions are summarized below:

- Very high performance cache systems and support for virtually addressed caches
- Multiprocessor systems for fault-tolerance or increased performance
- A floating-point coprocessor for IEEE floating-point operations
- A Performance Monitor Coprocessor for performance measurement
- A Debug Special Function Unit to assist in software debugging
- Extremely large and efficient virtual memory system with 48-bit, 56-bit, or 64-bit addressing
- Demand-paged memory management
- Memory access protection through a hardware Translation Lookaside Buffer (TLB)
- Memory-mapped I/O
- Optimizing compilers
- Extendable instruction set for product specific requirements

PA-RISC 1.1 Enhancements

PA-RISC 1.1 includes the following enhancements to the PA-RISC 1.0 architecture which are designed to improve performance and future extensibility:

- An optional E-bit in the Processor Status Word which enables memory references to instructions and data with big or little endian byte ordering.
- Two instructions (LOAD COHERENCE INDEX and SYNCHRONIZE DMA) to enable cache coherent memory references by I/O modules.
- An optional U-bit in the TLB entry for each data page which controls cache move-in, and can be used to provide uncached access to data in the memory address space.
- Protection Identifiers (PIDs) and Access Identifiers (Access IDs) may be up to 18 bits wide. The minimum width remains 15 bits.
- Cache control hints for quicker and more efficient memory reference instructions.
- An optional performance monitor coprocessor which provides hardware assistance for performance analysis.
- A queue of condition bits, changes to the FLOATING-POINT COMPARE instruction, and new FLOATING-POINT TEST variants provide higher performance when doing graphics clip tests.
- An optional debug special function unit which provides instruction and data breakpoint support for Level 0 processors.
- Space and offset address aliasing between virtual and physical addresses to provide better support for process forks, message passing and memory mapped files.
- Three separate traps to accelerate trap handling, in place of the combined data memory protection/unaligned data reference trap.
- An increase in the alignment of the Interrupt Vector Address table to 2 Kbytes to allow for the future definition of other interruptions.
- A change to control registers 26 and 27 to make them readable at any privilege level so that operating systems can provide user-visible per-process or per-thread identifiers.
- A Denormalized as Zero bit in the Floating-Point Status Register which is a hint to the processor that it may treat denormalized sources and/or results as zero to accelerate calculations using numbers which tend to zero.
- Two floating-point multiple-operation instructions: FLOATING-POINT MULTIPLY/ADD (FMPYADD) and FLOATING-POINT MULTIPLY/SUBTRACT (FMPYSUB).
- A fixed-point unsigned multiply instruction: FIXED-POINT MULTIPLY UNSIGNED (XMPYU).
- 16 additional floating-point registers, increasing the number of floating-point registers to 32.
- The capability to address the registers in the floating-point register file either as 64 single-precision (32-bit) floating-point registers or as 32 double-precision (64-bit) floating-point registers.
- An increase in the page size from 2 Kbytes to 4 Kbytes.
- Block TLB translations to allow the mapping of a large virtually continuous space to a continuous portion of physical memory without using several TLB entries.
- Shadow registers and a RETURN FROM INTERRUPTION AND RESTORE instruction to reduce the state save and restore time by eliminating the need for general register (GR) saves and restores in

interruption handlers.

System Organization

The PA-RISC processor is only one element of a complete system. A system also includes memory arrays, I/O adapters, and interconnecting busses. Figure 1-1 shows a typical multiprocessor system with a high-speed central bus and two connections to lower-speed busses. The processors reference main memory on the central bus and I/O adapters on the remote busses. The processors are modules on the bus and may be the target of transactions such as external interrupts and system resets.

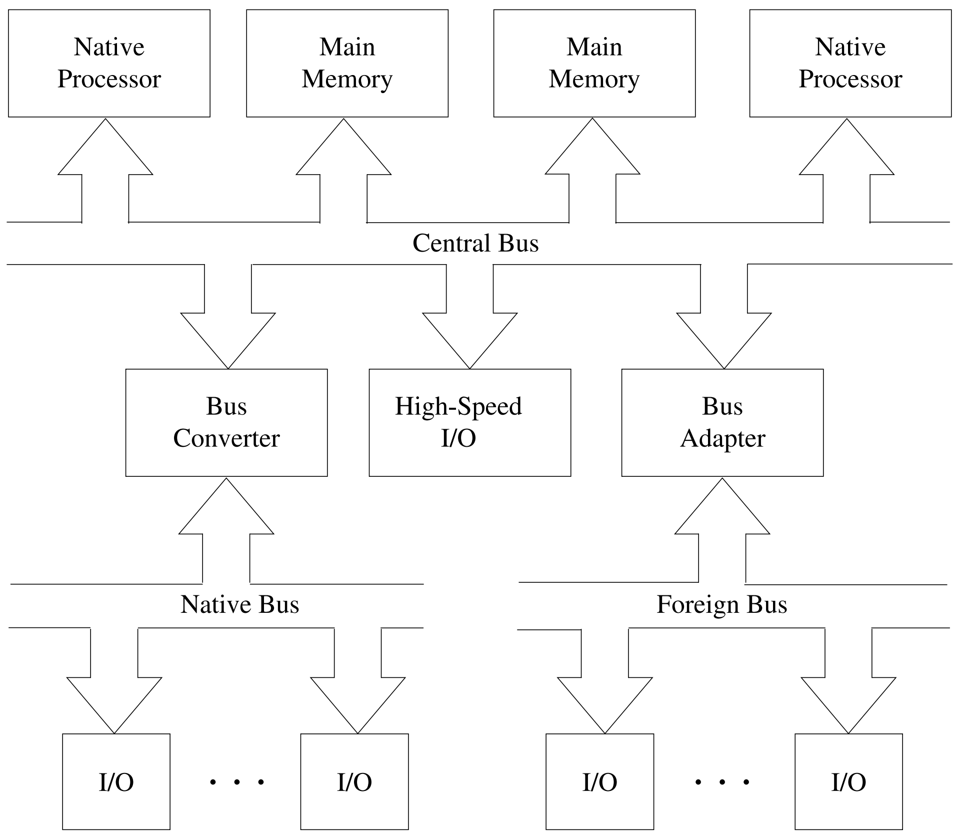


Figure 1-1. System Organization

The processor module is organized to provide a high performance computation machine. The Central Processing Unit (CPU) includes a general register set, virtual address registers and machine state registers. A cache is optional, but it is such a cost-effective component that nearly all processors incorporate this hardware. On processors that support virtual memory addressing, a hardware translation lookaside buffer (TLB) is included to provide virtual to absolute address translations.

Any processor may include Special Function Units (SFUs) and coprocessors. These dedicated hardware units substantially increase performance when executing selected hardware algorithms. Collectively, SFUs and coprocessors are called **assist processors**. Floating-point functions are provided by a coprocessor, while a signal processing algorithm could be enhanced with a specialized SFU.

I/O adapters with high bandwidth demands are connected to the higher performance central bus, while slower devices can be connected to more cost-effective remote busses.

Figure 1-2 shows a typical processor module with a cache, a TLB, one coprocessor and one SFU.

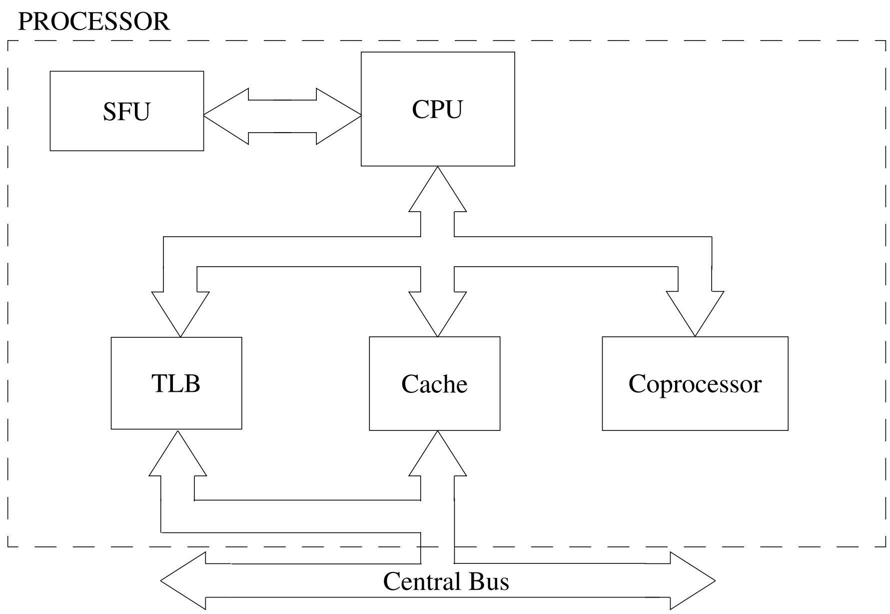


Figure 1-2. Processor Organization

Register-intensive computation is central to the architecture. Calculations are performed only between high-speed CPU registers or between registers and immediate constants. Register-intensive operation simplifies data and control paths thereby improving processor performance.

Load and store instructions are the only instructions that reference main memory. To minimize the number of memory references, optimizing compilers allocate the most frequently used variables to general-purpose registers.

Storage System

The PA-RISC storage system is an explicit hierarchy that is visible to software. The architecture provides for buffering of information to and from main memory in high-speed storage units (visible caches).

The memory hierarchy achieves nearly the speed of the highest (fastest and smallest) memory level with the capacity of the lowest (largest and slowest) memory level. The levels of this memory hierarchy from highest to lowest are the general registers, caches (if implemented), main memory and direct access storage devices such as disks.

A cache system, when implemented, is an integral part of the processor. Caches hold frequently accessed data and instructions in order to minimize access time to main memory. A system may have a separate instruction cache (I-cache) and data cache (D-cache), or may have a single, combined cache that holds both instructions and data.

In systems which support virtual addressing, to perform translations from virtual addresses to absolute addresses, a hardware feature called the Translation Lookaside Buffer (TLB) is included. The TLB contains translations for recently accessed virtual pages. Each TLB entry also contains information used to determine valid access to that memory page and the type of access permitted. While the TLB determines the proper translation of the virtual address, access information is checked and access is either granted or denied. TLBs may be split on a processor, one for instructions (ITLB) and one for data (DTLB).

Virtual Addressing

A generalized virtual memory system is an integral part of the architecture on all but the smallest PA-RISC systems. The virtual memory system supports 48-bit, 56-bit, or 64-bit virtual addresses. Program-supplied addresses are treated as logical addresses and translated to absolute addresses by the TLB when memory is referenced. Address translations are made at the page level. In systems without virtual addressing, the absolute address and virtual address are the same. Direct access to physical memory locations is also supported in the instruction set.

The global virtual memory is organized as a set of linear spaces with each space being 4 Gbytes (2^{32} bytes) long. Each space is specified with a space identifier and divided into fixed-length 4 Kbyte pages.

Data Types

PA-RISC supports the following data types:

- 8-bit ASCII characters (values 0 through 127)
- HP's 8-bit extended Roman-8 characters (values 128 through 255)
- Signed and unsigned 16-bit integers
- Signed and unsigned 32-bit integers
- Unsigned 64-bit integers
- Packed decimal; 7, 15, 23, or 31 BCD (Binary Coded Decimal) digits
- Unpacked decimal; one or more bytes
- Single-word (32-bit) IEEE floating-point
- Double-word (64-bit) IEEE floating-point
- Quadruple-word (128-bit) IEEE floating-point

Instruction Set

There are two primary addressing modes for memory accesses: base relative and indexed. Memory references can be specified by either virtual or absolute addressing.

Memory Reference Instructions transfer data between the general registers and main memory or the I/O system. Load and store instructions are the only instructions that reference memory. Operands required for a given operation are first brought into a CPU register from memory with a load instruction. The result of the operation is explicitly saved to memory with a store instruction.

Instructions access system I/O in a similar way to main memory. System I/O is memory-mapped such that I/O modules are mapped into physical pages which are not part of the main memory, but which are addressed in the same way. This provides the same flexibility, security, and protection mechanisms provided for main memory.

Arithmetic and logical instructions provide a simple but powerful set of functions. Besides the usual arithmetic and logical operations, there are shift-and-add instructions to accelerate integer multiplication, extract and deposit instructions for bit manipulations, and several instructions to provide support for packed and unpacked decimal arithmetic.

Multiple-precision arithmetic is supported with carry-sensitive instructions. More complex arithmetic functions (including packed, unpacked and zoned decimal operations) are supported by language compilers through execution of a sequence of simple instructions.

The control flow of a program is affected by branch instructions and by instructions that skip the following instruction. The condition resulting from an operation can immediately determine whether or not a branch should be taken. Unconditional branch and procedure call instructions are provided to alter control flow. The need for some branch sequences is eliminated as most computational instructions can specify skipping of the next instruction. This permits such common functions as range checking to be performed in a simple, non-branching instruction sequence.

Floating-point instructions support the defined IEEE standard operations of addition, subtraction, multiplication, division, square root, conversions, and round-to-integer.

System control instructions provide the support needed to implement an operating system including: returning from interruptions, executing instruction breaks and probing access rights. They also control the Processor Status Word, special registers, caches, and translation lookaside buffers.

Input/Output Organization

The PA-RISC I/O architecture is **memory-mapped**, which means that complete control of all attached modules is exercised by the execution of memory read and write commands. Processors invoke these operations by executing load and store instructions to either virtual or absolute addresses.

This approach permits I/O drivers to be written in high-level languages. Since the usual page-level protection mechanism is applied during virtual-to-absolute address translation, user programs can be granted direct control over particular I/O modules without compromising system integrity.

Direct I/O is the simplest and least costly type of system I/O interface because it has little or no local state and is controlled entirely by software. Since direct I/O responds only to load and store instructions and never generates memory addresses, it may be mapped into virtual space and controlled directly by user programs.

Direct Memory Access (DMA) I/O adapters contain sufficient state to control the transfer of data to or from a contiguous range of absolute addresses and to perform data chaining. This state is initialized prior to the start of a transfer by a privileged driver which is responsible for the mapping and validation of virtual addresses. During the transfer, the virtual page(s) involved must be locked in physical memory and protected from conflicting accesses through software.

Assist Processors

Assist processors are hardware units that can be added to the basic PA-RISC system to enhance its performance or functionality. Two categories of assist processors are defined and are distinguished by the level at which they interface with the memory hierarchy.

The first type of assist processor is the **special function unit** (SFU) which interfaces to the memory hierarchy at the general register level. This acts as an alternate ALU or as an alternate path through the execution unit of the main processor. It may have its own internal state.

The second type of assist processor is the **coprocessor**, which shares the main processor caches. Coprocessors are typically used to enhance performance of special operations such as high-performance floating-point calculations. Coprocessors generally have their own internal state and hardware evaluation mechanism.

Multiprocessor Systems

Multiprocessor support for various types of multiprocessor systems is built into the architecture. Multiprocessors can be configured to provide incremental performance improvement via distribution of the system workload over multiple CPUs, or can be configured redundantly to provide fault-tolerance in the system. In systems sharing a single virtual address space, the architecture defines a model of a single consistent cache and TLB. Software is still responsible for maintaining coherence for modifying instructions, and for virtual address mapping. Systems may choose to only share physical memory and form more loosely-coupled configurations. All multiprocessor systems synchronize using a semaphore lock in shared main memory.

2 System Organization

Introduction

The PA-RISC instruction set is only one aspect of the processor architecture; the following components are also specified:

- Memory and I/O Addressing — how system memory and the input and output facilities are organized and accessed
- Data Types — how data is organized and what data types are available to the user
- Processing Resources — what registers and register sets are available to the user

Data storage is organized as a storage hierarchy with user-accessible registers as the highest level. This is followed by the memory system which consists of high-speed buffers that hold recently referenced instructions and/or data, and main memory. These buffers, called **instruction** and/or **data caches**, reduce the effective access time to main memory.

The I/O system is memory-mapped. I/O modules are mapped into physical pages that are not part of the main memory, but are addressed in the same way. With virtual pages mapped into physical pages and I/O registers represented by words in a page, communication between a processor and an I/O module can be performed with load and store instructions to virtual addresses. The privilege level and access rights of such a page provide versatile protection. Non-privileged code may therefore be given direct access to some I/O modules without compromising system security.

The software-accessible registers (i.e., the processing resources) are the storage elements within a processor that are manipulated by the instructions. These resources participate in instruction control flow, computations, interruption processing, protection mechanisms, and virtual memory management. The processing resources available to software are listed below:

- General Registers (GR 0..GR 31)
- Shadow Registers (SHR 0..SHR 6)
- Space Registers (SR 0..SR 7)
- Processor Status Word (PSW)
- Instruction Address Queues
- Control Registers (CR 0..CR 31)
- Special Function Unit Registers
- Coprocessor Registers
- Floating-point Registers (FPR 0..FPR 31)

All of these resources are described in this chapter, with the exception of the floating-point registers which are described in Chapter 6, “Floating-point Coprocessor”.

Memory and I/O Addressing

Objects in the main memory and I/O system are addressed using 32-bit absolute addresses. An absolute address is a 32-bit unsigned integer whose value is the address of the lowest-addressed byte of the operand it designates (see Figure 2-1).

Absolute Byte Address
32

Figure 2-1. Absolute Pointer

Figure 2-2 illustrates the relationship of the I/O address space to the main memory address space. Addresses 0 through 0xEFFFFFFF reference memory. Addresses 0xF0000000 through 0xFFFFFFFF reference I/O registers. This structure gives nearly 4 Gbytes of memory address space and 256 Mbytes of I/O address space.

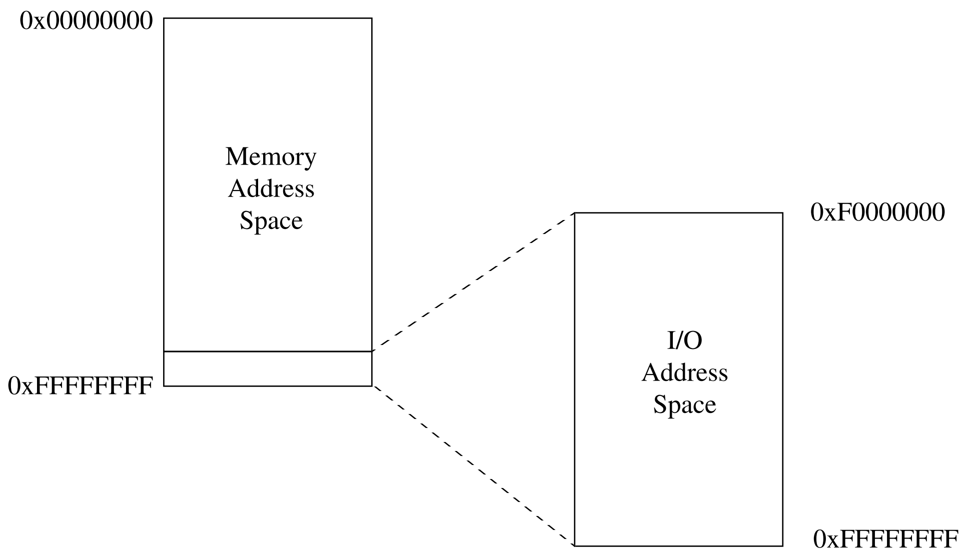


Figure 2-2. Memory and I/O Addresses

Memory is always referenced with byte addresses, starting with address 0 and extending through the largest defined non-I/O address (0xEFFFFFFF). Addressable units are bytes, halfwords (2 bytes), words (4 bytes), and doublewords (8 bytes). A comparison of the addressable units is shown in Figure 2-3 with the relative byte numbers indicated inside the blocks.

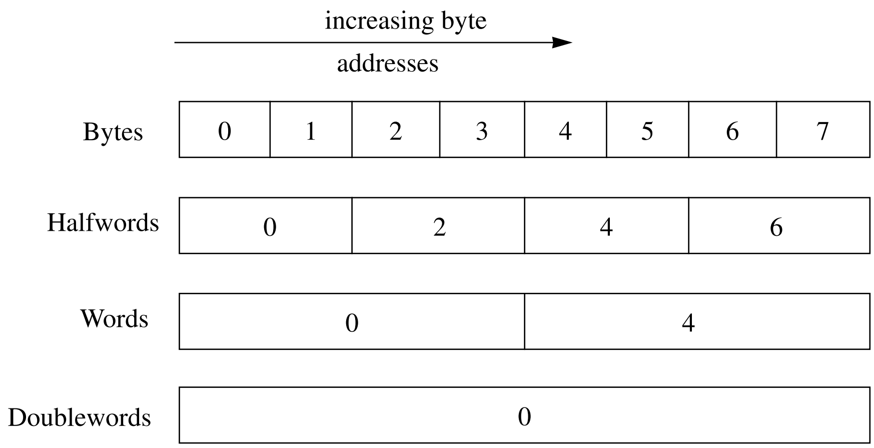


Figure 2-3. Physical Memory Addressing and Storage Units

All addressable units must be stored on their naturally aligned boundaries. A byte may appear at any address, halfwords must begin at even addresses, words must begin at addresses that are multiples of 4, and doublewords begin at addresses that are multiples of 8. If an unaligned virtual address is used, an interruption occurs.

Bits within larger units are always numbered from 0 starting with the most significant bit.

I/O address space is referenced in words, halfwords, and bytes. I/O registers are accessed using the normal load and store instructions.

Virtual memory is organized into linear spaces of 2^{32} = 4,294,967,296 bytes each. Each space is designated by a space identifier or **space ID**. The object within the space is specified by a 32-bit **offset**. The concatenation of a space identifier and this offset forms a complete virtual address.

Translation from virtual to absolute addresses is accomplished by translation lookaside buffers (TLBs), which are described in Chapter 3, “Addressing and Access Control”. Fields in the TLB entry for a particular page permit control of access to the page for reading, writing or execution. Such access may be restricted to a single process, or a set of processes, or may be permitted to all processes.

Byte Ordering (Big Endian/Little Endian)

The optional E-bit in the PSW controls whether loads and stores use big endian or little endian byte ordering. When the E-bit is 0, all larger-than-byte loads and stores are big endian — the lower-addressed bytes in memory correspond to the higher-order bytes in the register. When the E-bit is 1, all larger-than-byte loads and stores are little endian — the lower-addressed bytes in memory correspond to the lower-order bytes in the register. Load byte and store byte instructions are not affected by the E-bit. The E-bit also affects instruction fetch.

Processors which implement the PSW E-bit must also provide an implementation-dependent, software writable default endian bit. The default endian bit controls whether the PSW E-bit is set to 0 or 1 on interruptions and also controls whether data in the page table is interpreted in big endian or little endian format by the hardware TLB miss handler (if implemented).

Figure 2-4 shows various loads in big endian format. Figure 2-5 shows various loads in little endian format. Stores are not shown but behave similarly.

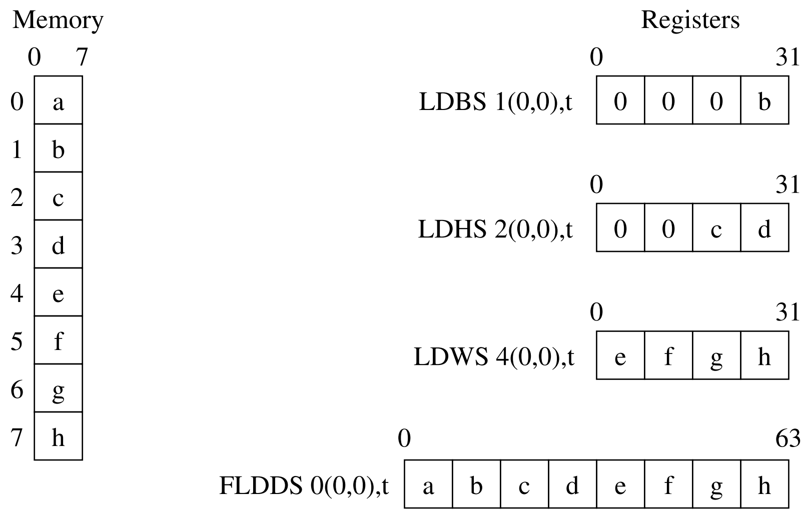


Figure 2-4. Big Endian Loads

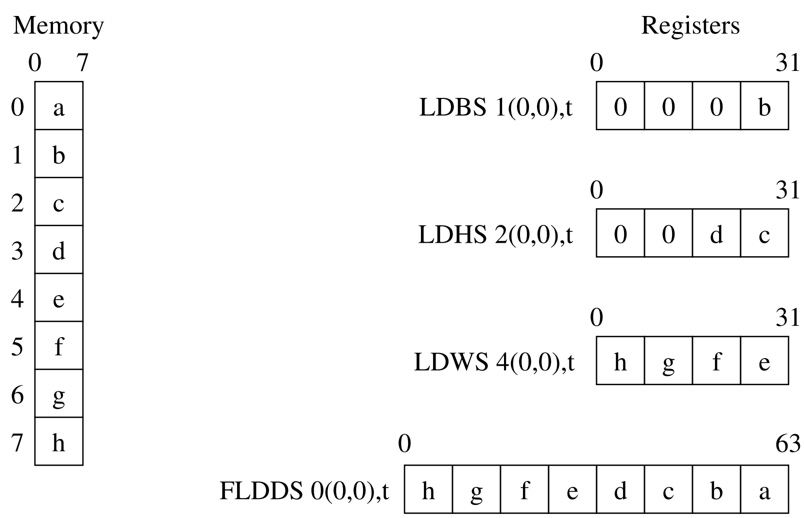


Figure 2-5. Little Endian Loads

The E-bit also affects instruction fetch. When the E-bit is 0, instruction fetch is big endian — the lower-addressed bytes in memory correspond to the higher-order bytes in the instruction. When the E-bit is 1, instruction fetch is little endian — the lower-addressed bytes in memory correspond to the lower-order bytes in the instruction.

Architecturally, the instruction byte swapping can occur either when a cache line is moved into the instruction cache or as instructions are fetched from the I-cache into the pipeline.

Because processors are allowed to swap instructions as they are moved into the I-cache, software is required to keep track of which pages might have been brought into the I-cache in big endian form and in little endian form, given the cache move-in rules, and before executing the code, flush all lines on any page that might have been moved in with the wrong form. Note that the move-in rules allow all lines on a page, plus the next sequential page, to be moved in, so guard pages (that will never be executed) must be used between code pages which will execute with opposite endian form.

Levels of PA-RISC

Four **levels** of the processor architecture have been defined: **0**, **1**, **1.5**, and **2**. Level 0 systems support absolute memory addressing only; virtual memory is not supported, and so space identifiers are not used. Level 1, 1.5, and 2 systems have virtual addressing and differ only in the number of significant bits in their space identifiers. They have 2^{16}, 2^{24}, and 2^{32} virtual spaces, respectively. To provide for growth to larger systems, each higher level processor has a superset of the capabilities of the lower level processors.

Data Types

The fundamental data types that are recognized are bits, bytes, integers, floating-point numbers, and decimal numbers. Their formats are described briefly in this section. Each item of data is addressed by its lowest-numbered byte.

Bits — Memory is not addressed to the resolution of bits; however, efficient support is provided to manipulate and test individual bits in the general registers.

Bytes — Bytes are signed or unsigned 8-bit quantities:

Signed Byte

s	value
1	7

Unsigned Byte

value
8

Bytes are packed four to a word and may represent a signed value in the range -128 through +127, an unsigned value in the range 0 through 255, an arbitrary collection of eight bits, or an ASCII character.

The character codes conform to the ASCII standard for byte values in the range 0 through 127 and to HP's 8-bit extended Roman-8 character set for byte values in the range 128 through 255.

Integers

Integers may be 16 or 32 bits wide, signed or unsigned, or 64 bits wide, unsigned only:

Signed Halfword

s	value
1	15

Unsigned Halfword

value
16

Signed Word

s	value
1	31

Unsigned Word

value
32

Unsigned Doubleword

value (high word)
32

value (low word)
32

Signed integers are in two's complement form. Halfword integers can be stored in memory only at even byte addresses, word integers only at addresses evenly divisible by four, and doubleword integers only at addresses evenly divisible by eight.

Floating-Point Numbers

The binary floating-point number representation conforms to the ANSI/IEEE 754-1985 standards. Single-word (32-bit), double-word (64-bit), and quadruple-word (128-bit) binary formats are supported.

Single-precision floating-point numbers must be aligned on word boundaries. Double-precision and quad-precision numbers must be aligned on doubleword boundaries. See Chapter 6, "Floating-point Coprocessor", for detailed information on the floating-point formats.

Packed Decimal Numbers

Packed decimal data is always aligned on a word boundary. It consists of 7, 15, 23, or 31 BCD digits, each four bits wide and having a value in the range of 0x0 to 0x9, followed by a 4-bit sign as shown in the following figure:

MSD		• • •	LSD	sign
4	4		4	4

The standard sign for a positive number is 0xC, but any value except 0xD will be interpreted as positive. 0xD indicates a minus sign for a negative number. 0xB is not supported as an alternative minus sign.

Processing Resources

The architecture provides registers, state information, and protocols for computation, addressing, and control of execution and interruptions. Some of these resources are described below.

Unused Registers and Bits

Currently, there are several registers and bit-fields within registers that do not have any function assigned to them. All such processing resources are classified into five categories:

1. Reserved bits — Currently unused bits, but reserved for possible future use. A READ operation is legal, and the value read back is all zeros. A WRITE operation is legal but the value written must be all zeros. Writing ones is an undefined operation. (For example, writing ones may cause these bits to no longer read as zeros.)
2. Nonexistent bits — Architecturally these bits do not exist. A READ operation is legal and may return zeros or what was last written. A WRITE operation is also legal, but does not have any effect on system functionality.
3. Undefined bits — Architecturally these bits are undefined. A READ operation is legal and the value read is undefined. A WRITE operation is also legal, but does not have any effect on system functionality.
4. Reserved registers — A register that is numbered but currently unused. Both READ and WRITE operations are undefined operations.
5. Nonexistent registers — A register that does not exist in Level 0 systems. A READ operation returns zeros. A WRITE operation has no effect (executes as a null instruction).

General Registers

Thirty-two 32-bit **general registers** provide the central resource for all computation (Figure 2-6). They are numbered GR 0 through GR 31, and are available to all programs at all privilege levels.

GR 0, GR 1, and GR 31 have special functions. GR 0, when referenced as a source operand, delivers zeros. When GR 0 is used as a destination, the result is discarded. GR 1 is the implied target of the ADD IMMEDIATE LEFT instruction. GR 31 is the instruction address offset link register for the base-relative interspace procedure call instruction (BRANCH AND LINK EXTERNAL). GR 1 and GR 31 can also be used as general registers; however, software conventions may at times restrict their use.

	0 31
GR 0	Permanent zero
GR 1	Target for ADDIL or General use
GR 2	General use
	•
	•
	•
GR 30	General use
GR 31	Link register for BLE or General use

Figure 2-6. General Registers

Shadow Registers

Implementations must provide seven registers called **shadow registers**, numbered SHR 0 through SHR 6, into which the contents of GRs 1, 8, 9, 16, 17, 24, and 25 are copied upon interruptions. The contents of these general registers are restored from their shadow registers when a RETURN FROM INTERRUPTION AND RESTORE instruction is executed.

Space Registers

In systems which support virtual memory, eight **space registers**, numbered SR 0 through SR 7, contain space identifiers for virtual addressing. Instructions specify space registers either directly in the instruction or indirectly through general register contents.

Instruction addresses, computed by branch instructions, may use any of the space registers. SR 0 is the instruction address space link register for the base-relative interspace procedure call instruction (BRANCH AND LINK EXTERNAL). Data operands can specify SR 1 through SR 3 explicitly, and SR 4 through SR 7 indirectly, via general registers.

SR 1 through SR 7 have no special functions; however, their use will normally be constrained by software conventions. For example, the following convention supports non-overlapping process groups. SR 1 through SR 3 provide general-use virtual pointers. SR 4 tracks the instruction address (IA) space and provides access to literal data contained in the current code segment. SR 5 points to a space containing process private data, SR 6 to a space containing data shared by a group of processes, and SR 7 to a space containing the operating system’s public code, literals, and data. Figure 2-7 illustrates this convention.

SRs 5 through 7 can be modified only by code executing at the most privileged level.

SR 0	Link code space ID
SR 1	General use
SR 2	General use
SR 3	General use
SR 4	Tracks IA space
SR 5	Process private data
SR 6	Shared data
SR 7	Operating system’s public code, literals, and data

Figure 2-7. Space Registers

Space registers, as well as IASQ, IIASQ, and ISR which are described later, may be nonexistent, 16, 24, or 32 bits wide, as shown in Figure 2-8.

	0	7	15	23	31
Level 0	nonexistent register				
Level 1	nonexistent bits		defined		
Level 1.5	nonexistent bits	defined			
Level 2	defined				

Figure 2-8. Width of SRs, IASQ, IIASQ, and ISR in Different Levels

Processor Status Word (PSW)

Processor state is encoded in a 32-bit register called the Processor Status Word (PSW). When an interruption occurs, the old value of the PSW is saved in the Interruption Processor Status Word (IPSW) and usually all defined PSW bits are set to 0. The format of the PSW is shown in Figure 2-9.

0	1	2–4	5	6	7	8	9	10	11	12	13	14	15	16–23	24	25	26	27	28	29	30	31
Y	Z	rv	E	S	T	H	L	N	X	B	C	V	M	C/B	rv	G	F	R	Q	P	D	I

Figure 2-9. Processor Status Word

The PSW is set to the contents of the IPSW by the RETURN FROM INTERRUPTION and RETURN FROM INTERRUPTION AND RESTORE instructions. The interruption handler may restore the original PSW, modify selected bits, or may change the PSW to an entirely new value.

The G, F, R, Q, P, D, and I bits of the PSW are known as the system mask. Each of these bits, with the exception of the Q-bit, may be set to 1, set to 0, written, and read by the system control instructions that manipulate the system mask. The Q-bit is specially defined. It can be set to 0 by system control instructions that manipulate the system mask, but setting it to 1 when the current value is 0 is an undefined operation. The only instructions that can set the Q-bit to 1 are the RETURN FROM INTERRUPTION and RETURN FROM INTERRUPTION AND RESTORE instructions.

Some of the PSW bits are termed **mask/unmask** bits whereas others are termed **disable/enable** bits. Interruptions that are masked remain pending whereas those that are disabled are ignored.

The PSW fields are described below:

Field	**Description**
rv	Reserved bits.
Y	Data debug trap disable. The Y-bit is set to 0 after the execution of each instruction, except for the RETURN FROM INTERRUPTION and RETURN FROM INTERRUPTION AND RESTORE instructions which may set it to 1. When 1, data debug traps are disabled. This bit allows a simple mechanism to trap on a data access and then proceed past the trapping instruction. Implementation of this bit is required only if the data debug trap is implemented. If it is not implemented, this bit is a reserved bit.
Z	Instruction debug trap disable. The Z-bit is set to 0 after the execution of each instruction, except for the RETURN FROM INTERRUPTION and RETURN FROM INTERRUPTION AND RESTORE instructions which may set it to 1. When 1, instruction debug traps are disabled. This bit allows a simple mechanism to trap on an instruction access and then proceed past the trapping instruction. Implementation of this bit is required only if the instruction debug trap is implemented. If it is not implemented, this bit is a reserved bit.
E	Little endian memory access enable. When 0, all memory references are big endian. When 1, all memory references are little endian. Implementation of this bit is optional. If it is not implemented, all memory references are big endian and this bit is a reserved bit.
S	Secure Interval Timer. When 1, the Interval Timer is readable only by code executing at the most privileged level. When 0, the Interval Timer is readable by code executing at any privilege level.
T	Taken branch trap enable. When 1, any taken branch is terminated with a taken branch trap.
H	Higher-privilege transfer trap enable. When 1, a higher privilege transfer trap occurs whenever the following instruction is of a higher privilege.
L	Lower-privilege transfer trap enable. When 1, a lower privilege transfer trap occurs whenever the following instruction is of a lower privilege.
N	Nullify. The current instruction is nullified when this bit is 1. This bit is set to 1 by an instruction that nullifies the following instruction.
X	Data memory break disable. The X-bit is set to 0 after the execution of each instruction, except for the RETURN FROM INTERRUPTION and RETURN FROM INTERRUPTION AND RESTORE instructions which may set it to 1. When 1, data memory break traps are disabled. This bit allows a simple mechanism to trap on all

data stores and proceed past them.

B — Taken branch. The B-bit is set to 1 by any taken branch instruction and set to 0 otherwise. This is used to ensure that the privilege increasing instruction does not compromise system security.

C — Code (instruction) address translation enable. When 1, instruction addresses are translated and access rights checked.

V — Divide step correction. The integer division primitive instruction records intermediate status in this bit to provide a non-restoring divide primitive.

M — High-priority machine check mask. When 1, High Priority Machine Checks (HPMCs) are masked. Normally 0, this bit is set to 1 after an HPMC and set to 0 after all other interruptions.

C/B — Carry/borrow bits. The following instructions update the PSW carry/borrow bits from the corresponding carry/borrow outputs of the 4-bit digits of the ALU:

ADDIT	ADDI	SUBI	SUB
ADDITO*	ADDIO*	SUBIO*	SUBO*
ADD	SH1ADD	SH2ADD	SH3ADD
ADDO*	SH1ADDO*	SH2ADDO*	SH3ADDO*
ADDC	SUBB	SUBT	DS
ADDCO*	SUBBO*	SUBTO*	

The instructions marked with an asterisk set the carry/borrow bits only if the instruction does not cause an overflow trap.

After an add which sets them, each bit is set to 1 if a carry occurred out of its corresponding digit, and set to 0 otherwise. After a subtract which sets them, each bit is set to 0 if a borrow occurred into its corresponding digit, and set to 1 otherwise.

G — Debug trap enable. When 1, the data debug trap and the instruction debug trap are enabled and can cause an interruption. When 0, the traps are disabled. If the debug SFU is not implemented, this bit is a reserved bit.

F — Performance monitor interrupt unmask. When 1, the performance monitor interrupt is unmasked and can cause an interruption. When 0, the interruption is held pending. If the performance monitor is not implemented or never interrupts, this bit is a reserved bit.

R — Recovery Counter enable. When 1, recovery counter traps occur if bit 0 of the recovery counter is a 1. This bit also enables decrementing of the recovery counter.

Q — Interruption state collection enable. When 1, interruption state is collected. Used in processing the interruption and returning to the interrupted code, this state is recorded in the Interruption Instruction Address Queue (IIAQ), the Interruption Instruction Register (IIR), the Interruption Space Register (ISR), and the Interruption Offset Register (IOR).

P Protection identifier validation enable. When this bit and the C-bit are both equal to 1, instruction references check for valid protection identifiers (PIDs). When this bit and the D-bit are both equal to 1, data references check for valid PIDs. When this bit is 1, probe instructions check for valid PIDs.

D Data address translation enable. When 1, data addresses are translated and access rights checked.

I External interrupt, power failure interrupt, and low-priority machine check interruption unmask. When 1, these interruptions are unmasked and can cause an interruption. When 0, the interruptions are held pending.

In Level 0 systems, the X, C, P, and D bits are nonexistent bits. In non-Level 0 systems, the Y, Z, and G bits are reserved bits.

Instruction Address Queues

The Instruction Address Queues hold the instruction address of the currently executing instruction and the address of the instruction that will be executed after the current instruction, termed the **following** instruction. Note that the following instruction is not necessarily the next instruction in the linear code space. These two queues are each two elements deep. The Instruction Address Offset Queue (IAOQ) elements are each 32 bits wide. The high-order 30 bits contain the word offset of the instruction while the 2 low-order bits maintain the **privilege level** of the corresponding instruction. There are four privilege levels: 0, 1, 2, and 3 with 0 being the most privileged level. In Level 0 systems, there are only two distinct privilege levels - 0 and nonzero; privilege levels 1, 2, and 3 are equivalent.

The Instruction Address Space Queue (IASQ) contains the space ID of the current and following instructions. The IASQ may be nonexistent, 16, 24, or 32 bits wide, as shown in Figure 2-8 on page 2-9. The space ID of the current instruction, when executing without instruction address translation enabled, is not specified and may contain any value.

The front elements of the two queues (IASQ_Front and IAOQ_Front) form the virtual address of the current instruction while the back elements of the two queues (IASQ_Back and IAOQ_Back) contain the address of the following instruction. Figure 2-10 shows this structure. Two addresses are maintained to support the delayed branching capability.

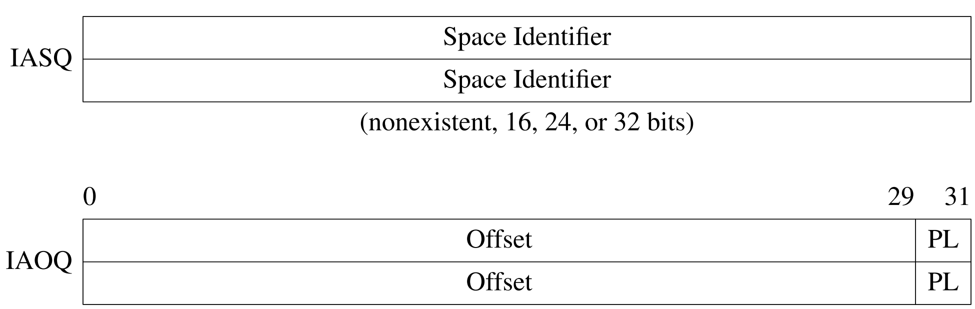


Figure 2-10. Instruction Address Queues

Control Registers

There are twenty-five defined **control registers**, numbered CR 0, and CR 8 through CR 31, which contain system state information.

CR 11, the Shift Amount Register, is readable and writable by code executing at any privilege level. CR 16, the Interval Timer, is readable and writable only by privileged software, unless the PSW S-bit is 0, in which case it is readable by code executing at any privilege level. CR 26 and CR 27, two of the temporary registers, are readable by code executing at any privilege level and writable only by code executing at the most privileged level. All other defined control registers are accessible only by code executing at the most privileged level.

The control registers are shown in Figure 2-11 and described in the following sections. Moving from control registers into general registers copies the register right aligned into the general register. Moving to control registers from general registers copies the entire general register into the control register.

Control registers 1 through 7 are reserved registers, and the unused bit positions of the PIDs and the Coprocessor Configuration Register are reserved bits. The unused bits of the Shift Amount Register are nonexistent bits.

In Level 0 systems, CRs 8, 9, 12, 13, 17, and 20 are nonexistent registers.

Recovery Counter

The Recovery Counter (CR 0) is a 32-bit counter that can be used to provide software recovery of hardware faults in fault-tolerant systems, and can also be used for debugging purposes. CR 0 counts down by 1 during the execution of each non-nullified instruction for which the PSW R-bit is 1. The recovery counter is restored if the instruction terminates with a group 1, 2, or 3 interruption (see Chapter 4, “Flow Control and Interruptions”). When the leftmost bit of the Recovery Counter is 1, a recovery counter trap occurs. The trap and the decrement operation can be disabled by setting the PSW R-bit to 0. The value of the Recovery Counter may be read reliably only when the PSW R-bit is 0. The Recovery Counter may be written reliably only when the PSW R-bit is 0. Otherwise, writing the Recovery Counter is an undefined operation. If the PSW R-bit is set to 0 by either the RESET SYSTEM MASK or the MOVE TO SYSTEM MASK instruction, the recovery counter may not be read or written reliably prior to the execution of the eighth instruction after the RESET SYSTEM MASK or the MOVE TO SYSTEM MASK instruction. An interruption, or a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction which sets the PSW R-bit to 0, does not have this restriction.

Protection Identifiers

The protection identifiers (CRs 8, 9, 12, 13) designate up to four groups of pages which are accessible to the currently executing process. When translation is enabled, the four protection identifiers (PIDs) are compared with a page access identifier in the TLB entry to validate an access. The rightmost bit of each of the four PIDs is the write disable (WD) bit. When the WD-bit is 1, that PID cannot be used to grant write access. This allows each process sharing memory to have different access rights to the memory without the overhead of changing the access identifier and access rights in the TLB. When the PSW P-bit is 0, the PIDs, including the WD-bits, are ignored.

Register	0 … 31	Size
CR 0	Recovery Counter	(32 bits)
	reserved	
CR 8	reserved | Protection ID 1 | WD	(16-19 bits)
CR 9	reserved | Protection ID 2 | WD	(16-19 bits)
CR 10	reserved | SCR | CCR	(16 bits)
CR 11	nonexistent | SAR	(5 bits)
CR 12	reserved | Protection ID 3 | WD	(16-19 bits)
CR 13	reserved | Protection ID 4 | WD	(16-19 bits)
CR 14	Interruption Vector Address | reserved	(21 bits)
CR 15	External Interrupt Enable Mask	(32 bits)
CR 16	Interval Timer	(32 bits)
CR 17	Interruption Instruction Address Space Queue	(16, 24, or 32 bits)
CR 18	Interruption Instruction Address Offset Queue	(32 bits)
CR 19	Interruption Instruction Register	(32 bits)
CR 20	Interruption Space Register	(16, 24, or 32 bits)
CR 21	Interruption Offset Register	(32 bits)
CR 22	Interruption Processor Status Word	(32 bits)
CR 23	External Interrupt Request Register	(32 bits)
CR 24	Temporary Register	(32 bits)
	• • •	
CR 31	Temporary Register	(32 bits)

Figure 2-11. Control Registers

The PID registers are defined to be from 16 to 19 bit registers (including the WD bit), with the remaining bits being reserved bits. The length of the PID registers is implementation-dependent. In Level 0 systems, CRs 8, 9, 12, and 13 are nonexistent registers.

Coprocessor Configuration Register

The Coprocessor Configuration Register or CCR (bits 24..31 of CR 10), is an 8-bit register which records the presence and usability of coprocessors. The bit positions are numbered 0 through 7, and correspond to a coprocessor with the same unit identifier. Bits 0 and 1 correspond to the floating-point coprocessor, and bit 2 corresponds to the performance monitor coprocessor. Bit 7 is the rightmost bit of the CCR. It receives bit 31 from a general register when a general register is written to CR 10. The upper 16 bits of CR 10, and bits within the CCR corresponding to coprocessors which are not present, are reserved bits.

The behavior of the floating-point coprocessor with respect to the state of CCR bits 0 and 1 and the behavior of the performance monitor coprocessor with respect to the state of CCR bit 2, are specified in “Coprocessor Instructions” on page 5-178. For other coprocessors, setting a bit in the CCR to 1 enables the use of the corresponding coprocessor, if present and operational. If a CCR bit is 0, the corresponding coprocessor, if present, is logically decoupled. This decoupling must ensure that the state of a coprocessor does not change as long as its corresponding CCR bit is 0. When a CCR bit is set to 0 and an attempt is made to execute an instruction which references the corresponding coprocessor, it causes an assist emulation trap. The operation of a coprocessor when its corresponding CCR bit is 0 is explained in more detail in “Coprocessor Instructions” on page 5-178. It is an undefined operation to set to 1 any CCR bit corresponding to a coprocessor which is not present.

SFU Configuration Register

The SFU Configuration Register or SCR (bits 16..23 of CR 10), is an 8-bit register which records the presence and usability of special function units. The bit positions are numbered 0 through 7, and correspond to an SFU with the same unit identifier. Bit 1 corresponds to the debug SFU. Bit 7 is the rightmost bit of the SCR. It receives bit 23 from a general register when a general register is written to CR 10. The upper 16 bits of CR 10, and bits within the SCR corresponding to SFUs which are not present, are reserved bits.

For all SFUs, setting a bit in the SCR to 1 enables the use of the corresponding SFU, if present and operational. If an SCR bit is 0, the corresponding SFU, if present, is logically decoupled. This decoupling must ensure that the state of an SFU does not change as long as its corresponding SCR bit is 0. When an SCR bit is set to 0 and an attempt is made to execute an instruction which references the corresponding SFU, it causes an assist emulation trap. The operation of an SFU when its corresponding SCR bit is 0 is explained in more detail in “Special Function Unit (SFU) Instructions” on page 5-177. It is an undefined operation to set to 1 any SCR bit corresponding to an SFU which is not present.

Shift Amount Register

The Shift Amount Register or SAR (CR 11), is a 5-bit register used by the variable shift, extract, deposit, and branch on bit instructions. It specifies the number of bits a quantity is to be shifted. The remaining 27 bits are nonexistent bits and any value can be safely written in those positions.

Interruption Vector Address

The Interruption Vector Address or IVA (CR 14) contains the absolute address of the base of an array of service procedures assigned to the interruption classes. This address must be a multiple of 2048. This is because the lower 11 bits are reserved bits. Use of an unaligned address is an undefined operation. The array of interruption service procedures is indexed by the interruption numbers given in Chapter 4, “Flow Control and Interruptions”.

External Interrupt Enable Mask

The External Interrupt Enable Mask or EIEM (CR 15), is a 32-bit register containing a bit for each of the 32 external interrupts. When 0, bits in the EIEM mask interruptions pending for the external interrupts corresponding to those bit positions.

Interval Timer

The Interval Timer (CR 16) consists of two internal registers. One of the internal registers is continually counting up by 1 at a rate which is implementation-dependent and between twice the "peak instruction rate" and half the "peak instruction rate". Reading the Interval Timer returns the value of this internal register. The other internal register contains a comparison value and is set by writing to the Interval Timer. When the counter register and the comparison register contain identical values, bit 0 of the External Interrupt Request Register is set to 1. This causes an external interrupt, if enabled.

The Interval Timer can only be written by code executing at the most privileged level. If the PSW S-bit is 1, the Interval Timer can only be read by code executing at the most privileged level; otherwise, it can be read by code running at any privilege level.

In a multiprocessor system, each processor must have its own interval timer. Each interval timer need not be synchronized with the other interval timers in the system, nor do they need to be clocked at the same frequency.

If, as part of a power-saving mode, the processor clock is reduced below the "peak instruction rate", the Interval Timer continues to count at its peak rate. If the processor clock is stopped, the Interval Timer may also stop.

Interruption Instruction Address Queues

The Interruption Instruction Address Space Queue or IIASQ (CR 17) and the Interruption Instruction Address Offset Queue or IIAOQ (CR 18) are collectively termed the interruption instruction address or IIA queues. They are used to save the Instruction Address and privilege level information for use in processing interruptions. The registers are arranged as two two-element deep queues. The queues generally contain the addresses (including the privilege level field in the rightmost two bits of the offset part) of the two instructions in the IA queues at the time of the interruption. The IIASQ may be nonexistent, 16, 24, or 32 bits wide, as shown in Figure 2-8 on page 2-9.

The IIA queues are continually updated whenever the PSW Q-bit is 1 and are frozen by an interruption (PSW Q-bit becomes 0). After such an interruption, the IIA queues contain copies of the IA queues. Reading the IIAOQ (CR 18) while the PSW Q-bit is 0 retrieves the offset and privilege level portions of the front element in the IIAOQ. Writing into IIAOQ while the PSW Q-bit is 0 advances the IIAOQ and then sets the offset and privilege level portions of the back element of the IIAOQ. Reading the IIASQ (CR 17) while the PSW Q-bit is 0 retrieves the space portion of the front element of the IIASQ. Writing into IIASQ while the PSW Q-bit is 0 advances the IIASQ and then writes into the back element of the IIASQ. The effect of reading or writing either queue register while the PSW Q-bit is 1 is an undefined operation. The Interruption Instruction Address Queues are shown in Figure 2-12.

The state contained in the IIA queues is undefined when a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction sets the PSW Q-bit to 0, or when system control instructions are used to set the PSW Q-bit to 0. If an interruption is taken with the PSW Q-bit equal to 0, the IIA queues are unchanged.

IIASQ	Space Identifier
	Space Identifier

(nonexistent, 16, 24, or 32 bits)

	0	29 31
IIAOQ	Offset	PL
	Offset	PL

Figure 2-12. Interruption Instruction Address Queues

Interruption Parameter Registers

The Interruption Instruction Register or IIR (CR 19), Interruption Space Register or ISR (CR 20), and Interruption Offset Register or IOR (CR 21) are collectively termed the interruption parameter registers or IPRs. They are used to pass an instruction and a virtual address to an interruption handler. The values in these registers for each interruption class are specified in Chapter 4, “Flow Control and Interruptions”. These values are set (or frozen) at the time of the interruption whenever the PSW Q-bit is 1. The ISR may be nonexistent, 16, 24, or 32 bits wide, as shown in Figure 2-8 on page 2-9.

The value loaded into the IOR is the full 32-bit offset of the virtual address without truncating the rightmost bits or setting them to 0. Writing into the interruption parameter registers is an undefined operation.

The values contained in the interruption parameter registers can be read reliably only when the PSW Q-bit is 0. The state contained in the IPRs is undefined when a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction sets the PSW Q-bit to 0, or when system control instructions are used to set the PSW Q-bit to 0. If an interruption is taken with the PSW Q-bit equal to 0, the IPRs are unchanged.

Interruption Processor Status Word

The Interruption Processor Status Word or IPSW (CR 22) receives the value of the PSW when an interruption occurs. The layout of IPSW is identical to that of the PSW. The IPSW always reflects the state of the machine at the point of interruption, regardless of whether the PSW Q-bit was 1 or not. As in the PSW, the unnamed bits are reserved bits. In Level 0 systems, the X, C, P, and D bits of IPSW are nonexistent bits.

The value contained in the IPSW can be read or written reliably only when the PSW Q-bit is 0. The state contained in the IPSW is undefined when a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction sets the PSW Q-bit to 0, or when system control instructions are used to set the PSW Q-bit to 0.

External Interrupt Request Register

The External Interrupt Request register or EIRR (CR 23) is a 32-bit register containing a bit for each external interrupt. When 1, a bit designates that an interruption is pending for the corresponding external interrupt. Both the PSW I-bit (external interrupt, power failure interrupt, and low-priority machine check unmask) and the corresponding bit position in the External Interrupt Enable Mask (CR 15) must be 1 for an interruption to occur.

A MOVE TO CONTROL REGISTER instruction with CR 23 as its target bitwise ANDs the complement of the contents of the source register with the previous contents of CR 23, and places this result in CR 23. Thus the processor can only set the EIR register bits to 0.

The IO_EIR register, which is part of the I/O subsystem, is the external name for the EIR register. When a module writes to it, the bit specified by the value written is set to 1.

Temporary Registers

Six of the eight 32-bit temporary registers (CRs 24, 25, 28 .. 31) are accessible only by code executing at the most privileged level. They provide space to save the contents of the general registers for interruption handlers in the operating system kernel.

The other two temporary registers (CRs 26 and 27) are readable by code executing at any privilege level and writable only by code executing at the most privileged level.

Coprocessor Registers

Each coprocessor may have its own register set. The coprocessor mechanism is described in "Assist Instructions" on page 5-176. The floating-point coprocessor registers are described in Chapter 6, "Floating-point Coprocessor". The performance monitor coprocessor registers are described in Chapter 7, "Performance Monitor Coprocessor".

SFU Registers

Each special function unit may have its own register set. The SFU mechanism is described in "Assist Instructions" on page 5-176. The debug SFU registers are described in Chapter 8, "Debug Special Function Unit".

3 Addressing and Access Control

Introduction

PA-RISC processors use byte addressing to fetch instructions and data from main memory or the I/O registers. The byte addresses may be either absolute addresses or virtual addresses. When absolute addresses are used directly, no protection or access rights checks are performed. Memory accesses using absolute addresses are called **absolute accesses**. Virtual addresses are translated to absolute addresses and undergo protection and access rights checking. Memory accesses using virtual addresses are called **virtual accesses**.

The instructions that reference memory are load (memory-to-register), store (register-to-memory), and semaphore instructions. Several system control and cache-related instructions generate addresses that use the address translation, protection, and access rights checking mechanisms. Computation instructions do not reference memory, but perform data transformations by using values obtained from general registers and returning results to these registers.

Four **Levels** of the processor architecture have been defined: **0**, **1**, **1.5**, and **2**. Level 0 systems are fundamentally different from Level 1, 1.5, and 2 systems. Level 0 systems support only absolute addressing and have no Space Registers. Level 1, 1.5, and 2 systems provide virtual addressing through the use of 16-bit, 24-bit, and 32-bit Space Registers, respectively.

Virtual memory is structured as a set of virtual spaces, each containing 2^{32} bytes (4 gigabytes). Level 1 processors have 2^{16} address spaces, Level 1.5 processors have 2^{24} address spaces, and Level 2 processors have 2^{32} address spaces.

During virtual address translation, a space is selected by a **space identifier** contained in the upper portion of the virtual address. The byte **offset** within the space is specified by the lower 32 bits of the virtual address.

For memory management purposes, the address space is logically subdivided into **pages**, each 4 Kbytes in length. The byte offset into the page is specified by the least significant 12 bits of the virtual address. Figure 3-1 illustrates the structure of spaces, pages, and offsets.

Support is provided for the emulation of larger page sizes. Eight contiguous pages, with the first of these pages beginning on a 32K byte boundary, are referred to as a **page group**.

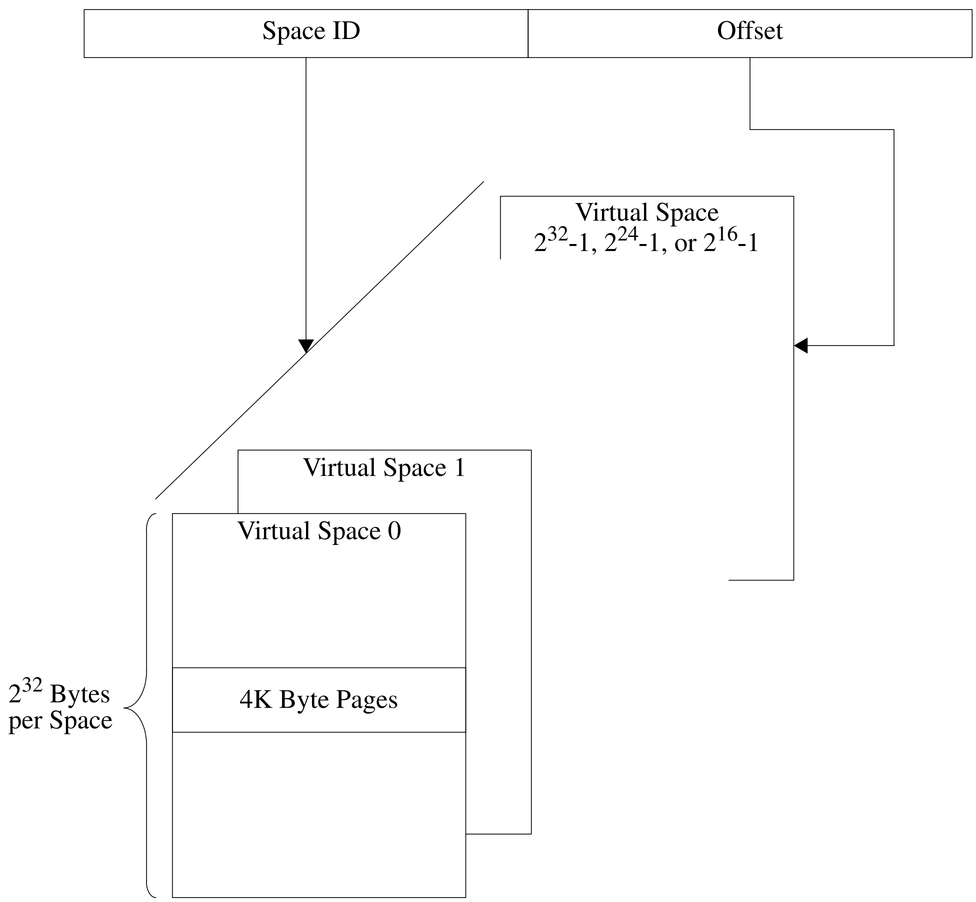


Figure 3-1. Structure of Spaces, Pages, and Offsets

Pointers and Address Specification

In systems which support virtual addressing, the eight Space Registers, the Instruction Address Space Queue, and two of the Control Registers are used to maintain space identifiers. They are used in virtual address calculations for both instructions and data. In Level 0 systems, the Space Registers, IASQ, and CRs 17 and 20 are nonexistent registers.

The Space Registers are used to compute instruction addresses for instruction cache flush, instruction TLB instructions, and for some branch target calculations. Addresses for instruction fetch and some branch target calculations are generated from the IA queues. When an instruction address is computed for an external branch target, the 3-bit s-field in the instruction selects the Space Register to be used. Instruction addresses are aligned on word boundaries and the least significant two bits of the offset are used to hold the privilege level.

The current instruction address (IA) consists of a space identifier and a 32-bit byte offset. The byte offset is a word-aligned address and contains, in its least significant two bit positions, the current privilege level. This privilege level controls both instruction and data references. The current instruction address is maintained in the front elements of the Instruction Address Queues.

Data addresses are computed for load, store, semaphore, data cache, probe, and data TLB instructions. The 32-bit offset within the virtual address space is the sum of all 32 bits of the base register plus the 32-bit index register or a sign-extended displacement.

The space identifier, for data references, is selected from Space Registers 1 through 7 by the following procedure. The 2-bit s-field of the instruction, when nonzero, selects corresponding Space Registers 1, 2, or 3. When the s-field is zero, four is added to the two most significant bits of the base register to select one of the Space Registers 4 through 7. Figure 3-2 illustrates space identifier selection. Data references with the s-field equal to zero permit addressing of four distinct spaces selected by program data. This is called short pointer addressing since a 32-bit value is an offset and selects a Space Register. Only one fourth of the space is directly addressable by the base register with short pointers and corresponds to the quadrant selected by the upper two bits. For example, if a base register contains the value 0x40001000 and the s-field is zero, Space Register 5 is used as the space identifier and the second quadrant of the space is directly addressable.

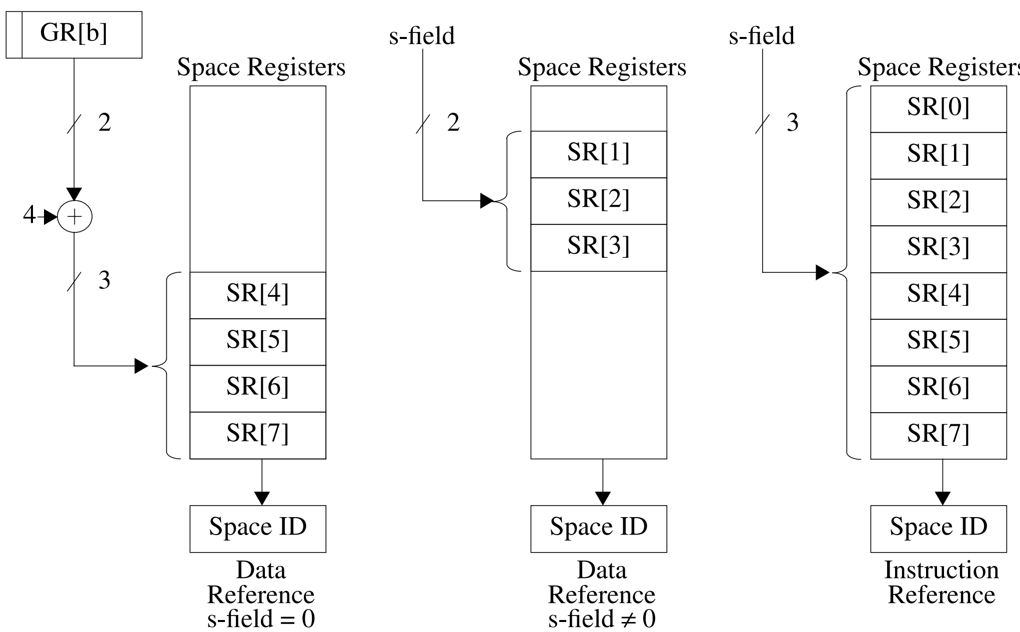


Figure 3-2. Space Identifier Selection

In Level 0 systems or when data translation is disabled (PSW D-bit is 0), the s-field of the instruction is ignored and the 32-bit offset is directly used as the address.

Address Resolution and the TLB

Virtual addresses are translated to absolute addresses using a hardware structure called the **Translation**

Lookaside Buffer (TLB). A TLB accepts a Virtual Page Number and returns the corresponding Physical Page Number. Since all references in Level 0 systems are absolute accesses, these systems do not have TLBs. A TLB is typically not large enough to hold all the current translations. Translations for all pages in memory are stored in a memory structure called the **Page Table**. The TLB is organized as two parts. The instruction TLB (ITLB) is only used for instruction references, while the data TLB (DTLB) is only used for data references. A system may implement a **combined** TLB which is used for both instruction and data references.

Additionally, translations are supported for large address ranges. Such translations are called **block translations** and are stored in a block TLB. Block translations map address ranges larger than a page. Block translations are useful in mapping virtual address ranges which do not get paged in and out. These block translations increase the virtual address range of the TLB thereby minimizing the virtual address translation overhead.

Given a virtual address, the selected TLB is searched for an entry matching the Virtual Page Number. If the entry exists, the 20-bit Physical Page Number (contained in the TLB entry) is concatenated with the original 12-bit page offset to form a 32-bit absolute address. If no such entry exists, the TLB is updated by either software TLB miss handling or hardware TLB miss handling.

In systems with software TLB miss handling, a TLB miss fault interruption routine performs the translation, explicitly inserts the translation and protection fields into the appropriate instruction TLB or data TLB, and restarts the interrupted instruction. To insure the completion of instructions, the TLB must be organized to simultaneously hold all necessary translations.

NOTE

To fulfill the requirement of holding all the translations simultaneously, it is necessary to have either split instruction and data TLBs, or a two or greater way set associative combined TLB. In a system with hardware TLB miss handling, the machine need only insure forward progress. Normally, at most only one instruction and one data address translation are needed. The FIC instruction is special. Its definition allows using the DTLB for the translation of the target's address.

In implementations that provide hardware for TLB miss handling, the hardware attempts to find the virtual to physical page translation in the Page Table. If the hardware is successful, it inserts the translation and protection fields into the appropriate instruction or data TLB. No interruption occurs in this case. If hardware is not successful, due to a search of the Page Table that was not exhaustive or due to the appropriate translation not existing in the Page Table, an interruption occurs so that the software can complete the process.

The translation lookaside buffer performs other functions in addition to the basic address translation. The other functions include access control, program debugging support and operating system support for virtual memory. Figure 3-3 summarizes the information maintained for each TLB entry.

Virtual Page Number

36, 44, or 52 bits

Physical Page Number

20 bits

Access Rights

7 bits

Access ID

15 to 18 bits

E	U	T	D	B	Single Bit Flags

Figure 3-3. TLB Fields

The following describes the function of each of the 1-bit fields.

E Entry Valid. When 1, the translation is valid.

U Uncacheable. When 0, memory references to a page from memory address space may be moved into the cache. When 1, memory references to a page from I/O address space or memory address space must not be moved into the cache. The U-bit must be set to 1 for pages which map to the I/O address space, and is commonly set to 1 for pages in the memory address space where I/O module written data and processor written data must co-exist within the same cache line. Implementation of the U-bit is optional. See “Data Cache Move-In” on page 3-21 for additional details.

T Page Reference Trap. When 1, data references using this translation cause a page reference trap. The T-bit is most commonly used for program debugging.

D Dirty. When 0, store and semaphore instructions cause a TLB dirty bit trap. When 1, no trap occurs. The D-bit may be used by the operating system to determine which pages have been modified.

B Break. When 1, instructions that could modify data using this translation cause a data memory break trap, if enabled. Store instructions, the PURGE DATA CACHE instruction, and semaphore instructions are the only instructions that potentially modify data. The B-bit is most commonly used for program debugging.

Since the ITLB is not used for data operands, the U, T, D, and B bits are only implemented in the DTLB or a combined TLB. The translation lookaside buffer is managed by a mixture of hardware and software mechanisms. Translations are brought into the TLB by either hardware or software when a TLB miss occurs. In systems which provide hardware for TLB miss handling, the Page Table holds the information needed for the TLB. For systems with software TLB miss handling, and for explicit insertion of a translation by systems with hardware TLB miss handling, a pair of TLB management instructions provide the TLB with this information. The INSERT INSTRUCTION TLB ADDRESS and INSERT INSTRUCTION TLB PROTECTION instructions place the complete translation and access control information into the ITLB. A similar pair (IDTLBA, IDTLBP) places the complete translation and access control information and also initializes the system software and debugging support bit fields in the DTLB.

Software can be written to operate with a logical page size of 4, 8, 16, or 32 Kbytes. When performing TLB miss handling for a given page, up to all of the eight translations for the page group containing that page may be inserted into the TLB, provided that the translation for the given page is inserted last.

On systems with software TLB miss handling, TLB miss traps do not occur on nullified instructions.

Address Aliasing

Normally, a virtual address does not translate to two different absolute addresses. It is the responsibility of memory management software to avoid the ambiguity such occurrences would create.

Caches are required to permit a physical memory location to be accessed by both an absolute and a virtual address when the virtual address is equal to the absolute address. Such a virtual address is said to be **equivalently-mapped**.

The instruction and data caches are required to detect that the same physical memory location is being accessed by two virtual addresses that satisfy all the following requirements:

1. The two virtual addresses map to the same absolute address.
2. Offset bits 12 through 31 are the same in both virtual addresses.
3. If the use of space bits in generating the cache index is enabled, the two virtual addresses differ only in the following space identifier bits: 0, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, and 19.

Processors must provide an implementation-dependent mechanism to enable/disable the use of space bits in generating the cache index.

These rules provide offset aliasing on 1 Mbyte boundaries, with optional support for offset aliasing on smaller power of two sized boundaries, and either restricted or unlimited space aliasing.

Two virtual addresses that satisfy all of the above requirements are called **equivalent aliases** of each other. Virtual addresses that satisfy rule 1 but violate rule 2 or 3 are **non-equivalent aliases**, and are more restricted in their use. For non-equivalent aliases, **read-only** aliasing is supported with minimal restrictions, and **many-reader/one-writer** aliasing is supported with more significant restrictions.

Generally, if system software must use multiple addresses for the same data, these addresses are equivalent aliases (or are an absolute address and an equivalently-mapped virtual address). This description of non-equivalent aliasing, and the restrictions on software only apply in rare situations when non-equivalent aliasing is necessary.

For the purposes of supporting non-equivalent aliasing, a **read-only translation** is defined as one where the TLB and page table both meet at least one of the following conditions:

- The page type in the access rights field is 0, 2, 4, 5, 6, or 7. (See "Access Control" on page 3-10.)
- The D-bit (dirty bit) is 0.

A translation not meeting this requirement is termed a **write-capable translation**.

Software is allowed to have any number of read-only non-equivalently aliased translations to a physical page, as long as there are no other translations to the page. This is refered to as **read-only** non-equivalent aliasing.

Before a write-capable translation is enabled, *all* non-equivalently-aliased translations must be removed from the page table and purged from the TLB. (Note that the caches are not required to be flushed at this time.) The write-capable translation may then be used to read and/or modify data on that page. Before any non-equivalent aliased translation is re-enabled, the virtual address range for the writable page (the entire page) must be flushed from the cache, and the write-capable translation removed from the page table and purged from the TLB. If an old read-only translation is re-enabled, or a translation is enabled that is equivalently-aliased to an old translation, the virtual address range for the re-enabled translation must be flushed from the cache before accesses are made to the page. (This flushing is only required if the re-enabled virtual page has not been flushed since it was last accessed.) This is refered to as **many-reader/one-writer** non-equivalent aliasing.

Absolute read accesses can be made to a page which is mapped with a non-equivalently-mapped read-only transaltion, as long as the absolute address range accessed is flushed before enabling any write-capable translation. Since absolute accesses do not cause prefetching, it is not necessary to flush the entire page - only the accessed range need be flushed.

All other uses of non-equivalent aliasing (including simultaneously enabling multiple non-equivalently aliased translations where one or more allow for write access) are prohibited, and can cause machine checks or silent data corruption, including data corruption of unrelated memory on unrelated pages. It is the responsibility of privileged software to avoid non-equivalent aliasing, except as described above. This requires flushing the affected address range from the caches prior to any of the following:

- Changing the address mapping in the TLBs.
- Making an absolute access to a location which might reside in the caches as a result of an access by a virtual address that was not equivalently mapped.
- Making a virtual access to a location which might reside in the caches as a result of an access by its absolute address that was not equivalently mapped.
- Making a virtual access to a location which may reside in the caches as a result of an access by another virtual address that was not equivalently aliased.

NOTE

The restrictions on non-equivalent aliases are necessary to allow the design of high-performance caches and memory interconnect, including multi-level caches (including victim or miss caches) and directory-based coherency structures. Coherency schemes are greatly simplified by allowing the assumption that there is at most a single private copy of a physical line at any time. The read-only translation informs hardware to request a shared or public copy of the line.

TLB Control

TLBs function as buffers for the most frequently used address translations. Terms used to describe TLBs are given below.

Entry — The term **entry** refers to a translation, either valid or invalid, which is present in the TLB. Entries are visible to software through either references (such as load, store, and semaphore instructions, access rights probes, and the LOAD PHYSICAL ADDRESS instruction) or insert TLB protection instructions (IITLBP and IDTLBP).

Slot — Hardware resources in the TLB which hold entries are referred to as **slots**.

Invalidate — An entry is **invalidated** when its E-bit is set to 0 leaving the Virtual Page Number and Physical Page Number fields unchanged. Invalid entries in the TLB are still visible to software through insert TLB protection instructions.

Remove — An entry is **removed** when some action causes it to be inaccessible to software.

Insertion of translations into the TLB, for example, causes other entries to be removed. TLB systems can also remove an entry by logically decoupling from the TLB the slot which contained the entry.

NOTE

In a two-level TLB, moving a translation from the second level to the first level can cause a translation to disappear from the TLB (because the level 1 TLB need not be a proper subset of the level 2 TLB). This is an example of removal.

Alter — An entry is **altered** when its E-bit is set to 0, and the Physical Page Number field is modified, but the Virtual Page Number field is left unchanged. Altered entries in the TLB are still visible to software through the insert TLB protection instructions.

Several mechanisms can be used by software to invalidate or remove a specific translation from the ITLB or DTLB. First, when a new translation is inserted into the TLB, the old translation for the same Virtual Page Number (if present) is removed. Second, a specific virtual address may be used to purge (invalidate or remove) the associated translation from the TLB. The PURGE INSTRUCTION TLB and PURGE DATA TLB instructions perform this function. These instructions also cause the translation to be invalidated or removed from the TLBs of other processors in a multiprocessor system. There is no instruction to purge a translation from both the instruction and data TLBs simultaneously.

Translations may also be invalidated or removed from the TLB using the PURGE INSTRUCTION TLB ENTRY and the PURGE DATA TLB ENTRY instructions. These purge zero or more machine specific entries in the TLB without regard for the translation. These instructions are used by system software to clear the entire instruction or data TLB.

Because the TLB is managed by a mixture of hardware and software mechanisms, software may not, in general, rely on the existence of translations in the TLB and hardware may, in general, invalidate or remove TLB entries at any time, provided that forward progress is assured. There are limited situations, however, in which software may rely on a translation existing in the TLB. This means that software may make virtual accesses using this **relied-upon translation**, and no TLB miss fault will occur. Hardware is required to retain this TLB entry as long as the constraints of the limited situation are met. These situations are described in “TLB Operation Requirements” on page 3-18.

NOTE

As a result, the following hardware actions are allowed, except in the defined limited situations.

- A TLB miss fault may be taken even though the translation exists in the TLB.
- In the event of a TLB error, one or more entries may be invalidated or removed.

Note also that software may not rely on the existence of any translations in the TLB immediately after any group 1, 2, or 4 interruption.

The following are the only ways in which TLB entries may exist which are visible to software and which were not inserted into the TLB by software.

1. When a TLB entry is inserted by a hardware TLB miss handler.
2. When a valid TLB entry is invalidated by a purge TLB instruction, the TLB entry may be altered.
3. When a valid TLB entry is invalidated by a purge TLB instruction executed on another processor in a multiprocessor system, the TLB entry may be altered.
4. The PURGE INSTRUCTION TLB ENTRY and PURGE DATA TLB ENTRY instructions, as well as initialization and power failure can cause all fields but the E-bit to be set to arbitrary values in some or all of the TLB entries. The E-bit is set to 0 in all entries modified by these instructions.

Software must manage the virtual to absolute address mappings with the knowledge that purge TLB instructions can alter translations in other processors and that initialization, power failure, and purge TLB entry instructions leave the TLB with arbitrary invalid translations.

Software TLB Miss Handling

In order to insure forward progress, some restrictions are placed on software which performs TLB miss handling.

For instruction TLB miss handling, the following restrictions apply:

- Software can insert multiple instruction address translations into the ITLB, using INSERT INSTRUCTION TLB ADDRESS and INSERT INSTRUCTION TLB PROTECTION instructions, provided that the translation which caused the trap is inserted last.
- Software must not execute a purge TLB instruction using the virtual address corresponding to the data address translation needed for the execution of the trapping instruction.
- Software must not insert translations into the DTLB.

For non-access instruction TLB miss handling, the following restrictions apply:

- Software can only insert into the ITLB up to all of the eight translations for the page group. The translation which caused the trap must be inserted last.
- Software must not execute a purge TLB instruction using the virtual address corresponding to the data address translation needed for the execution of the trapping instruction.

- Software must not insert translations into the DTLB.

For data TLB miss handling, the following restrictions apply:

- Software can only insert into the DTLB up to all of the eight translations for the page group. The translation which caused the trap must be inserted last.
- Software must not execute a purge TLB instruction using the virtual address corresponding to the instruction address translation needed for the execution of the trapping instruction.

For non-access data TLB miss handling, the following restrictions apply:

- Software can only insert into the DTLB up to all of the eight translations for the page group. The translation which caused the trap must be inserted last.
- Software must not execute a purge TLB instruction using the virtual address corresponding to the instruction address translation needed for the execution of the trapping instruction.

The following restrictions apply to all four TLB miss handlers:

- Software must not make any virtual references.
- Software must not execute any PURGE DATA TLB ENTRY or PURGE INSTRUCTION TLB ENTRY instructions.

Hardware TLB Miss Handling

The default endian bit (see “Byte Ordering (Big Endian/Little Endian)” on page 2-3) determines how data from the hardware-visible page table is interpreted by the hardware TLB miss handler, if implemented. If the default endian bit is 0, the hardware-visible page table entries are loaded as words in big-endian format; if the default endian bit is 1, the entries are loaded as words in little-endian format.

Access Control

A set of mechanisms is available to system software to provide a secure and protected environment for user processes. These mechanisms are collectively known as access control and are provided as a part of the address translation mechanism. Processor resources, including the PSW, Control Registers, and TLB entries, contain information used to determine the allowed use of a page. Access control is available only when address translation is enabled, and is done on a per-page basis. Access control is not available in Level 0 systems.

An access is validated if the check of the access rights and the protection identifiers both succeed. If the access is validated, the instruction reference or data reference is completed. If the access is not validated, the instruction is terminated with a protection trap. Instruction access violations are reported with instruction memory protection traps. Data read and write access violations are reported with data memory access rights or data memory protection ID traps. Probe instructions are special; they save the result of the access validation in a General Register and do not cause a protection trap. An access rights check is based on the type of access and the current privilege level. The protection identifier check compares the Protection ID Registers with a page-based access identifier in the TLB. State bits within the PSW determine when these checks are enabled.

The type of access, privilege level, the current values in the Protection ID Registers, and the state of the PSW completely describes the access to the TLB. These resources are managed for each process by the operating system and collectively termed the **process attributes**. The following defines each of the process attributes.

Privilege Level (PL)

Every instruction is fetched and executed at one of four privilege levels (numbered 0, 1, 2, 3) with 0 being the most privileged. In Level 0 systems, there are only two distinct privilege levels - 0 and non-zero; 1, 2, and 3 are equivalent. The privilege level is kept in bits 30 and 31 of the current instruction's address (the front element of IAOQ). For all accesses, except the probe instructions, the privilege check uses the privilege level of the current instruction. The probe instructions explicitly specify the privilege level to be used in the access rights check.

Access type

The access type is either read, write, or execute. Load, semaphore, and read probe instructions make **read accesses** to their operands. Store, semaphore and write probe instructions and cache purge operations make **write accesses** to their operands. Note that semaphore instructions make both read and write accesses to their operands. The only **execute access** occurs when the current instruction is fetched for execution.

Protection IDs

The four Control Registers CR 8, CR 9, CR 12, and CR 13 contain the protection identifiers associated with the current process (Figure 3-4). These registers are used to allow several different protection groups to be accessed. The least significant bit is the write-disable (WD) bit. When 0, write accesses that match that protection ID are allowed. The remaining 15 to 18 bits hold the protection ID. Figure 3-4 depicts the maximum width of the protection identifier.

0 … 17	18
Protection ID	WD
18	1

Figure 3-4. Protection ID

PSW access attributes

The PSW protection validation (P-bit), code address translation (C-bit), and data address translation (D-bit) bits further qualify the process attributes. When address translation is enabled and the P-bit is 1, the protection ID check is performed. When 0, the protection ID check is always considered successful. An execute access uses the C-bit to determine if address translation and access rights check are enabled. When 1, address translation is performed and execute access rights checks are made. When 0, no address translation is performed and the access is always allowed. Read and write accesses use the D-bit in an equivalent manner. For probe instructions, address translation is performed, and access rights checks are made independent of the state of the PSW C and D-bits.

For each entry in the TLB, the **access ID** and the **access rights** fields determine if an access is allowed. The access ID is a 15- to 18-bit field in the TLB that is used with the protection IDs in the protection ID check. The length of the access ID is implementation dependent but must match the length of the protection ID (excluding the WD bit).

The access rights field (Figure 3-5) is a 7-bit field that encodes the allowed access types and the needed privilege levels. In some cases a minimum privilege is specified, while other access types may be specified with an upper and a lower bound. The three sub-fields **type**, **PL1** (privilege level 1), and **PL2** (privilege level 2) combine to form the access rights field. The type sub-field defines the type of access that can be made to this page. Any of read-only, read/write, read/execute, read/write/execute, or execute-only is allowed. The PL1 sub-field qualifies read and execute accesses. The PL2 sub-field qualifies write and execute accesses.

Type	PL1	PL2
3	2	2

Figure 3-5. Access Rights Field

The access rights check compares the current privilege level with the appropriate sub-field of the TLB access rights field and checks if the type of access is allowed. For a read access, the current privilege level must be at least as privileged as PL1 and the type field must allow read access. The read probe instructions explicitly specify the privilege level.

For a write access, the current privilege level must be at least as privileged as PL2 and the type field must allow write access. The write probe instructions explicitly specify the privilege level.

For an execute access, the current privilege level must be at least as privileged as PL1 and no more privileged than PL2. PL1 and PL2 are a lower and an upper bound, respectively, for execute access. The type field must also allow execute access.

The type field is also used by the GATEWAY instruction to specify the new privilege level. When the type value is 4 or greater and the encoded new privilege level is of greater privilege, then promotion occurs at the target of the branch. Promotion may occur at the instruction following the GATEWAY instruction for some implementations. Software cannot depend on the privilege level of the instruction following the GATEWAY instruction.

Table 3-1 defines the type encodings and the necessary conditions of the PL1 and PL2 fields with the current privilege level (PL). This table uses the actual binary encoding when doing the privilege level comparison.

The protection identifier check compares the four Protection ID Registers with the TLB entry's access ID. This check is validated if one or more of the protection IDs compare equal with the access ID. In case of a write access, the write disable bit of at least one of the matching protection IDs must be zero for the check to be validated. An access ID of zero is special and specifies a public page. A public page always satisfies a protection ID check for any type of access and only an access rights check is performed. If no match occurs and a public page is not being referenced, then the access is not allowed.

The PSW P-bit determines whether the protection ID check is performed. When 0, no protection check occurs and only the access rights check is performed. Figure 3-6 on page 3-14 illustrates the access rights and protection ID checks and the processor resources that participate.

Table 3-1. Access Rights Interpretation

Type value (in binary)	Allowed access types and GATEWAY promotion	Privilege check
000	Read-only: data page	read: $PL \leq PL1$ write: Not allowed execute: Not allowed
001	Read/Write: dynamic data page	read: $PL \leq PL1$ write: $PL \leq PL2$ execute: Not allowed
010	Read/Execute: normal code page	read: $PL \leq PL1$ write: Not allowed execute: $PL2 \leq PL \leq PL1$
011	Read/Write/Execute: dynamic code page	read: $PL \leq PL1$ write: $PL \leq PL2$ execute: $PL2 \leq PL \leq PL1$
100	Execute: promote to privilege level 0*	read: Not allowed write: Not allowed execute: $PL2 \leq PL \leq PL1$
101	Execute: promote to privilege level 1*	read: Not allowed write: Not allowed execute: $PL2 \leq PL \leq PL1$
110	Execute: promote to privilege level 2*	read: Not allowed write: Not allowed execute: $PL2 \leq PL \leq PL1$
111	Execute: remain at privilege level 3*	read: Not allowed write: Not allowed execute: $PL2 \leq PL \leq PL1$

* Change of privilege level only occurs if the indicated new value is of higher privilege than the current privilege level; otherwise the target of the GATEWAY executes at the same privilege as the GATEWAY itself.

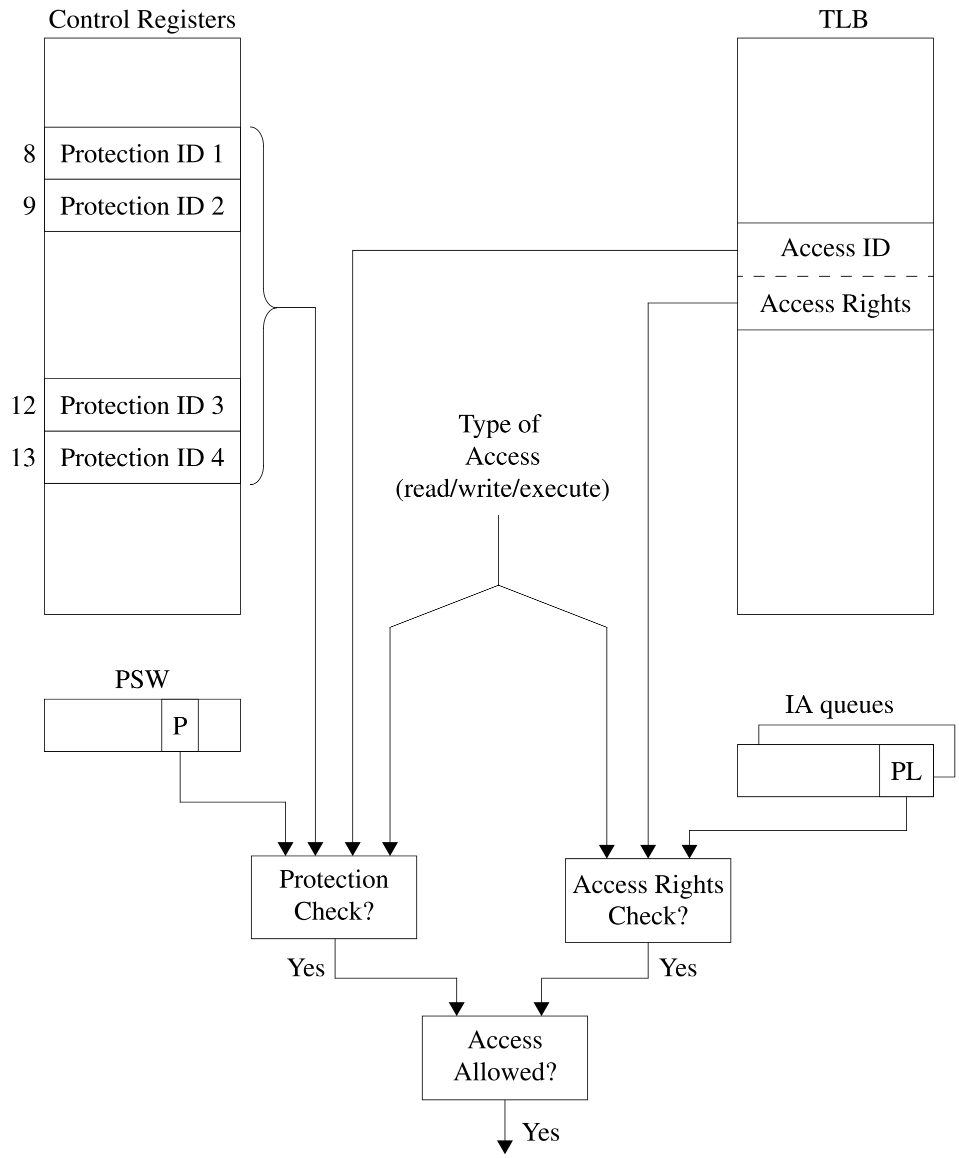


Figure 3-6. Access Control Checks

Page Table Structure

Address translations are stored in memory in a structure called the **Page Table**. The exact form of these tables is a software convention, but many aspects of the page tables are common.

The most common use of the Page Table is to translate a virtual address to a physical address after a TLB miss. The virtual address space is quite large, and a traditional approach of a multi-level forward mapped table, where each level is directly indexed by a portion of the virtual address, requires too many memory accesses and hence is an inefficient way to provide virtual to physical translations.

A better approach is to index the Page Table using the result of a hash function applied to the virtual address. The purpose of the hash function is to translate virtual addresses to a smaller, more uniform name space. The particular function used is implementation dependent. Collisions created by multiple addresses hashing to the same entry can be resolved using a sequentially searched linked list or some other structure.

The number of entries in the Page Table is typically a power of two. One possible format of a table entry is shown in Figure 3-7.

<table>
<tr><td>V</td><td colspan="9">Tag (31)</td></tr>
<tr><td>R</td><td>S</td><td>T</td><td>D</td><td>B</td><td>Access Rights (7)</td><td>U</td><td colspan="2">Access ID (15, 16, 17, or 18)</td><td>S</td></tr>
<tr><td colspan="3">S (3)</td><td colspan="3">0 (4)</td><td colspan="2">Physical Page Number (20)</td><td>S (2)</td><td>0 (3)</td></tr>
<tr><td colspan="10">Next Page Table Entry (32)</td></tr>
</table>

Figure 3-7. Page Table Entry

The fields are:

V — is the valid bit. If V = 1, this entry represents a valid translation.

Tag — is a unique key used to identify the virtual address that this entry translates.

R — is the reference bit. If R = 1, the page has been accessed (read, write, or execute) by a processor since the bit was last cleared to 0.

Physical Page Number

is the physical page number corresponding to the virtual address, provided this entry is valid and the virtual address matches the tag.

Next Page Table Entry

is an index/pointer to perhaps another structure containing overflow page table entries.

0 — is a reserved bit field.

S — is a bit field reserved for operating system use.

The U, T, D, B, Access Rights, and Access ID fields correspond to those for TLB entries (see “Address Resolution and the TLB” on page 3-3).

Caches

Cache memories are high-speed intermediate storage buffers which contain recently accessed instructions and data. These caches need not remain coherent with memory and I/O. Storage items may be brought into the cache, only as a result of references made by the instruction stream. Software can explicitly remove items from the cache. As a result, software can control which portions of memory may be present in the cache. Items in the cache may be removed by hardware at any time. Software may therefore not rely on particular items remaining in the cache.

A consistent software view of cache operation requires that implementations never write a clean cache line back to memory. **Clean** means "not stored into" as opposed to "not changed". **Dirty** means "stored into". A cache line which was stored into in such a way that it was unchanged is considered to be dirty.

To insure memory system coherence, and to minimize cache flushing, instructions and data in memory may be brought into the caches only under certain circumstances. This operation of bringing information from memory into a cache is referred to as **move-in**. In general, only instructions and data referenced by executed instructions may be moved in. Software may use reference bits and other mechanisms controlled by interruptions to determine when lines are potentially in the instruction cache, data cache, or both.

No data reference may cause a move-in to the instruction cache and no instruction reference may cause a move-in to the data cache. This means that the execution of a FLUSH DATA CACHE or PURGE DATA CACHE instruction guarantees that the addressed line, if it has been referenced as data but not as instructions, is no longer present in the cache system. Similarly, only a FLUSH INSTRUCTION CACHE instruction is required to guarantee that a line which has been referenced as instructions but not as data, is no longer present in the cache system. The actions which constitute a reference are described in “Data Cache Move-In” on page 3-21 and “Instruction Cache Move-In” on page 3-22.

If implemented, the U (Uncacheable) bit is found in the data TLB entry associated with a page. Whether or not the U-bit is implemented, the state of this bit if implemented, whether the memory reference is virtual or absolute, and whether the reference is made from a page in the memory or I/O address spaces determine if the reference may be moved into the data cache. The detailed rules for moving references into the data cache are specified in “Data Cache Move-In” on page 3-21.

Software must set the U-bit associated with all pages in the I/O address space to 1. Referencing a page in the I/O address space for which the U-bit is 0 is an undefined operation.

Changing the state of the U-bit for a page has no effect on the data cache lines from that page which already exist in the cache. A page from the memory address space which has its U-bit set to 0 is called a **cacheable** page. Pages from the I/O address space and pages which have their U-bit set to 1 are called **uncacheable** pages. It is possible for data cache lines from an uncacheable page to exist in a data cache. This case may be caused by changing a cacheable page to uncacheable after references to this page were moved into the data cache.

The Synchronization Primitive

Program synchronization can be done using the LOAD AND CLEAR WORD instructions, which perform indivisible semaphore operations. These instructions are required to use 16-byte aligned addresses. When using semaphores to synchronize with I/O, care must be taken in placing other information in the same cache line as the semaphore. Data which is writable, can only be placed in the same cache line as a semaphore if access to write the data is controlled by the semaphore.

Cache Coherence with I/O

Accesses to memory by I/O modules may be either coherent or non-coherent with processor data caches.

Coherent I/O

Processors in systems with coherent I/O modules must implement the LOAD COHERENCE INDEX instruction, which loads the coherence index corresponding to a given virtual address into a General Register. Coherent I/O modules provide the coherence index along with the absolute address of data it is reading from or writing to memory. The coherence index must provide enough information such that, together with the absolute address, the processor can find data that was brought into its data cache by the original virtual address.

Software need not flush or purge data from the data cache when sharing the data with a coherent I/O module. For I/O output (e.g., memory to secondary storage), the coherent I/O module performs coherent read operations which will read the data from memory or a processor's data cache depending on where the most up-to-date copy is located. For I/O input, the coherent I/O module performs coherent write operations which will write the data to memory and also update or invalidate matching lines in processor data caches.

Coherent I/O operations are not coherent with instruction caches. Software is responsible for flushing the appropriate instruction cache lines before or after the I/O operation.

Non-coherent I/O

Non-coherent I/O modules process data in memory; this data can be non-coherent with processor caches. Software is required to insure that:

1. The contents of the appropriate caches are flushed to main memory prior to an I/O output (e.g., memory to secondary storage) operation.
2. The contents of the appropriate caches are purged or flushed prior to an I/O input (e.g., secondary storage to memory) operation.
3. The contents of the appropriate caches are purged following an I/O input operation, if the cache move-in rules would have allowed the processor to move the data into the cache during the I/O operation.

Cache Coherence in Multiprocessor Systems

Multiprocessor systems may include PA-RISC processors as well as other processors.

The cache-coherent part of a multiprocessor system is required to behave as if there were logically a single D-cache and a single I-cache. If there are multiple physical D-caches, they must cross-interrogate for current data and must broadcast purge and flush operations except for FDCE and FICE. Purge and flush operations do not cause TLB faults on other processors. Multiple I-caches require only that flushes be broadcast. The I-cache is read-only, and software is responsible for modifications to the instruction stream.

The non-cache-coherent part of a multiprocessor system (if any) may either cross-interrogate with the caches in the cache-coherent part of the system, or may have an independent cache system. This design decision is generally based on the frequency of data sharing.

In the cache-coherent part of a multiprocessor system, all data references to cacheable pages must be satisfied by data that was obtained using cache coherence checks, and has remained coherent since the data was moved in. Data references to uncacheable pages do not need to be satisfied by data that was obtained using cache coherence checks. Data from an uncacheable page could be in a cache if it was moved in when that page was marked cacheable, but the page is now marked uncacheable.

Implementations with write buffers must also check buffer contents on cache coherence checks, in order to insure proper ordering of storage accesses.

Instruction references need not be satisfied by data that was obtained using cache coherence checks.

Instruction caches are read-only. In the case of a separate instruction cache implementation, instruction cache lines must never be written back to main memory.

TLB Coherence in Multiprocessor Systems

The cache-coherent part of a multiprocessor system is required to behave as if there were logically a single DTLB and a single ITLB.

All TLBs in a multiprocessor system are required to broadcast purges, except PDTLBE and PITLBE, to all other TLBs. The originating processor's purge instruction suspends until all target processors complete the purge.

TLB Operation Requirements

Software may rely on the existence of particular translations in the TLB only in certain situations. The following describes the situations in which software may rely upon the fact that a specific translation will continue to exist in the TLB. In these situations, software may make virtual accesses using the relied-upon translation, and no TLB miss fault will occur (including non-access TLB miss faults).

1. When an instruction takes one of the following interruptions, the associated data address translation

will remain in the DTLB, and is termed the relied-upon translation.

Intr. No.	Interruption
18	data memory protection/unaligned data reference trap
19	data memory break trap
20	TLB dirty bit trap
21	page reference trap
22	assist emulation trap
26	data memory access rights trap
27	data memory protection ID trap
28	unaligned data reference trap

The translation will continue to remain in the DTLB, meaning no data TLB miss fault will occur on virtual data accesses which use this translation, for as long as software meets the following constraints:

- No virtual data references are made to pages other than the page corresponding to the relied-upon translation.
- The execution stream does not contain nullified instructions which, had they not been nullified, would have made virtual data references to pages other than the page corresponding to the relied-upon translation.
- No memory management instructions other than LPA are executed. (See "Memory Management Instructions (Mem_Mgmt)" on page D-5 for a list of memory management instructions.)
- No purge TLB instructions which would purge the relied-upon translation are executed by other processors in a multiprocessor system.
- No virtual instruction references are made.
- No DIAGNOSE instructions are executed.
- No attempt is made to execute undefined instructions.

PROGRAMMING NOTE

Software may rely upon this translation in order to improve performance in handling the above-mentioned traps. For example, the absolute address which corresponds to the virtual address used in the trapping instruction can be determined by using this code sequence:

```
LPA     x(s,b),t
```

Because no TLB miss fault can occur, the interruption handler need not incur the overhead of making itself interruptible.

---

2. If the PSW Q-bit is 1, and is set to 0 by a RESET SYSTEM MASK or MOVE TO SYSTEM MASK

instruction, the instruction address translation used to fetch the RSM or MTSM instruction will continue to remain in the ITLB, and is termed the relied-upon translation. No instruction TLB miss fault will occur on virtual instruction accesses which use this translation for as long as software meets the following constraints:

- The RSM or MTSM instruction which sets the PSW Q-bit to 0 (the **clearing** RSM or MTSM) is preceded by another RSM, SSM, or MTSM instruction which does not affect the PSW Q-bit, and which appears at least 8 instructions prior.
- The instructions between the initial RSM, SSM, or MTSM instruction and the clearing RSM or MTSM do not include any memory management instructions, virtual data references, or instruction references to pages other than the code page containing the clearing RSM or MTSM instruction.
- The clearing RSM or MTSM instruction is not within 8 instructions of a page boundary.
- No virtual data references are made.
- The execution stream does not contain nullified instructions which would have made virtual data references had they not been nullified.
- No FLUSH INSTRUCTION CACHE instructions are executed with the PSW D-bit equal to 1.
- No memory management instructions are executed. (See Appendix D, “Operation Codes” for a list of memory management instructions.)
- No purge TLB instructions which would purge the relied-upon translation are executed by other processors in a multiprocessor system.
- No instruction references are made to pages other than the code page containing the clearing RSM or MTSM instruction.
- No DIAGNOSE instructions are executed.
- No undefined instructions are attempted to be executed.
- No instructions are executed which are followed, within 8 words, by a branch instruction, a memory management instruction, a DIAGNOSE instruction, an undefined instruction, or a page boundary.

PROGRAMMING NOTE

Software may rely upon this instruction translation in order to improve performance in process dispatch. For example, in this code sequence:

```
SSM      0,gr0    ; initial RSM, SSM or MTSM
LDW               ; set up process state
.                 ; must be at least 7 instructions
.                 ; between the system mask instrs
.
LDW
RSM      8,gr0    ; set PSW Q-bit to 0
MTCTL    reg1,cr20; set up IIASQ
```

```
MTCTL      reg2,cr20
MTCTL      reg3,cr21; set up IIAOQ
MTCTL      reg4,cr21
LDW                   ; set up last of process state
.
.
.
LDW
RFI                   ; dispatch process
```

Because no TLB miss fault can occur, the interruption handler need not incur the overhead of disabling code translation just prior to process dispatch. Note that the LDW instructions in this sequence must use absolute addresses. (Use absolute loads, or do these with the PSW D-bit equal to 0.)

Data Cache Move-In

Data lines are brought into the cache only as a result of references, and only if the page containing the reference is cacheable. Except where noted, a data reference may move in all of the lines on the cacheable page containing the reference. The following actions constitute a data reference, and may cause move-in to the data cache:

- Execution of a load, store, or semaphore instruction
- Interruption of a load, store, or semaphore instruction by any interruption except the ones listed below:

Intr. No.	Interruption
6	instruction TLB miss fault
7	instruction memory protection trap
8	illegal instruction trap
10	privileged operation trap
15	data TLB miss fault
18	data memory protection/unaligned data reference trap
19	data memory break trap
20	TLB dirty bit trap
21	page reference trap
26	data memory access rights trap
27	data memory protection ID trap
28	unaligned data reference trap

NOTE

Because protection is checked (interruptions 10, 18, 26, and 27), the reference cannot bring in any data which could not have been accessed. This does not apply, however, to absolute accesses.

- A load or store instruction which is left at the front of the interruption queues because of a prior instruction which took a group 4 interruption, provided that the load or store would not have taken any of the above interruptions (6, 7, 8, 10, 15, 18, 19, 20, 21, 26, 27, 28).

Data items which would have been referenced by a nullified load, store, or semaphore instruction are not moved in.

The instructions LDWAX, LDWAS, and STWAS are exceptions to the general rule that a data reference may cause all of the lines in the page containing the reference to be moved in. These instructions can cause only the referenced line to be moved into the data cache.

In addition to the above rules, the following properties determine if the cache line associated with a memory reference may be moved into the data cache or a combined data and instruction cache:

- Whether the memory reference is virtual or absolute.
- Whether or not the optional TLB U (Uncacheable) bit is implemented.
- The state of the U-bit, if implemented.
- Whether the reference is made to a page in the memory or the I/O address space.

Table 3-2 specifies the rules under which the above properties determine if a memory reference may be moved into the data cache.

TLB U-bit	absolute access		virtual access	
	memory address space	I/O address space	memory address space	I/O address space
not implemented	line may be moved in	line must not be moved in	line may be moved in	line must not be moved in
0	line may be moved in	line must not be moved in	line may be moved in	undefined operation
1	line may be moved in	line must not be moved in	line must not be moved in	line must not be moved in

Table 3-2. Data Cache Move-In Rules

Instruction Cache Move-In

Instructions are brought into the instruction cache, or combined data and instruction cache, only as a result of references. Except where noted, an instruction reference may move in all of the lines on the page containing the reference, as well as all of the lines on the next sequential page, provided access and protection check requirements are met for each page. (If the PSW C-bit is 0, then the next sequential page is the next sequential physical page. Otherwise, if the PSW C-bit is 1, then it is the next sequential virtual page.) The following actions constitute an instruction reference, and may cause move-in to the instruction cache:

- Execution of an instruction
- Execution of a nullified instruction which would not have taken any of the following interruptions had it not been nullified. This action can cause only those lines on the page containing the instruction to be moved in

Intr. No.	Interruption
6	instruction TLB miss fault
7	instruction memory protection trap

NOTE

Because protection is checked (interruption 7) the reference cannot bring in any instructions which could not be executed. This does not apply, however, to absolute accesses.

- Execution of a branch can cause all of the lines on the page containing the target of the branch to be moved in.
- Execution of a branch to a target instruction which is the last instruction on a page, followed by an instruction which traps (in the branch delay slot), can cause all of the lines on the page containing the target instruction, as well as all of the lines on the next sequential page to be moved in.
- Interruption of an instruction by any interruption except for the ones listed below:

Intr. No.	Interruption
6	instruction TLB miss fault
7	instruction memory protection trap

- A branch instruction which takes a group 4 interruption can cause all of the lines on the page containing the instruction which would have been branched to, to be moved in.

Instructions which would have been branched to by nullified or untaken branches are not moved in.

If lines on the next sequential page are to be moved in, that page must meet the normal access and protection check requirements.

PROGRAMMING NOTE

If a data page immediately follows an instruction page, it is possible that the entire data page may have been moved into the instruction cache because of these move-in rules. Software must be aware of this fact and flush **both the instruction and the data** caches in order to remove the data page from the cache.

To insure that lines from a particular data page are not moved into the instruction cache, software must not make absolute instruction references to the immediately preceding physical page, and must prevent virtual instruction references to the data page. This latter can be accomplished by not assigning execute access rights to the data page. If, however, the data page is assigned execute access rights, and virtual instruction references are prevented by some other means, then software must also prevent virtual instruction references to the immediately preceding page.

In addition to the above rules, the following properties determine if the cache line associated with a instruction reference may be moved into the instruction cache or a combined data and instruction cache:

- If the reference is made to a page in the memory address space, the referenced line may be moved into the cache.
- If the reference is made to a page in the I/O address space, the referenced line must not be moved into the cache.
- If there is a combined instruction and data TLB, then an instruction fetch from a page with its U-bit 1 is undefined.

Flushing

A flush cache, purge cache, or purge TLB instruction to a page stops (disables) any subsequent move-in operations to that page until another reference to that page is made. In a multiprocessor system, these instructions stop any subsequent move-in operations to that page on all processors until another reference to that page is made.

Once a reference has been made, and a line could have been brought into a cache, the only way software can insure that the line has been removed from the cache is to flush or purge it and execute a SYNC instruction, or to flush the entire cache with flush-entry instructions and execute a SYNC instruction. Once a line is referenced, even if it is subsequently forced out of the cache by other references, the cache system can move it in again without another reference, until the line is flushed.

4 Flow Control and Interruptions

Introduction

The architecture defines a model in which the flow of control passes to the next sequential instruction in memory unless directed otherwise by branch instructions, nullification of instructions, or interruptions. The architecture requires that a CPU program appear to execute instructions in the order in which they appear although in reality the order may be changed internally. The instruction execution model described in this chapter provides a logical view of the steps involved in instruction execution. The sections on nullification, branching, and interruptions show how flow control can be altered during the course of program execution.

Instruction Execution

Instruction flow involves calculating the address of the current instruction and then fetching, decoding, and executing that instruction. This process involves performing the sequence of events listed below regardless of the instruction type. (Although these events are listed in sequence, many of them may occur in parallel. It is only necessary that they appear to be logically sequential.) In the description that follows, the values of the PSW bits are the values that exist before the instruction is executed. Changes to the PSW bits only affect instructions after the current instruction. This flow of instruction execution is shown in Figure 4-1.

1. If the PSW M-bit is 0, then high priority machine checks (HPMCs) may occur.
2. The processor checks for group 2 interruptions:
   a. A power failure interrupt that is not masked by the PSW I-bit .
   b. A recovery counter trap. This trap is enabled when the PSW R-bit is 1 and the most significant bit of the recovery counter is 1.
   c. An external interrupt or low-priority machine check, both of which are unmasked by the PSW I-bit.
   d. A performance monitor interrupt that is not masked by the PSW F-bit.
3. Depending on the state of the PSW N-bit, one of two events occur:
   a. If the current instruction is **nullified** (the PSW N-bit is 1), group 3 interruptions must not be taken. The instruction address queue is advanced and the back of the queue is written with the new front element + 4. The privilege level is the same as the new front element. The PSW X-bit, Y-bit, Z-bit, N-bit, and B-bit are set to 0.
   b. If the current instruction is **not nullified** (the PSW N-bit is 0), then the instruction is fetched using the front elements of the instruction address (IA) queues. If a group 3 interruption occurs during execution, the processor rolls back the effect of the current instruction by restoring the beginning state and takes the interruption. If the PSW C-bit is 1, virtual address translation of

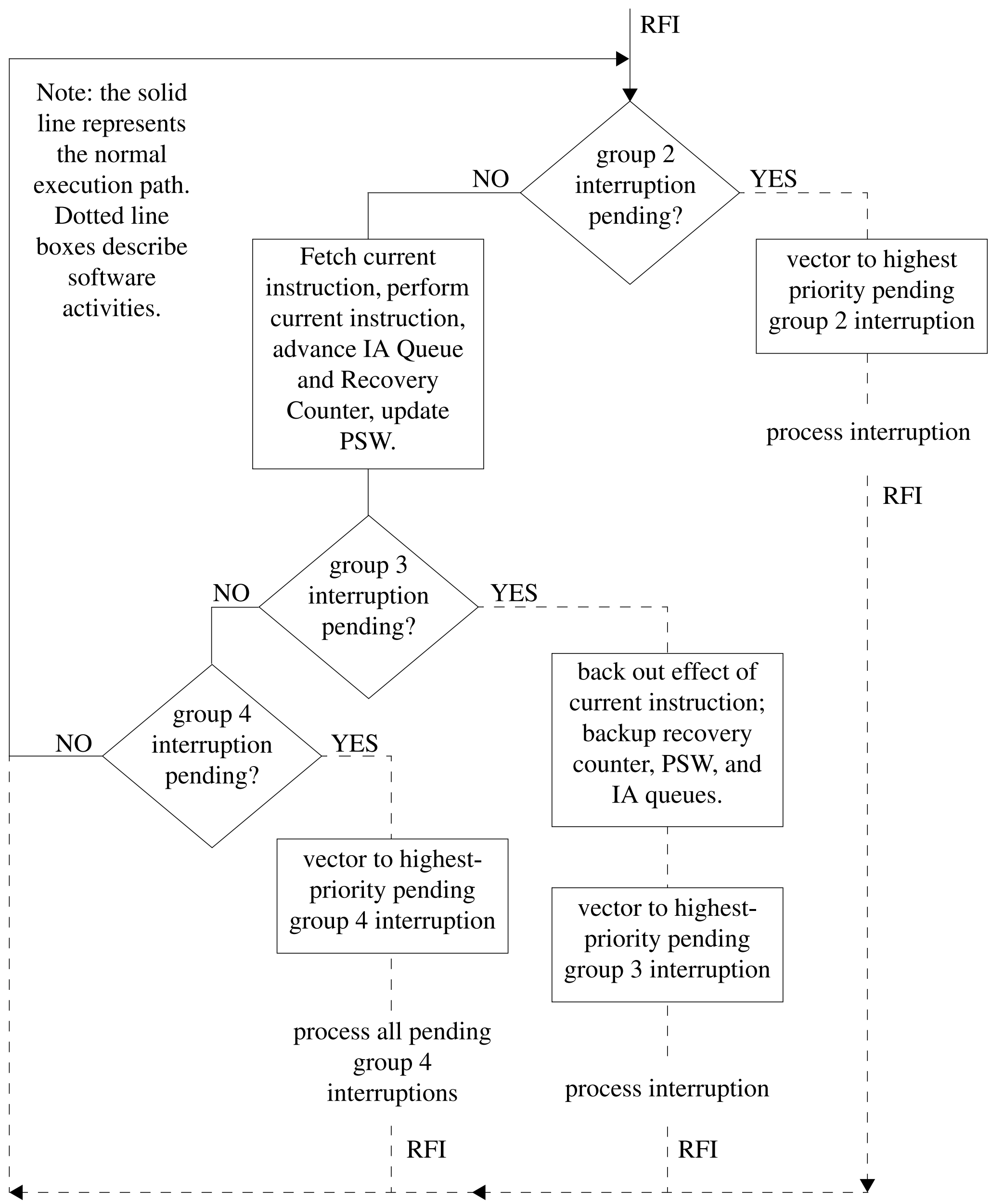


Figure 4-1. Interruption Processing

the instruction address is performed. The PSW P-bit enables protection checking. On a split TLB system, the instruction TLB is used for instruction address translation. The fetching of the current instruction may result in an instruction TLB miss fault/instruction page fault or an instruction memory protection trap.

The Recovery Counter is decremented if the PSW R-bit is 1. The current instruction is executed and the PSW X-bit, Y-bit, and Z-bit are set to 0. If the next instruction is to be nullified, the PSW N-bit is set to 1, and the instruction address queues are updated. The nature of that update depends on whether the current instruction is a taken branch:

- For a taken branch: the instruction address queues are advanced, the back of the queue is loaded with the target address including the privilege level which is computed by the branch instruction, and the PSW B-bit is set to 1.
- For a branch that is not taken: the instruction address queues are advanced, the back of the instruction address offset queue is written with the new front element + 4, the privilege level of the back element is set the same as the new front element, and the PSW B-bit is set to 0.
- If the current instruction is a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction, the IA queues and the PSW are updated with the new values and the following instruction is executed based on these new values.

4. Group 4 traps are handled after execution is complete. If the new privilege level is lower than that of the just completed instruction and the PSW L-bit was 1, a lower-privilege transfer trap is taken. If the new privilege level is higher than that of the just completed instruction and the PSW H-bit was 1, a higher-privilege transfer trap is taken. The term "new privilege" level refers to the privilege level at which the following instruction executes.

   If neither transfer trap is taken, the instruction just completed is a taken branch, and the PSW T-bit was 1, then a taken branch trap occurs.

Atomicity of Storage Accesses

All load and semaphore instructions access storage atomically. All store instructions, except STORE BYTES SHORT when accessing the I/O address space, access storage atomically. For example, a double-word load instruction executing on one processor concurrently with a double-word store instruction to the same address executing on another processor will receive either the entire old value or the entire new value.

Ordering of Accesses

Accesses to the address space (both to memory and I/O) through load, store, and semaphore instructions are **strongly ordered**. This means that accesses appear to software to be done in program order. In multiprocessor systems, accesses by a given processor must appear to that processor, as well as to all other processors in the system, to be done in program order.

If processor A executes a load instruction and receives the value stored by a store instruction executed on processor B, then processor A is said to have **observed** the store by processor B.

If a store by a processor is observed by another processor or by an I/O module, then the store is said to be **performed**.

NOTE

An example of the observability of the ordering of accesses can be seen by considering two processes, A and B, running concurrently on a multiprocessor system.

process A	process B
load x	load y
store y	store x

If process B observes A's store to y, then process B may rely on the fact that its store to x will not affect the value seen by process A.

This strong ordering applies to I/O accesses as well. For example, the sequence of an I/O access followed by a memory access appears to be performed in program order.

Cache flush operations are, however, not strongly ordered. Flush operations may be delayed or held pending, and a sequence of flush operations may be executed in any order. The SYNC instruction is used to ensure ordering. The SYNC instruction enforces the ordering of only those accesses caused by the instructions executed on the same processor which executes the SYNC instruction. After executing a SYNC instruction, any pending flush operations are completed before performing any subsequent load, store, semaphore, flush, or purge instructions. Flush operations do get performed, however, prior to a subsequent purge operation to the same cache line, to prevent loss of data. In multiprocessor systems, to allow non-privileged code to do cache management, system software must execute a SYNC instruction when switching processes.

Cache purge operations are strongly ordered, but only with respect to accesses to the same cache line. A SYNC instruction ensures ordering with respect to subsequent accesses.

PROGRAMMING NOTE

Software which executes Purge Data Cache instructions to memory locations which are shared with other processes must be aware of a potential problem. Consider two processes sharing a memory location **x** which is protected by a semaphore *s*.

process A on Processor 1	process B on Processor 2	note
LDCW *s*		A acquires semaphore
PDC *x*		A executes purge
SYNC		Force completion of purge
STW *s*		A releases semaphore
	LDCW *s*	B acquires semaphore
	STW *x*	

In the absence of the SYNC instruction, it is possible that process B stores to *x* before the purge is actually completed.

MTCTL and MFCTL instructions involving the EIR and the EIEM must appear to preserve program order.

Modification of resources which affect data access take effect immediately. Acknowledgement of a data TLB purge request from another processor must not be made until after the purge has logically been performed. Data access resources include Protection Identifier Registers, PSW, and TLB entries.

Interrupts must be masked immediately following a MTCTL to the EIEM register that masks interrupts or an RSM or MTSM that sets the PSW I-bit to 0.

Completion of Accesses

PA-RISC processors are inherently asynchronous and software may not rely on instruction timing for correct operation. Implementations are permitted to execute instructions out of order and need only preserve the appearance of sequential execution. For example, in the absence of other constraints which would force execution, flush and purge operations may be indefinitely delayed. To insure that progress is made, however, the following requirements must be met.

- Instruction streams must make forward progress. This means that any operation, on which an instruction stream is dependent, must be performed in some finite period.
- All load and store operations to the I/O space must be performed in some finite period.
- Execution of a SYNC instruction forces all prior flush operations from the same instruction stream to be performed in some finite period.

For performance and testability reasons, it is occasionally necessary to know when an access to I/O space has completed. This would not normally be possible due to the asynchronous nature of execution. In order to provide this capability, the following two special sequences are defined. These sequences place additional requirements on implementations for the completion of accesses. When these code sequences are used, the additional completion requirements hold.

When this code sequence is executed, the instruction labeled 'access A completed on bus' is not executed, and the source registers not read until after the LDW (or STW) labeled 'access A' has completed on the bus.

```
LDW (or STW)       from (to) I/O space                  ; access A
SYNC
LDW                from I/O space, but not to GR 0
(at least seven instructions)
Instruction                                             ; access A completed on bus
```

When this code sequence is executed, the instruction labeled 'access B completed on module' is not executed, and the source registers not read until after the STW instruction labeled 'access B' has completed on the I/O module.

STW	to I/O space	; access B
LDW	from the same I/O space module	
SYNC		
LDW	from I/O space, but not to GR 0	
(at least seven instructions)		
Instruction		; access B completed on module

Instruction Pipelining

The architecture permits implementations to prefetch up to seven instructions from the cache (including branch prediction) beyond the instruction currently executing. Instructions may modify resources which affect instruction fetch on the machine they are executing on. Instruction fetch resources include protection identifier registers, the PSW, and TLB entries. When such an event takes place, it affects instructions that are fetched 8 instructions later (at the latest), or after the next RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction, whichever occurs first.

Instructions may also modify resources on other processors in a multiprocessor system, which affect the instruction fetch of the target processors. When such an event takes place (the modification of the resource is acknowledged), it affects instructions that are fetched, on the target processors, after they have finished executing 8 instructions (at the latest) except as noted below.

When a processor executes an instruction which purges an instruction TLB entry in other processors, the target processors must acknowledge completing the purge. The target processors may not complete a move-in, which was initiated using the purged translation, after acknowledging the removal. Acknowledgement of a data TLB purge request from another processor must not be made until after the purge has logically been performed.

Modification of code, while discouraged, may be performed using the following protocol:

1. Modify the code in the data cache.
2. Flush the modified code from the data cache.
3. Issue a SYNCHRONIZE CACHES instruction to ensure the flush is completed and subsequent move-in will observe the memory version.
4. Flush the location of the modified code from the instruction cache.
5. Issue a SYNCHRONIZE CACHES instruction to ensure the flush is completed.
6. Delay at least an additional seven instructions or execute a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction.

In a multiprocessor system, software must ensure that no other processor is executing code that is in the process of being modified.

Nullification

A nullified instruction is an instruction that is skipped over. It has no effect on the machine state (except that the IA queues advance and the B-bit, N-bit, X-bit, Y-bit, and Z-bit in the PSW are set to 0). The recovery counter **is not** decremented for a nullified instruction. Nullified instructions do not take group 3 interruptions (although they may take group 1, 2, or 4 interruptions).

All branch instructions and computational instructions can nullify the execution of the following instruction. For branch instructions, nullification can be specified explicitly. In the case of computational instructions, nullification is performed conditionally based on the outcome of a test.

Branching

Branches are another way of altering the flow during program execution. The architecture provides both unconditional and conditional branch instructions. Unconditional branch instructions always branch to the specified target. Conditional branch instructions first perform some operation (move, compare, add, or bit test) and then branch if the outcome of the specified condition is met.

Concept of Delayed Branching

All branch instructions exhibit the delayed branch feature. This implies that the major effect of the branch instruction, the actual transfer of control, occurs one instruction after the execution of the branch. As a result, the instruction following the branch (located in the **delay slot** of the branch instruction) is executed before control passes to the branch destination. The concept of delayed branching is illustrated in Figure 4-2.

Execution of the delay slot instruction, however, may be skipped ("nullified") by setting the "nullify" bit in the branch instruction to 1.

PROGRAM SEGMENT			
Location	Instruction		Comment
100	STW	r3, 0(r6)	; non-branch instruction
104	BLR	r8, r0	; branch to location 200
108	ADD	r7,r2, r3	; instruction in delay slot
10C	OR	r6,r5, r9	; next instruction in linear code sequence
.	.		
.	.		
.	.		
200	LDW	0(r3), r4	; target of branch instruction

EXECUTION SEQUENCE			
Location	Instruction		Comment
100	STW	r3, 0(r6)	;
104	BLR	r8, r0	;
108	ADD	r7,r2, r3	; delay slot instruction is executed before
200	LDW	0(r3), r4	; execution of target instruction

Figure 4-2. Delayed Branching

Conditional and Unconditional Branches

There are two kinds of branches - **unconditional** branches, and **conditional** branches. **Unconditional** branches are not dependent on the outcome of any test operation. **Conditional** branches provide a mechanism to branch based on the outcome of a specified test. When the test is successful, the conditional branch is said to be **taken**, and, when the test is unsuccessful, the conditional branch is said to be **not-taken**. Unconditional branches are always **taken**.

Branching and Spaces

Certain branch instructions can only branch to a location within the same space, while others can branch to another space. Branches within the same space are referred to as **intraspace** or **local** branches. Branches to another space are referred to as **interspace** or **external** branches.

Target Address Computation

In systems which support virtual addressing, the target of a branch instruction, just like any instruction address, consists of a space ID and an offset. In Level 0 systems, the target of a branch instruction consists of an offset only.

The space ID of the target of an intraspace branch is not changed by the branch instruction. A space ID calculation is performed for interspace branches. The offset portion of the address is computed in one of several ways based on the particular branch instruction. When a displacement is added to the current instruction address offset, the branch is called **IA relative**. When a general register is used as a base offset, it is called **base relative**. Also, if the displacement is a fixed value that is known at compilation, it is known as **static** displacement. If the value is computed during the course of program execution, and is read from a general register, it is known as **dynamic** displacement.

For interspace branches, the space ID of the target address is always specified in a space register, and is copied into the IASQ when the branch is performed. The offset of the target is computed by adding a 17-bit signed word displacement to the base register specified in a general register. The two rightmost bits in the base register denote the new privilege level and are ignored during the offset computation. Also, the 17-bit signed word displacement is shifted left by two before adding to the base register. Interspace branches are always base relative.

In the case of intraspace branches, the space ID is not changed by the branch. The offset of the target, however, can be computed in one of three ways. For IA relative branches with static displacement, a 12-bit or 17-bit signed word displacement is shifted left by two and added to the current instruction address offset plus eight. For IA relative branches with dynamic displacement, the value specified in the index register is shifted left by three and added to current instruction address offset plus eight. For base relative branches with dynamic displacement, the value specified in the index register is shifted left by three and added to the value in the specified base register.

It should be noted that for IA relative branches, the target is computed from the current instruction by adding a displacement or an index value. Since the instruction in the delay slot must be executed if it is not nullified, an additional value of eight is added in the offset computation to arrive at the target correctly. This is done to ensure that a branch with a displacement of zero will branch to the instruction following the delayed instruction. Also, this helps users build case tables immediately following the delay slot instruction.

Privilege Level Changes

Branch instructions may change the privilege level depending on the type of branch performed. Since privilege levels are determined by the two rightmost bits in the offset part of the instruction address, privilege level changes are a function of the offset computation.

Unconditional branches can be IA relative or base relative. IA relative branches compute the target address relative to their own IA value, and since the two rightmost bits are unchanged, the privilege level of the branch instruction and the target are the same. Base relative branches (intraspace or interspace) may lower the privilege level if the two rightmost bits in the base register are of a lower privilege level. The GATEWAY instruction is an IA relative branch, however, it behaves differently for privilege computation. It can promote the privilege level to that specified by the two rightmost bits of the **type** field, located in the TLB entry for the page from which the GATEWAY instruction is fetched. In Level 0 systems, execution of a GATEWAY instruction causes the privilege level to be promoted to 0.

Conditional branch instructions always perform IA relative branches and the privilege level of the target instruction and the branch instruction is the same.

The change of privilege level takes effect at the target instruction except in the case of the GATEWAY. Change of privilege level for a GATEWAY instruction can occur either at the delay slot or the target instruction.

PROGRAMMING NOTE

Since a branch instruction may be executed in the delay slot of another branch instruction, an interesting case arises because of the way the privilege level changes are defined to take effect.

Consider the case where a taken IA relative branch is placed in the delay slot of a base relative branch that lowers the privilege level of its target instruction. First, the base relative branch will execute and schedule change of privilege level for its target. Then, in the delay slot, the IA relative branch will execute and it will schedule its target to execute at the same privilege level as its own. Then, the target of the base relative branch will execute at the new (demoted) privileged level. The next instruction, however, which is the target of the IA relative branch, will have the same privilege level as that of the IA relative branch, and thus will cause the privilege level to be restored to the original (higher) value as shown in the following:

PROGRAM SEGMENT			
Location	Instruction		Comment
100	STW	r7, 0(r8)	; non-branch instruction
104	BV	r0(r7)	; branch vectored to 200 and change priv -> 2
108	BLR	r4, r0	; IA relative branch to location 400
10C	ADD	r2,r6, r9	; next instruction in linear code sequence
.	.		
.	.		
.	.		
200	LDW	0(r3), r11	; target of branch vectored instruction
.	.		
.	.		
.	.		
400	LDW	0(r15), r4	; target of IA relative branch instruction
404	STW	r4, 0(r18)	

EXECUTION SEQUENCE			
Location	Instruction		Comment
100	STW	r7, 0(r8)	; priv = 0
104	BV	r0(r7)	; priv = 0
108	BLR	r4, r0	; priv = 0
200	LDW	0(r3), r11	; priv = 2 decreased by branch vectored instr
400	LDW	0(r15), r4	; priv = 0 changed back by IA relative branch
404	STW	r4, 0(r18)	; priv = 0

Linkage

Linkage is provided in certain branch instructions to allow a return path for procedure calls. The return point is four bytes after the following instruction. Since the execution of all branches is followed by the execution of the instruction in the delay slot (or null if nullified), it should be noted that the return point is always specified as four bytes after the following instruction and **not** eight bytes after the BRANCH AND LINK instruction. When the following instruction is not spatially sequential, then four bytes after the following instruction is not the same as eight bytes after the BRANCH AND LINK instruction.

The linkage mechanism is available for both intraspace and interspace branches. For intraspace branches, the offset of the return point is saved in the specified target register GR t. For interspace branches, the offset of the return point is always saved in GR 31, and the space ID of the return point is saved in SR 0 (except in Level 0 systems where SR 0 is nonexistent).

Conditional Branching and Nullification

When nullification is specified by a conditional branch instruction, the effect of nullification depends on the direction of the branch. This maximizes useful work done during loops and "if-then" constructs.

For a backward conditional branch, the following instruction is nullified only when the backward conditional branch is **not** taken. For forward conditional branches, the following instruction is nullified only when the forward conditional branch **is** taken. For unconditional branches, if nullification is specified, the following instruction is nullified **independent** of the direction of branch.

Branching and Address Queues

The concept of delayed branching makes it necessary to maintain the instruction address (IA) in a pair of two element queues. The **front** elements point to the currently executing instruction and the **back** elements point to the following instruction that will be executed. The term **next** refers to the Space Identifier and the offset of the next instruction address, which will enter the back elements of the queues when the queues are updated. The queues are said to be updated when the **back** elements become the **front** and **next** become the **back** elements.

For taken branches, the IA queues get updated with the address of the branch target. Both the word offset and the privilege level are updated. IAOQ_Next receives the value of the branch target offset. For not-taken branches, IAOQ_Next gets IAOQ_Back + 4. The privilege level is obtained from the back element of the queue. For external branches (BE and BLE), IASQ_Next gets the value of the branch target Space ID. Otherwise, IASQ_Next receives the content of IASQ_Back. Figure 4-3 shows how the IA queues are updated, using a pseudo-code representation.

Instruction Address Offset Queue (IAOQ)
IAOQ_Front $\leftarrow$ IAOQ_Back;
IAOQ_Back $\leftarrow$ IAOQ_Next;
if (taken branch)
IAOQ_Next $\leftarrow$ Branch target offset;
else
IAOQ_Next $\leftarrow$ IAOQ_Back + 4;

Instruction Address Space Queue (IASQ)
IASQ_Front $\leftarrow$ IASQ_Back;
IASQ_Back $\leftarrow$ IASQ_Next;
if (BE or BLE)
IASQ_Next $\leftarrow$ Branch target Space ID;
else
IASQ_Next $\leftarrow$ IASQ_Back;

Figure 4-3. Updating Instruction Address Queues

Consider the situation shown in Figure 4-4; a taken branch instruction, I2, is executed in the delay slot of a preceding taken branch, I1. When this occurs, the first branch I1 schedules its target instruction, I3, to execute after I2, and the second branch, I2, schedules its target instruction, I4, to execute after I3. The net effect is the out-of-line execution of I3, followed by the execution of I4. Also, if I3 were to be a taken branch, its target, I5, would execute after I4, and I4 would also have been executed out of its spatial context.

Note that if nullification is specified in the instruction currently executing, the nullification affects the instruction to be executed next, regardless of whether that instruction immediately follows the currently executing instruction in the linear code sequence. For example, if the instruction, I2, specified nullification of the next instruction, then I3 would have no effect except that the PSW X-bit, Y-bit, Z-bit, N-bit, and B-bit would be set to 0.

Traps Associated with Branches

Branch instructions may cause various traps based on the value of PSW bits. If the PSW T-bit is 1, and a branch is taken, a taken branch trap occurs. This trap may be used for the purposes of debugging. If the PSW H-bit is 1, and a branch instruction raises the privilege level, a higher-privilege transfer trap occurs. If the PSW L-bit is 1, and a branch instruction lowers the privilege level, a lower-privilege transfer trap occurs.

Restrictions in Branching

It is illegal for a GATEWAY instruction to execute in the delay slot of a taken branch instruction. The PSW B-bit ensures that this sequence is not permitted. Whenever a branch is taken, the PSW B-bit is set to 1 and, if the next instruction is a GATEWAY, an illegal instruction trap occurs.

PROGRAM SEGMENT				
Location	Instruction		Comment	
100	STW	r7, 0(r8)	; non-branch instruction	
104	BV	r0(r7)	; branch vectored to location 200	I1
108	BLR	r4, r0	; IA relative branch to location 400	I2
10C	ADD	r2,r6, r9	; next instruction in linear code sequence	
.	.			
.	.			
.	.			
200	LDW	0(r3), r11	; target of branch vectored instruction	I3
204	ADD	r11,r12, r14	;	
.	.			
.	.			
.	.			
400	LDW	0(r15), r4	; target of IA relative branch instruction	I4
404	STW	r4, 0(r18)	;	I5

EXECUTION SEQUENCE				
Location	Instruction		Comment	
100	STW	r7, 0(r8)	;	
104	BV	r0(r7)	; schedules execution at 200 after delay instr	I1
108	BLR	r4, r0	; schedules execution at 400 after delay instr	I2
200	LDW	0(r3), r11	; target of first branch executes out of context	I3
400	LDW	0(r15), r4	; target of second branch (is a non-branch)	I4
404	STW	r4, 0(r18)	; next instruction is in linear code sequence	I5

Figure 4-4. Branch in the Delay slot of a Branch

Interruptions

Interruptions are anomalies that occur during instruction processing, causing the flow control to be passed to an interruption handling routine. In the process, certain processor state is saved automatically by the hardware. Upon completion of interruption processing, a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction is executed, which restores the saved processor state, and the execution proceeds with the interrupted instruction.

From the viewpoint of response to interruptions, the processor behaves as if it were not pipelined. That is, it behaves as if a single instruction is fetched and executed, and any interruption conditions raised by that instruction are handled at that time. If there are none, the next instruction is fetched, and so on.

Faults, traps, interrupts, and **checks** are the different classes of interruptions that may happen during instruction processing. Definitions of the four classes of interruptions are as follows:

Fault	The current instruction requests a legitimate action which cannot be carried out due to a system problem, such as the absence of a page from main memory. After the system problem has been corrected, the faulting instruction will execute normally. Faults are synchronous with respect to the instruction stream.
Trap	Traps include two sorts of possibilities: either the function requested by the current instruction cannot or should not be carried out, or system intervention is desired by the user before or after the instruction is executed. Examples of the first type include arithmetic operations that result in signed overflow and instructions executed with insufficient privilege for their intended function. Such instructions are normally not re-executed. Examples of the second type include the debugging support traps. Traps are synchronous with respect to the instruction stream.
Interrupt	An external entity (e.g., an I/O device or the power supply) requires attention. Interrupts are asynchronous with respect to the instruction stream.
Check	The processor has detected an internal malfunction. Checks can be either synchronous or asynchronous with respect to the instruction stream.

All four classes of interruptions are handled in the same way. The interruptions are categorized into four groups based on their priorities:

Group 1:	1	High-priority machine check
Group 2:	2	Power failure interrupt
	3	Recovery counter trap
	4	External interrupt
	5	Low-priority machine check
	29	Performance monitor interrupt
Group 3:	6	Instruction TLB miss fault/Instruction page fault
	7	Instruction memory protection trap
	30	Instruction debug trap
	8	Illegal instruction trap
	9	Break instruction trap
	10	Privileged operation trap
	11	Privileged register trap
	12	Overflow trap
	13	Conditional trap
	14	Assist exception trap
	15	Data TLB miss fault/Data page fault
	16	Non-access instruction TLB miss fault
	17	Non-access data TLB miss fault/Non-access data page fault
	26	Data memory access rights trap
	27	Data memory protection ID trap
	28	Unaligned data reference trap

	18	Data memory protection trap/Unaligned data reference trap
	19	Data memory break trap
	20	TLB dirty bit trap
	21	Page reference trap
	31	Data debug trap
	22	Assist emulation trap
Group 4:	23	Higher-privilege transfer trap
	24	Lower-privilege transfer trap
	25	Taken branch trap

The interruption numbers in the above list are the individual vector numbers that determine which interruption handler is invoked for each interruption. The group numbers determine when the particular interruption will be processed during the course of instruction execution. The order the interruptions are listed within each group (**not** the interruption numbers) determines the priority of simultaneous interruptions (from highest to lowest).

Interruption Handling

Interruption handling is implemented as a fast context switch (much simpler than a complete process swap). When an interruption occurs, the hardware takes the following actions:

1. The PSW in effect at the time of the interruption is saved in the IPSW. For group 2 and 3 interruptions, the saved PSW is the value at the beginning of execution. For group 4 interruptions, the saved PSW is the value after the execution of the instruction.
2. The defined bits in the PSW are set as follows:

E	Set to the value of the default endian bit.
M	Set to 1 if the interruption is a high-priority machine check; otherwise, set to 0.
all other bits	Set to 0 (interrupts are masked, absolute accesses are enabled, etc.).

3. IA information in the IIA queues is frozen (as a result of setting the PSW Q-bit to 0 in step 2 above).

   In order to enable restarting of instructions in the presence of delayed branching, at least two addresses must be saved, pointing to the next two instructions to be executed after returning from the interruption. The hardware, therefore, maintains IIA Space and IIA Offset queues, which have two elements and contain the addresses and privilege levels of these instructions. The IIA queues are kept up-to-date whenever the Q-bit in the PSW is 1. When an interruption is taken, the addresses of the pending instructions are preserved in the queues. The elements of the queues may be obtained by reading the IIASQ and IIAOQ registers (CRs 17 and 18, respectively).
4. The current privilege level is set to the highest privilege level (zero).
5. Information about the interrupting instruction is saved in the IPRs if the PSW Q-bit was 1 at the time of the interruption. If the PSW Q-bit was 0, the IPRs are unchanged. If the details of an instruction associated with the interruption are potentially useful in processing it, the instruction is

loaded into the Interruption Instruction Register (IIR or CR 19). If there is an address associated with the interruption, it is loaded into the Interruption Space and Interruption Offset registers (ISR or CR 20, and IOR or CR 21). When data address translation is not enabled, the ISRs contents are undefined. The value loaded into the IOR includes all 32 bits of the offset.

6. General registers 1, 8, 9, 16, 17, 24, and 25 are copied to the shadow registers.
7. Execution begins at the address given by:

   Interruption Vector Address + (32 * interruption_number)

   Interruption_number is the unique integer value assigned to that particular interruption. Vectoring is accomplished by performing an indexed branch into the Interruption Vector Table indexed by this integer. The Interruption Vector Table contains the starting points of execution of the interruption handling routines. The value in the Interruption Vector Address register (CR 14) must be aligned on a 2 Kbyte boundary.

PROGRAMMING NOTE

It is the responsibility of interruption handlers to unmask external interrupts (by setting the PSW I-bit to 1) as soon as possible, so as to minimize the worst-case latency of external interrupts.

Instruction Recoverability

When execution of instructions is interrupted, the minimal processor state that is required to be saved and restored is that necessary to correctly continue execution of the instruction stream following processing of the interruption. Processor state is defined to include any register contents, PSW bits, or other information that may affect the operation performed by an instruction. For example, if an interruption is taken immediately before an ADD instruction, its source registers must be restored, but its target register need not be (unless it is also one of the source registers).

Masking and Nesting of Interruptions

Disabling an interruption prevents it from occurring. The interruption does not wait until re-enabled. It is not kept pending. **Masking** an interruption does not prevent the recognition of a pending interruption condition, but delays the occurrence of the interruption until it is "unmasked".

The IA state is collected in the IIA queues only while the PSW Q-bit is 1; it is usually not possible to resume execution after an interruption which is taken while the PSW Q-bit is 0.

The machine state is saved in registers rather than memory when an interruption occurs, and interruption handlers must leave interruptions disabled until they have saved the machine state in memory. Once the machine state is saved, nested interrupts can be allowed.

Since it is desirable to catch hardware faults as soon as possible, interruption handlers should generally not mask high-priority machine checks. If a machine check occurs before the machine state has been saved, the interrupted process may need to be aborted. The occurrence of traps and faults within interruption handlers can be avoided by careful writing of the handlers.

Interruption Priorities

High-priority machine checks (which belong to Group 1) may occur and be processed at any time. They may be synchronous or asynchronous with instruction processing, may be associated with more than one instruction, and their precise meaning and processing is implementation dependent.

All interruptions other than high-priority machine checks are taken between instructions. Multiple simultaneous interruptions may occur because a number of instructions are capable of raising several synchronous interruptions simultaneously, and because certain interruptions are asynchronous with respect to the instruction stream.

Group 2 interruptions occur asynchronously with respect to the instruction stream.

Group 3 interruptions are synchronous with respect to the instruction stream and are signalled before completion of the instruction that produces them.

Group 4 interruptions are synchronous with respect to the instruction stream and are signalled either after completion of the instruction that causes them, or when a change in privilege level is about to happen.

Relative priorities are not assigned to the 32 external interrupts by the hardware. When multiple external interrupts occur simultaneously, software may select their order of service, based on the contents of EIR.

Return from Interruption

The RETURN FROM INTERRUPTION and RETURN FROM INTERRUPTION AND RESTORE instructions restore the PSW and instruction address queues. If the old PSW stored in IPSW (CR 22) has interruptions enabled (or unmasked), interruptions are re-enabled before execution of the first of the continuation instructions. The PSW Q-bit may be set to 1 reliably only by a RETURN FROM INTERRUPTION or RETURN FROM INTERRUPTION AND RESTORE instruction. An attempt to set the PSW Q-bit to 1 with a SET SYSTEM MASK or MOVE TO SYSTEM MASK instruction is an undefined operation.

The RETURN FROM INTERRUPTION AND RESTORE instruction does everything that the RETURN FROM INTERRUPTION instruction does, and in addition causes the values in the shadow registers to be copied to GRs 1, 8, 9, 16, 17, 24, and 25. Execution of a RETURN FROM INTERRUPTION AND RESTORE instruction leaves the contents of the shadow registers undefined.

Executing a RETURN FROM INTERRUPTION or a RETURN FROM INTERRUPTION AND RESTORE instruction with the PSW Q-bit 0 and the IPSW Q-bit 0 leaves the IPRs unchanged.

PROGRAMMING NOTE

Only those interruptions which are themselves uninterruptible may return from the interruption using the RFIR instruction. Interruption handling code which is interruptible must return from the interruption using the RFI instruction.

Fast interruption handling is achieved using shadow registers, since GRs 1, 8, 9, 16, 17, 24, and 25 are copied to the shadow registers on interruptions. In this example, it is assumed that at most seven general registers need to be used in the interruption handling routine.

using RFI	using RFIR
interrupt	interrupt
save GRs	<no save>
[process interrupt]	[process interrupt]
restore GRs	<no restore>
RFI	RFIR

Descriptions of Interruptions

Group 1 Interruptions

Name	**High-priority machine check** (1)
Cause	A hardware error has been detected that must be handled before processing can continue
Parameters	Implementation dependent
IIA Queue	Front – Implementation dependent Back – Implementation dependent
Notes	The actions taken when a hardware error is detected depend on the seriousness of the error. Damage extensive enough to prevent proper execution of instructions will halt the machine and generate an external indication of the occurrence of the check. Damage which allows a subset of the instructions to execute (e.g., inoperative TLB) generates a high-priority machine check interruption. This is maskable by setting the PSW M-bit to 1, so that machine checks within the machine check handler can be prevented. The causes of high-priority machine checks are implementation dependent, as is the means of controlling their reporting.

Group 2 Interruptions

Name	**Power failure interrupt** (2)
Cause	The machine is about to lose power
Parameters	none
IIA Queue	Front – Address of the instruction to be executed at the time of the interruption Back – Address of the following instruction
Notes	This interruption is masked and kept pending when the PSW I-bit is 0.

Name	**Recovery counter trap** (3)
Cause	Bit 0 of the recovery counter is 1 and the PSW R-bit is 1

Parameters	none
IIA Queue	Front – Address of the instruction to be executed at the time of the interruption Back – Address of the following instruction
Notes	The recovery counter can be used to log interruptions during normal operation and to simulate interruptions during recovery from a fault.
Name	**External interrupt** (4)
Cause	A module writes to the processor's IO_EIR or to the broadcast IO_EIR register, or the interval timer compares equal to its associated comparison register
Parameters	none
IIA Queue	Front – Address of the instruction to be executed at the time of the interruption Back – Address of the following instruction
Notes	Each external interrupt level has associated with it one bit in the External Interrupt Enable Mask Register (CR 15) and one bit in the External Interrupt Request Register (CR 23). When a module writes into the EIR register, the bit position corresponding to the value written is set to 1. For example if the value 5 is written, then bit 5 of the EIR register is set to 1. If the corresponding bit in CR 15 is 1 and the PSW I-bit is 1, an external interrupt is taken; otherwise, the interrupt is masked, and is kept pending. Interrupt handling software sets bits in the EIR to 0 by executing a MOVE TO CONTROL REGISTER instruction with the appropriate mask. If multiple sources can set the same interrupt, it is the responsibility of software to correctly respond to all of the interrupting sources.
Name	**Low-priority machine check** (5)
Cause	A hardware error has been detected which is recoverable and does not require immediate handling
Parameters	Implementation dependent
IIA Queue	Front – Address of the instruction to be executed at the time of the interruption Back – Address of the following instruction
Notes	Errors which have been detected and recovered from by hardware to the point that operation can continue in a degraded fashion are reported via the low-priority machine check interruption. This interruption is masked and kept pending when the PSW I-bit is 0. The causes of low-priority machine checks are implementation dependent, as is the means of controlling their reporting.
Name	**Performance monitor interrupt** (29)
Cause	An implementation-dependent event related to the performance monitor coprocessor requires software intervention
Parameters	Implementation dependent

IIA Queue — Front – Address of the instruction to be executed at the time of the interruption
Back – Address of the following instruction

Notes — This interruption is masked and kept pending when the PSW F-bit is 0.

Group 3 Interruptions

Name — **Instruction TLB miss fault/Instruction page fault** (6)

Cause — The instruction TLB entry needed by instruction fetch is absent, and if instruction TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table

Parameters — none

IIA Queue — Front – Address of the instruction causing the fault
Back – Address of the following instruction

Notes — Only if an instruction is to be executed can an instruction TLB miss fault occur. This interruption does not occur in Level 0 systems.

Name — **Instruction memory protection trap** (7)

Cause — Instruction address translation is enabled and the access rights check fails for an instruction fetch or instruction address translation is enabled, the PSW P-bit is 1, and the protection identifier checks fails for an instruction fetch

Parameters — none

IIA Queue — Front – Address of the instruction causing the trap
Back – Address of the following instruction

Notes — This interruption does not occur for absolute accesses.

Name — **Instruction debug trap** (30)

Cause — The debug SFU has detected that the instruction address matches the parameters set up in the SFU

Parameters — none

IIA Queue — Front – Address of the instruction causing the trap
Back – Address of the following instruction

Notes — This trap is disabled if the PSW G-bit is 0 or if the PSW Z-bit is 1. It does not occur if the debug SFU is not implemented.

Name — **Illegal instruction trap** (8)

Cause — An attempt is being made to execute an illegal instruction or to execute a GATEWAY instruction with the PSW B-bit equal to 1

Parameters	IIR – The illegal instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	Illegal instructions are the unassigned major opcodes. Unassigned sub-opcodes are undefined operations (undefined sub-opcodes may cause the illegal instruction trap). On some implementations, DIAGNOSE may be an illegal instruction.
Name	**BREAK instruction trap** (9)
Cause	An attempt is made to execute a BREAK instruction
Parameters	IIR – The BREAK instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Name	**Privileged operation trap** (10)
Cause	An attempt is being made to execute a privileged instruction without being at the most privileged level (priv= 0)
Parameters	IIR – The privileged instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	The list of privileged instructions is: IDTLBA, IDTLBP, IITLBA, IITLBP, PDTLB, PDTLBE, PITLB, PITLBE, MTSM, SSM, RSM, RFI, RFIR, LDWAX, LDWAS, STWAS, DIAG, LPA, LCI.
Name	**Privileged register trap** (11)
Cause	An attempt is being made to write to a privileged space register or access a privileged control register without being at the most privileged level (priv= 0)
Parameters	IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	This interruption may be caused by the MOVE TO SPACE REGISTER, MOVE TO CONTROL REGISTER, or MOVE FROM CONTROL REGISTER instructions.
Name	**Overflow trap** (12)
Cause	A signed overflow is detected in an instruction which traps on overflow
Parameters	IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction

Name	**Conditional trap** (13)
Cause	The condition succeeds in an instruction which traps on condition
Parameters	IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction

Name	**Assist exception trap** (14)
Cause	A coprocessor or special function unit has detected an exceptional condition or operation. An exceptional operation may include unimplemented operations or operands.
Parameters	IIR – For immediate traps, the SFU or coprocessor instruction that was executing when an exception is reported with a trap. It may or may not be related to the condition causing the exception. For delayed traps, any instruction corresponding to the SFU or coprocessor. See “Interruptions and Exceptions” on page 6-26.
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction

Name	**Data TLB miss fault/Data page fault** (15)
Cause	The data TLB entry needed by operand access of a load, store, or semaphore instruction is absent, and if data TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table
Parameters	ISR – space identifier of data address IOR – offset of data address IIR – The instruction causing the fault
IIA Queue	Front – Address of the instruction causing the fault Back – Address of the following instruction
Notes	This interruption does not occur for absolute accesses.

Name	**Non-access instruction TLB miss fault** (16)
Cause	The instruction TLB entry needed for the target of a FLUSH INSTRUCTION CACHE instruction is absent, and if TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table
Parameters	ISR – space identifier of virtual address to be flushed IOR – offset of virtual address to be flushed IIR – The instruction causing the fault
IIA Queue	Front – Address of the instruction causing the fault Back – Address of the following instruction
Notes	This interruption source is distinguished from other TLB misses because a page fault

should not result in reading the faulting page from disk. This interruption does not occur for absolute accesses.

Name	**Non-access data TLB miss fault/Non-access data page fault** (17)
Cause	The data TLB entry needed by a LOAD PHYSICAL ADDRESS, PROBE READ ACCESS, PROBE READ ACCESS IMMEDIATE, PROBE WRITE ACCESS, PROBE WRITE ACCESS IMMEDIATE, FLUSH INSTRUCTION CACHE, PURGE DATA CACHE, or a FLUSH DATA CACHE instruction is not present, and if TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table
Parameters	ISR – space identifier of virtual address IOR – offset of virtual address IIR – The instruction causing the fault
IIA Queue	Front – Address of the instruction causing the fault Back – Address of the following instruction
Notes	These interruption sources are distinguished from other TLB misses because a page fault should not result in reading the faulting page from disk. This interruption does not occur for absolute accesses.

Name	**Data memory access rights trap** (26)
Cause	Data address translation is enabled, and an access rights check fails on an operand reference for a load, store, or semaphore instruction, or a cache purge operation
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	This interruption does not occur for absolute accesses.

Name	**Data memory protection ID trap** (27)
Cause	Data address translation is enabled, the PSW P-bit is 1, and a protection identifier check fails on an operand reference for a load, store, or semaphore instruction, or cache purge operation
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	This interruption does not occur for absolute accesses.

Name	**Unaligned data reference trap** (28)
Cause	Data address translation is enabled, and a load or store instruction is attempted to an unaligned address
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	Unaligned data reference traps are not detected for absolute accesses or semaphore instructions – they are undefined operations. Only unaligned virtual memory loads and stores (including coprocessor loads and stores) are defined to terminate with the unaligned data reference trap.
Name	**Data memory protection trap/Unaligned data reference trap** (18)
Cause	Data address translation is enabled, and an access rights check or a protection identifier check fails on an operand reference for a load, store, or semaphore instruction, or a cache purge operation; a load or store instruction is attempted to an unaligned address with virtual address translation enabled (unaligned absolute references and semaphore instructions are undefined operations)
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	This interruption does not occur for absolute accesses. Only unaligned virtual memory loads and stores (including coprocessor loads and stores) are defined to terminate with the data memory protection trap. Execution of a semaphore instruction with unaligned (16 byte boundaries) addresses is an undefined operation. This trap is retained for compatibility with the earlier revisions of the architecture. In PA-RISC 1.1 (Second Edition) and later revisions, processors must use traps 26, 27, and 28 which provide equivalent functionality.
Name	**Data memory break trap** (19)
Cause	Store and semaphore instructions or cache purge operations to a page with the TLB B-bit 1 in the data TLB entry
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction

Notes	This trap is disabled if the PSW X-bit is 1. This interruption does not occur for absolute accesses.
Name	**TLB dirty bit trap** (20)
Cause	Store and semaphore instructions to a page with the D-bit 0 in the data TLB entry
Parameters	ISR – space identifier of the data address IOR – offset of the data address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	Software is invoked to update the dirty bit in the data TLB entry and the Page Table. This interruption does not occur for absolute accesses.
Name	**Page reference trap** (21)
Cause	Load, store, and semaphore instructions to a page with the T-bit 1 in its data TLB entry
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	This interruption does not occur for absolute accesses.
Name	**Data debug trap** (31)
Cause	The debug SFU has detected that the data address of a load, store, or semaphore instruction, or a cache purge operation matches the parameters set up in the SFU
Parameters	ISR – space identifier of the virtual address IOR – offset of the virtual address IIR – The instruction causing the trap
IIA Queue	Front – Address of the instruction causing the trap Back – Address of the following instruction
Notes	This trap is disabled if the PSW G-bit is 0 or if the PSW Y-bit is 1. It does not occur if the debug SFU is not implemented.
Name	**Assist emulation trap** (22)
Cause	An attempt is being made to execute an SFU instruction for an SFU whose corresponding bit in the SFU Configuration Register (SCR) is 0 or to execute a coprocessor instruction for a coprocessor whose corresponding bit in the Coprocessor Configuration Register (CCR) is 0

Parameters ISR – space identifier of the data address
IOR – offset of the data address
IIR – The instruction causing the trap

IIA Queue Front – Address of the instruction causing the trap
Back – Address of the following instruction

Notes ISR and IOR contain valid data only if the instruction is a coprocessor load or store.

Group 4 Interruptions

Name **Higher-privilege transfer trap** (23)

Cause An instruction is about to be executed at a higher privilege level than the instruction just completed and the PSW H-bit is 1

Parameters none

IIA Queue Front – Address of the instruction with the higher privilege level
Back – Address of the following instruction

Name **Lower-privilege transfer trap** (24)

Cause An instruction is about to be executed at a lower privilege level than the instruction just completed and the PSW L-bit is 1

Parameters none

IIA Queue Front – Address of the instruction with the lower privilege level
Back – Address of the following instruction

Name **Taken branch trap** (25)

Cause A taken branch was executed, and the PSW T-bit is 1

Parameters none

IIA Queue Front – Address of the instruction to be executed after the branch
Back – Address of the branch target

Notes This interruption occurs after the execution of the branch instruction, and the address of the branch instruction itself is not available. The address at the front of the IIA queue is the address of the instruction to be executed next. If the branch has nullification specified, this is the address of the nullified instruction (the PSW N-bit is 1 in this case).

5 Instruction Set

Introduction

The PA-RISC instruction set consists of defined, undefined, illegal, and null instructions. This chapter discusses the concepts of undefined and null instructions and contains a detailed description of each defined instruction. Also included are descriptions of the conditions, their completers, and the notation used in the instruction descriptions.

The instruction descriptions are divided into the following functional groups:

1. Memory Reference instructions.
2. Branch instructions.
3. Long Immediate instructions.
4. Computation instructions.
5. System Control instructions.
6. Assist instructions.

Instructions are always 32 bits in width. A 6-bit major opcode is always the first field. Source registers, if specified, are often the next two 5-bit fields. Target registers, if specified, are not fixed in any particular 5-bit field. Depending on the major opcode, the remainder of the instruction word is divided into fields that specify immediate values, space registers, additional opcode extensions, conditions, and nullification.

Undefined and Illegal Instructions

Not all of the 64 possible major opcodes of the instruction set are defined as valid instructions. (See Appendix D, “Operation Codes”, for a list of the valid instruction opcodes.) An undefined major opcode is considered an **illegal instruction**. Execution of an illegal instruction causes an illegal instruction trap.

Within each major opcode, there may be undefined opcode extensions and modifiers (these are **undefined instructions**). Interpretation of these opcodes is left to the implementor, but system integrity is not compromised. An undefined instruction, or sequence of undefined instructions, executed at a given privilege level has no effect on system state other than what would have been produced by a sequence of defined instructions running at the same privilege level. This limits the possible side-effects that could result from undefined instructions.

Undefined operations are equivalently specified. These result from normally defined instructions but with operands or specifiers that are explicitly disallowed.

Executing an optional special operation or coprocessor instruction may cause an assist exception trap or an action that depends on the definition of the specific special function unit or coprocessor.

Reserved Instruction Fields

In the Format section of the instruction pages, instruction fields marked *rv* are Reserved instruction fields. These fields are reserved for future architectural definition. To avoid incompatibility with future revisions of the architecture, software must provide zeros in all Reserved fields. When decoding instructions, processors must ignore Reserved instruction fields.

Reserved Values of an Instruction Field

Certain values of some instruction fields are Reserved. These values are reserved for future architectural definition. To avoid incompatibility with future revisions of the architecture, software must not use the Reserved values. When decoding instructions, processors must treat the Reserved values as described for the specific field.

Null Instructions

Null instructions occur when unimplemented features of the architecture are accessed. The effect of a null instruction is identical to a nullified instruction except that the Recovery Counter is decremented. There is no effect on the machine state except that the IA queues are advanced and the PSW B-bit, N-bit, X-bit, Y-bit, and Z-bit are set to 0. Null instructions most commonly occur in Level 0 systems. For example, in a Level 0 system, the instruction that writes values into space registers is a null instruction.

Conditions and Control Flow

Many instructions utilize conditions derived from the values of the operators and the operation performed. The architecture defines three distinct sets of conditions that affect control flow:

1. Arithmetic/Logical Conditions.
2. Unit Conditions.
3. Shift/Extract/Deposit Conditions.

Every instruction that tests conditions uses one of these sets. Each set contains a maximum of eight separate conditions and their negations. Most instructions that use conditions may also select the negation of a condition. Exceptions are the instructions that use the extract/deposit conditions (shift, extract, deposit, and branch on bit instructions). The location of the bit specifying negation depends on the instruction format. The conditional local branch instructions have the negation controlled by the opcode.

The condition completer field, *cond*, in the assembly language form of the instructions specifies a condition or the negation of a condition. This field expands in the machine language form to fill both the 3-bit condition field, *c*, and the 1-bit negation field, *f*, as required.

Control flow is affected by checking the results of the operation performed by the current instruction. The ways that control flow can be affected are:

1. Branching – the result determines whether or not the branch is taken.

2. Nullifying – the result determines whether or not the next instruction is nullified.
3. Trapping – the result determines whether a conditional trap is taken or execution proceeds normally.

Arithmetic/Logical Conditions

The 32-bit arithmetic/logical operations generate the set of conditions as shown in Table 5-1. No overflow conditions result from logical operations. In the table, *c* is the machine language encoding indicating the condition. The conditions are computed based on the 32-bit result of the arithmetic operation, the (leftmost) carry bit of the result, and the overflow indication. The terms **signed overflow** and **unsigned overflow** are defined for the arithmetic instructions in Table 5-2.

Table 5-1. Arithmetic/Logical Operation Conditions

c	Description	
0	never; nothing	
1	all bits are 0	
2	(leftmost bit is 1) xor signed overflow	
3	all bits are 0 or (leftmost bit is 1 xor signed overflow)	
	adds	subtracts/compares
4	no unsigned overflow	unsigned overflow
5	all bits are 0 or no unsigned overflow	all bits are 0 or unsigned overflow
6	signed overflow	
7	rightmost bit is 1	

Table 5-2. Overflow Results

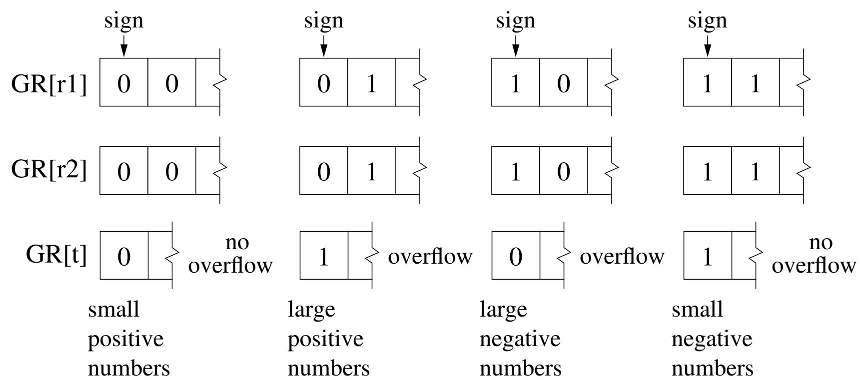
Instructions	Unsigned Overflow	Signed Overflow
Adds	The result of an unsigned addition is greater than $2^{32} - 1$ (carry == 1).	The result of signed addition is not representable in 32-bit two’s complement notation (both source operands have the same sign and the sign of the 32-bit result is different)
Subtracts and Compares	The result of an unsigned subtraction is less than 0 (i.e., *b* is greater than *a* in the operation *c* = *a* - *b*; borrow == 0).	The result of signed subtraction is not representable in 32-bit two’s complement notation (both the source operands have different signs and the sign of the 32-bit result differs from the sign of the first operand; i.e., *a* has a different sign than *b* and *c* in the operation *c* = *a* - *b*).
Divide Step and Shift and Adds	One or more of the bits shifted out is 1, or the result of the operation is not in the range 0 through $2^{32} - 1$.	One or more of the bits shifted out differs from the leftmost bit following the shift, or the result of the operation is not representable in 32-bit two’s complement notation.

When implementing the DIVIDE STEP and SHIFT AND ADD instructions, the overflow condition XORed into conditions 2 and 3 may optionally include the overflow that is generated during the pre-shift operation. The only overflow that must be included is the one actually generated by the arithmetic operation.

If a signed overflow occurs during the shift operation of a DIVIDE STEP or SHIFT AND ADD instruction, conditions 2 and 3 are not meaningful; therefore, the result of a condition 2 or condition 3 test is not predictable.

PROGRAMMING NOTE

The figure below shows signed number addition and indicates the signed overflow condition when both operands are small positive numbers, large positive numbers, large negative numbers, or small negative numbers.



Signed overflow can occur only when adding numbers with the same sign. Addition of numbers with unlike signs will always result with a "no overflow" condition.

The interpretation of the arithmetic/logical conditions varies according to the operation performed. The interpretation for comparisons and subtracts is shown in Table 5-3. In this table, *cond* is in assembly language format and *c* and *f* are in machine language format. *opd1* denotes operand 1 (an immediate value or a register's contents) in the assembly language instruction format and *opd2* denotes operand 2 (a register's contents). The condition, <<, "opd1 is less than opd2 (unsigned)" is equivalent to unsigned overflow in Table 5-1.

Table 5-3. Compare/Subtract Instruction Conditions

cond	Description	c	f
	never	0	0
=	opd1 is equal to opd2	1	0
<	opd1 is less than opd2 (signed)	2	0
<=	opd1 is less than or equal to opd2 (signed)	3	0
<<	opd1 is less than opd2 (unsigned)	4	0
<<=	opd1 is less than or equal to opd2 (unsigned)	5	0
SV	opd1 minus opd2 overflows (signed)	6	0
OD	opd1 minus opd2 is odd	7	0
TR	always	0	1
<>	opd1 is not equal to opd2	1	1
>=	opd1 is greater than or equal to opd2 (signed)	2	1
>	opd1 is greater than opd2 (signed)	3	1
>>=	opd1 is greater than or equal to opd2 (unsigned)	4	1
>>	opd1 is greater than opd2 (unsigned)	5	1
NSV	opd1 minus opd2 does not overflow (signed)	6	1
EV	opd1 minus opd2 is even	7	1

The interpretation for adds is shown in Table 5-4. *Cond* is in assembly language format and *c* and *f* are in machine language format.

Table 5-4. Add Instruction Conditions

cond	Description	c	f
	never	0	0
=	opd1 is equal to negative of opd2	1	0
<	opd1 is less than negative of opd2 (signed)	2	0
<=	opd1 is less than or equal to negative of opd2 (signed)	3	0
NUV	opd1 plus opd2 does not overflow (unsigned)	4	0
ZNV	opd1 plus opd2 is zero or no overflow (unsigned)	5	0
SV	opd1 plus opd2 overflows (signed)	6	0
OD	opd1 plus opd2 is odd	7	0
TR	always	0	1
<>	opd1 is not equal to negative of opd2	1	1
>=	opd1 is greater than or equal to negative of opd2 (signed)	2	1
>	opd1 is greater than negative of opd2 (signed)	3	1
UV	opd1 plus opd2 overflows (unsigned)	4	1
VNZ	opd1 plus opd2 is nonzero and overflows (unsigned)	5	1
NSV	opd1 plus opd2 does not overflow (signed)	6	1
EV	opd1 plus opd2 is even	7	1

The interpretation of the condition completers for the SHIFT AND ADD instructions is similar to the ADD instructions (Table 5-4). If no overflow occurs, *opd1* is the shifted value. For example, the completer "=" implies that the shifted *opd1* equals the negative of *opd2*. If overflow occurs, the interpretations in Table 5-4 do not apply. Table 5-1 and the definition of overflow in Table 5-2 can be used to determine if the condition is satisfied.

The interpretation of the condition completers for the DIVIDE STEP instruction are similar to the subtract or add conditions, depending on the state of the PSW V-bit. If no overflow occurs, then *opd1* is the shifted value. If overflow occurs, the interpretations in Tables 5-3 and 5-4 do not apply. Again, Tables 5-1 and 5-2 can be used to determine if the condition is satisfied.

For logical operations, the conditions are computed based only on the result. The interpretation of the arithmetic/logical conditions for logical instructions is shown in Table 5-5. In this table, *cond* is in assembly language format and *c* and *f* are in machine language format. The unlisted values of the condition field are undefined for the logical operations.

Table 5-5. Logical Instruction Conditions

cond	Description	c	f
	never	0	0
=	all bits are 0	1	0
<	leftmost bit is 1	2	0
<=	leftmost bit is 1 or all bits are 0	3	0
OD	rightmost bit is 1	7	0
TR	always	0	1
<>	some bits are 1	1	1
>=	leftmost bit is 0	2	1
>	leftmost bit is 0, some bits are 1	3	1
EV	rightmost bit is 0	7	1

Unit Conditions

The operations concerned with sub-units of a word generate the conditions shown in Table 5-6. The conditions are computed based on the 32-bit result of the unit operation and the eight 4-bit carries. In this table, *cond* is in assembly language format and *c* and *f* are in machine language format. The unlisted values of the condition field are undefined for the unit operations.

Table 5-6. Unit Instruction Conditions

cond	Description	c	f
	never	0	0
SBZ	Some Byte Zero	2	0
SHZ	Some Halfword Zero	3	0
SDC	Some Digit Carry	4	0
SBC	Some Byte Carry	6	0
SHC	Some Halfword Carry	7	0
TR	always	0	1
NBZ	No Bytes Zero	2	1
NHZ	No Halfwords Zero	3	1
NDC	No Digit Carries	4	1
NBC	No Byte Carries	6	1
NHC	No Halfword Carries	7	1

Shift/Extract/Deposit Conditions

The shift, extract, and deposit operations generate the conditions shown in Table 5-7. The conditions are computed based on the 32-bit result of the operation. In this table, *cond* is in assembly language format and *c* is in machine language format. The BRANCH ON BIT instructions use only the "<" (leftmost bit is 1) and ">=" (leftmost bit is 0) conditions. The MOVE AND BRANCH instructions also use the extract/ deposit conditions.

Table 5-7. Shift/Extract/Deposit Instruction Conditions

cond	Description	c
	never	0
=	all bits are 0	1
<	leftmost bit is 1	2
OD	rightmost bit is 1	3
TR	always	4
<>	some bits are 1	5
>=	leftmost bit is 0	6
EV	rightmost bit is 0	7

Instruction Notations

Each instruction is described in detail in the following pages. Each description includes the full name of the instruction, the assembly language mnemonic and syntax format, machine instruction format, purpose, a narrative description, an operational description, exceptions, and notes concerning usage. In some cases, programming notes are included for additional guidance to programmers. The instruction's operation is described in a C-like algorithmic language. This language is the same as the C programming language with a few exceptions. These are:

1. The characters "{ }" are used to denote bit fields.
2. The assignment operator used is "←" instead of "=".
3. The functions "cat" (concatenation), and "xor" (logical exclusive OR) take a variable number of arguments, for which there is no provision in C.
4. The switch statement usage is improper because we do not use constant expressions for all the cases.

For the complete syntax and other considerations used in writing assembly language programs, please refer to the *Assembly Language Reference Manual*.

Bit Ranges

A range of bits within a larger unit, is denoted by "unit{range}", where unit is the notation for memory, a register, a temporary, or a constant; range is a single integer to denote one bit, or two integers separated by ".." to denote a range of bits.

For example, "GR[1]{0}" denotes the leftmost bit of general register 1, "CR[24]{27..31}" denotes the rightmost five bits of control register 24, and "5{0..6}" denotes a 7-bit field containing the number 5. If m > n, then {m..n} denotes the null range.

Registers

In general, a register name consists of two or three uppercase letters. The name of a member of a register array consists of a register name followed by an index in square brackets. For example, "GR[1]" denotes general register 1.

The named registers and register arrays used in the operational descriptions are:

Register	Range	Description
GR[t]	t = 0..31	General registers
SHR[t]	t = 0..6	Shadow registers
SR[t]	t = 0..7	Space registers
CR[t]	t = 0, 8..31	Control registers
CPR[uid][t]	t = 0..31	Coprocessor "uid" registers
FPR[t]	t = 0..31	Floating-point coprocessor registers

The Processor Status Word and the Interruption Processor Status Word, denoted by "PSW" and "IPSW", are treated as a series of 1-bit and multiple-bit fields. A field of either is denoted by the register name followed by a field name in square brackets, and bit ranges within such fields are denoted by the usual notation. For example, PSW[C/B] denotes the 8-bit carry/borrow field of the PSW and PSW[C/B]{0} denotes bit 0 of that field.

Temporaries

A temporary name comprises three or more lowercase letters and denotes a quantity which requires naming, either for clarity, or because of limitations imposed by the sequentiality of the operational notation. It may or may not represent an actual processing resource in the hardware. The length of the quantity denoted by a temporary is implicitly determined and is equal to that of the quantity first assigned to it in an operational description.

Operators

The operators used and their meanings are as follows:

<table>
<tr><td>←</td><td>assignment</td><td>|</td><td>bitwise or</td></tr>
<tr><td>+</td><td>addition</td><td>==</td><td>equal to</td></tr>
<tr><td>–</td><td>subtraction</td><td><</td><td>less than</td></tr>
<tr><td>*</td><td>multiplication</td><td>></td><td>greater than</td></tr>
<tr><td>~</td><td>bitwise complement</td><td>!=</td><td>not equal to</td></tr>
<tr><td>&&</td><td>logical and</td><td><=</td><td>less than or equal to</td></tr>
<tr><td>&</td><td>bitwise and</td><td>>=</td><td>greater than or equal to</td></tr>
<tr><td>||</td><td>logical or</td><td></td><td></td></tr>
</table>

All operators are binary, except that “~” is unary and “–” is both binary and unary, depending on the context.

Control Structures and Functions

The control structures used in the notation are relatively standard. The expression statements describe a computation performed by the ALU or some other hardware for its side effects rather than the value of the computation. The functions listed below are used to localize long calculations that are used in several places. Semicolons separate the statements.

Function	Description
assemble_3(x)	Assembles a 3-bit space register number: return(cat(x{2},x{0..1}))
assemble_12(x,y)	Assembles a 12-bit immediate: return(cat(y,x{10},x{0..9}))
assemble_17(x,y,z)	Assembles a 17-bit immediate: return(cat(z,x,y{10},y{0..9}))
assemble_21(x)	Assembles a 21-bit immediate: return(cat(x{20},x{9..19},x{5..6},x{0..4},x{7..8}))
cat(x1, ..., xn)	Concatenates the passed arguments, *x1* through *xn*.
low_sign_ext(x,len)	Removes the rightmost bit of *x* and extends the field to the left with that bit to form a 32-bit quantity. The field is of size *len*: return(sign_ext(cat(x{len-1},x{0..len-2}),len))
lshift(arg1,arg2)	*arg1* is logically shifted left by the number of bits specified in *arg2*.
mem_load	See "Memory Reference Instructions" on page 5-15.
mem_store	See "Memory Reference Instructions" on page 5-15.
rshift(arg1,arg2)	*arg1* is logically shifted right by *arg2* bits.
send_to_copr(u,t)	Sends the 5-bit value *t* to coprocessor unit *u*.
sign_ext(x,len)	Extends *x* on the left with sign bits to form a 32-bit quantity, taking the leftmost bit for the field of size *len* as the sign bit.

sign_ext_64(x,len)	Identical to sign_ext(x,len) except that it extends the value to 64 bits:
	sign_ext_64(x,len)
	{
	if (x{0} == 1)
	return(cat(-1{0..31},sign_ext(x,len));
	else
	return(cat(0{0..31},sign_ext(x,len));
	}

store_in_memory(space,offset,low,high,hint,data)	The function store_in_memory is identical to mem_store except that it forces the data to be stored into main memory. The data may optionally remain in the cache.

Function	Description
xor(x1, ..., xn)	Produces the bitwise exclusive or of the passed arguments.
zero_ext(x,len)	Extends *x* on the left, for the field of size *len*, with zeros to form a 32-bit quantity.
zero_ext_64(x,len)	Identical to zero_ext(x,len) except that it extends the value to 64 bits: return(cat(0{0..31},zero_ext(x,len)))

Miscellaneous Constructs

Numerous mnemonic constructs are used to represent things that do not fit easily into the rest of the notation described above or whose details are more implementation-dependent than defined.

Function	Description
absolute_address(space,offset)	Returns the absolute address corresponding to the passed virtual address.
alloc_DTLB(space,offset,entry)	Allocates a slot in the data TLB based on the space and offset arguments. The position of the slot is returned through the pointer called entry.
alloc_ITLB(space,offset,entry)	Same as alloc_DTLB, except that a new slot is allocated in the instruction TLB.
broadcast_purge_DTLB(space,offset)	In a multiprocessor system, the other processors are made to search their data TLBs for a translation matching the specified virtual address. If found, they invalidate, remove, or alter the matching entries.
broadcast_purge_ITLB(space,offset)	In a multiprocessor system, the other processors are made to search their instruction TLBs for a translation matching the specified virtual address. If found, they invalidate, remove, or alter the matching entries.
coherence_index(space,offset)	Returns the coherence index corresponding to the passed effective address. See “Cache Coherence with I/O” on page 3-17.
coherent_system	Boolean; the value is 1 if the system is fully coherent; the value is 0 if the system is partially or completely non-coherent.
coprocessor_condition(id,opcode,n)	A coprocessor specific condition is returned based on the arguments and the current state of the coprocessor.
coprocessor_op(id,opcode,n,priv)	A coprocessor specific operation is performed based on the arguments and the current state of the coprocessor.

Function	Description
flush_data_cache(space,offset)	If the cache line containing the effective address is present, it is invalidated. If the line is dirty, it is written back to main memory.
flush_data_cache_entry(space,offset)	Zero or more cache lines specified by an implementation-dependent function of the address are invalidated. If any of these lines are dirty, they are written back to main memory.
flush_instruction_cache(space,offset)	If the cache line containing the effective address is present, it is invalidated. If the line is dirty it is written back to main memory.
flush_instruction_cache_entry(space,offset)	Zero or more cache lines specified by an implementation-dependent function of the address are invalidated. If any of these lines are dirty, they are written back to main memory.
level_0	Boolean; the value is 1 if the processor architecture is Level 0.
measurement_enabled	Boolean; when the value is 1, the performance monitor coprocessor is enabled to make measurements; when the value is 0, the measurements are disabled. This condition is independent of the state of CCR bit 2.
phys_mem_load(addr,low,high,hint)	Returns the data in physical memory (consisting of memory and the cache) starting at the low'th bit beyond the beginning of the byte at address, addr, and ending at the high'th bit beyond the beginning of the byte at address, addr. If the PSW E-bit is 1, the data bytes are swapped before they are returned. The cache control hint, hint, is a recommendation to the processor on how to resolve cache coherence. See "Cache Control" on page 5-17. This function is used for absolute accesses to memory.
phys_mem_store(addr,low,high,hint,data)	Stores the data in physical memory (consisting of memory and the cache) starting at the low'th bit beyond the beginning of the byte at address, addr, and ending at the high'th bit beyond the beginning of the byte at address, addr. If the PSW E-bit is 1, the data bytes are swapped before they are stored. The cache control hint, hint, is a recommendation to the processor on how to resolve cache coherence. See "Cache Control" on page 5-17. This function is used for absolute accesses to memory.

Function	Description
purge_DTLB(entry)	The specified TLB entry is invalidated, removed, or altered.
purge_DTLB_entry(entries)	The E-bit(s) of the zero or more data TLB entries specified are set to 0. All other fields of these TLB entries are undefined.
purge_ITLB(entry)	The specified TLB entry is invalidated, removed, or altered.
purge_ITLB_entry(entries)	The E-bit(s) of the zero or more instruction TLB entries specified are set to 0. All other fields of these TLB entries are undefined.
purge_or_flush_data_cache(space,offset)	If the cache line specified by the effective address is present, it is invalidated. If the line is dirty, it may optionally be written back to memory.
read_access_allowed(space,offset,x)	In non-Level 0 systems, returns 1 if read access is allowed to the effective address at the privilege level given by the two rightmost bits of *x*. Returns 0 otherwise. Always returns 1 in Level 0 systems.
search_DTLB(space,offset,entry)	Searches the data TLB for an entry (valid or invalid) whose virtual address matches the virtual address passed to it, and returns true if it finds one. As a side effect, if such a translation is found, the variable *entry* is set to point to the TLB slot containing this translation.
search_ITLB(space,offset,entry)	Same as search_DTLB, except that the instruction TLB is searched.
select_data_cache_entries(space,offset)	An implementation-dependent function which returns a list of zero or more entries.
select_instruction_cache_entries(space,offset)	An implementation-dependent function which returns a list of zero or more entries.
select_DTLB_entries(space, offset)	An implementation-dependent function which returns a list of zero or more entries.
select_ITLB_entries(space, offset)	An implementation-dependent function which returns a list of zero or more entries.
sfu_condition0(opcode,priv) sfu_condition1(opcode,priv) sfu_condition2(opcode,priv,r) sfu_condition3(opcode,priv,r1,r2)	An SFU specific condition is returned based on the SFU instruction format, the arguments, and the current state of the special function unit.
sfu_operation0(opcode,priv) sfu_operation1(opcode,priv) sfu_operation2(opcode,priv,r) sfu_operation3(opcode,priv,r1,r2)	An SFU specific operation is performed based on SFU instruction format, the arguments, and the current state of the special function unit.

Function	Description

space_select(s_field,base)	Returns the space ID selected by the s-field of the instruction and the base register value as follows:
	space_select(s_field,base)
	{
	    if (level_0)
	        return(0);
	    if (s_field == 0)
	        return(SR[base{0..1} + 4]);
	    else
	        return(SR[s_field]);
	}

virt_mem_load(addr,low,high,hint)	Returns the data in virtual memory (consisting of memory and the cache) starting at the low'th bit beyond the beginning of the byte at address, addr, and ending at the high'th bit beyond the beginning of the byte at address, addr. If the PSW E-bit is 1, the data bytes are swapped before they are returned. The cache control hint, hint, is a recommendation to the processor on how to resolve cache coherence. See "Cache Control" on page 5-17. This function is used for virtual accesses to memory.
virt_mem_store(addr,low,high,hint,data)	Stores the data in virtual memory (consisting of memory and the cache) starting at the low'th bit beyond the beginning of the byte at address, addr, and ending at the high'th bit beyond the beginning of the byte at address, addr. If the PSW E-bit is 1, the data bytes are swapped before they are stored. The cache control hint, hint, is a recommendation to the processor on how to resolve cache coherence. See "Cache Control" on page 5-17. This function is used for virtual accesses to memory.
write_access_allowed(space,offset,x)	In non-Level 0 systems, returns 1 if write access is allowed to the effective address at the privilege level given by the two rightmost bits of *x*. Returns 0 otherwise. Always returns 1 in Level 0 systems.

Instruction Descriptions

Figure 5-1 illustrates the information presented in each of the instruction descriptions. The information presented in this figure is for illustrative purposes only and does not represent a valid instruction.

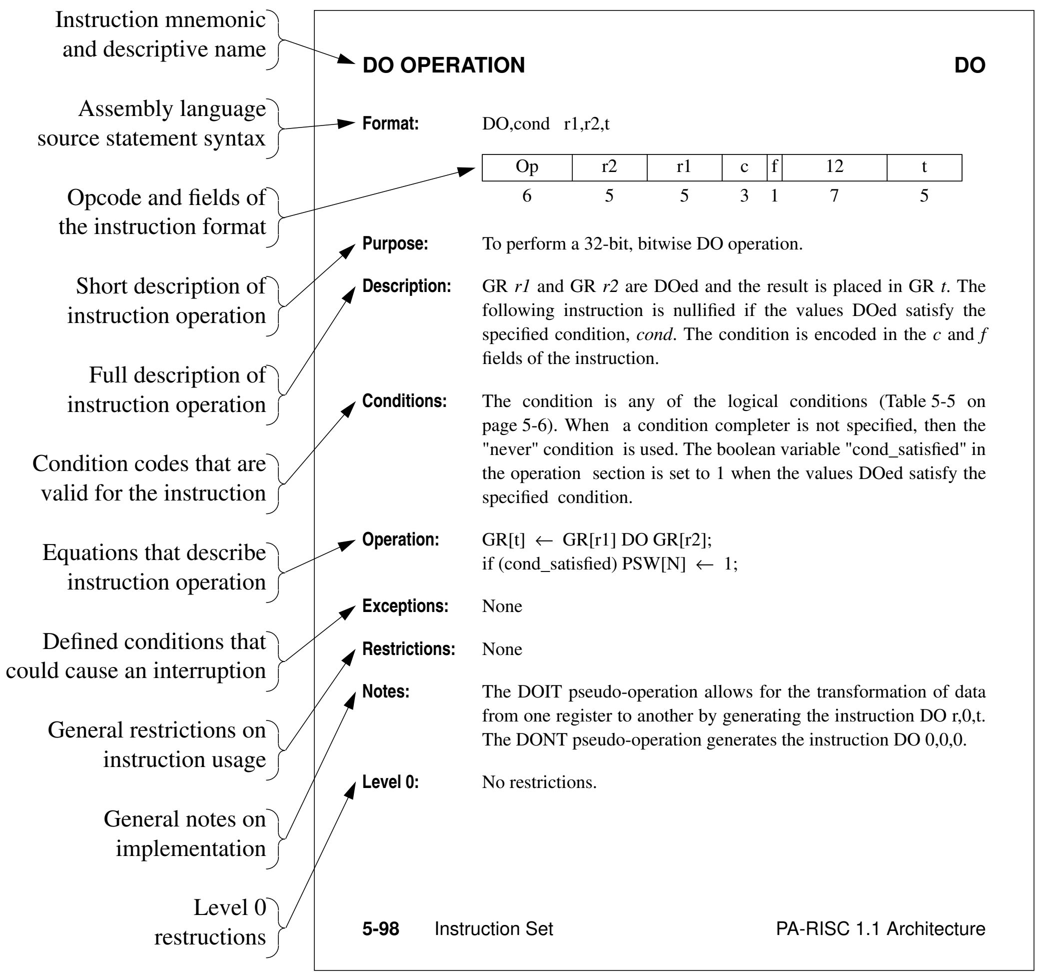


Figure 5-1. Instruction Description Example

Memory Reference Instructions

Memory reference instructions load values into and store values from the general registers. The types included are: short displacement, long displacement, and indexed. It is possible to modify the base value in a general register by the displacement or index.

The rightmost bits of computed addresses are not ignored. Unaligned load and store instructions with data address translation enabled to halfwords, words, or doublewords cause an unaligned data reference trap. Semaphore operations and absolute accesses to unaligned data are undefined operations.

Memory reference instructions work directly between the registers and main memory. They also can operate between the registers and the data cache on implementations so equipped. A load instruction loads a general register with data from the data cache. A store instruction stores a data value from a general register into the data cache. Normally this distinction is transparent to the programmer, but provisions are made for cache and TLB operations requiring cognizance of the data cache (see "System Control Instructions" on page 5-136). A cache line can be 16, 32, or 64 bytes in length.

Depending on the state of the PSW D-bit (data address translation bit), all load and store instructions perform virtual accesses (when the PSW D-bit is 1) or physical accesses (when the PSW D-bit is 0). In Level 0 systems or when data translation is disabled (PSW D-bit is 0), the s-field of the instruction is ignored and the 32-bit offset is directly used as the absolute address. LOAD WORD ABSOLUTE INDEXED, LOAD WORD ABSOLUTE SHORT, STORE WORD ABSOLUTE SHORT, and all loads and stores in Level 0 systems always perform absolute accesses.

The state of the PSW E-bit determines whether the data which is loaded or stored is big endian (when the PSW E-bit is 0) or little endian (when the PSW E-bit is 1).

Memory is accessed using the following procedures:

```
mem_load(space,offset,low,high,hint)
{
    addr ← cat(space, offset);
    if (PSW[D] == 0 || level_0)
        return(phys_mem_load(addr,low,high,hint));
    else
        return(virt_mem_load(addr,low,high,hint));
}

mem_store(space,offset,low,high,hint,data)
{
    addr ← cat(space, offset);
    if (PSW[D] == 0 || level_0)
        phys_mem_store(addr,low,high,hint,data);
    else
        virt_mem_store(addr,low,high,hint,data);
}
```

In order to determine if a memory reference instruction is defined or undefined and whether it is executed in an atomic or non-atomic fashion, the following must be considered:

1. Whether the memory reference is virtual or absolute.
2. Whether or not the optional U (Uncacheable) bit is implemented.
3. The state of the U-bit, if implemented.
4. Whether the reference is made to a page in the memory or the I/O address space.

The following paragraphs specify the rules under which the above considerations determine if an instruction is defined or undefined and if it is executed in an atomic or non-atomic fashion.

If the absolute or virtual reference is made to a page in the memory address space then:

- the load and store instructions operating on bytes, halfwords, and words are defined and atomic.
- the coprocessor load and store instructions operating on words and doublewords are defined and atomic.
- the STBYS instruction is defined and atomic.
- the flush and purge instructions are also defined and atomic.
- the LDCW instruction is defined, except if there is a virtual access to a page with its U-bit 1, in which case it is undefined.

If the reference is made to a page in the I/O address space and:

- the access is absolute, or
- the access is virtual and the U-bit is not implemented, or
- the access is virtual and the U-bit is implemented and is 1, then:
  - the load and store instructions operating on bytes, halfwords, and words are defined and atomic.
  - the STBYS instruction is defined but performed in a non-atomic fashion.
  - the coprocessor load and store instructions operating on words and doublewords are undefined.
  - the LDCW instruction is undefined.
  - the flush and purge instructions are undefined.

All virtual accesses to a page in the I/O address space with its U-bit 0 are undefined.

LOAD OFFSET, LOAD IMMEDIATE LEFT, LOAD PHYSICAL ADDRESS, LOAD COHERENCE INDEX, and LOAD SPACE IDENTIFIER are not memory reference instructions.

PROGRAMMING NOTE

Execution may be faster if software avoids dependence on register interlocks. Instruction scheduling to avoid the need for interlocking is recommended. A register interlock will occur if an instruction attempts to use a register which is the target of a previous load instruction that has not yet completed. This does not restrict the length of the delay a load instruction may incur in a particular system to a single execution cycle; in fact, the delay may be much longer for a cache miss, a TLB miss, or a page fault.

Debugging is facilitated by the data memory break trap. This trap occurs whenever a store (other than STORE WORD ABSOLUTE SHORT), a LOAD AND CLEAR WORD INDEXED, a LOAD AND CLEAR WORD SHORT, or a purge data cache operation is performed to a page with the B-bit 1 in its TLB entry and the PSW X-bit is 0.

Cache Control

Some memory reference instructions contain a 2-bit cache control field, *cc*, which provides a hint to the processor on how to resolve cache coherence. The processor may disregard the hint without compromising system integrity, but performance may be enhanced by following the hint.

There are three different categories of cache control hints: load instruction cache control hints, store instruction cache control hints, and load and clear word instruction cache control hints. The cache control hints are specified by the *cc* completer to the instruction and encoded in the *cc* field of the instruction.

The cache control hints for indexed load, coprocessor indexed load, and short displacement load instructions are shown in Table 5-8. Implementation of the hints by a processor is optional, but the processor must treat unimplemented and Reserved hints as if no hint had been specified.

The Spatial Locality cache control hint is a recommendation to the processor to fetch the addressed cache line from memory but to not displace any existing cache data because there is good spatial locality but poor temporal locality.

Table 5-8. Load Instruction Cache Control Hints

Completer	Description	cc
<none>	No hint	00
	Reserved	01
SL	Spatial Locality	10
	Reserved	11

The cache control hints for short displacement store, STORE BYTES SHORT, and coprocessor indexed store instructions are shown in Table 5-9. Implementation of the hints by a processor is optional, but the processor must treat unimplemented and Reserved hints as if no hint had been specified.

The Block Copy cache control hint is a recommendation to the processor not to fetch the addressed cache line if it is not found in the cache. Instead, the processor may create a cache line for the specified address and perform the store instruction on the created line. If the cache line is not fetched then the processor must zero the rest of the created cache line if the privilege level is 1, 2, or 3. The processor may optionally zero the rest of created the cache line if the privilege level is 0. If the store instruction with the Block Copy hint does not store into at least the first byte of the cache line, the processor must perform the store as if the cache control hint had not been specified.

The Spatial Locality cache control hint is a recommendation to the processor to fetch the addressed cache line from memory but to not displace any existing cache data because there is good spatial locality but poor temporal locality.

Table 5-9. Store Instruction Cache Control Hints

Completer	Description	cc
<none>	No hint	00
BC	Block Copy	01
SL	Spatial Locality	10
	Reserved	11

The cache control hints for the LOAD AND CLEAR WORD INDEXED and LOAD AND CLEAR WORD SHORT semaphore instructions are shown in Table 5-10. The implementation of the hints by the processor is optional. If no hints are implemented, the processor must treat all hints as if no hint had been specified. If the Coherent Operation hint is implemented, the processor must treat Reserved hints as if the Coherent Operation hint had been specified.

The Coherent Operation cache control hint is a recommendation to the processor, that if the addressed data is already in the cache, it can operate on the addressed data in the cache rather than having to update memory.

All software users of a semaphore must access the semaphore using the same cache control hint. Sharing a semaphore using different cache control hints is undefined.

Table 5-10. Load And Clear Word Instruction Cache Control Hints

Completer	Description	cc
<none>	No hint	00
CO	Coherent Operation	01
	Reserved	10
	Reserved	11

Loads and Stores

This section describes memory load and store instructions that have long displacements but which do not modify the base register. This class of instructions loads data from memory into the general register denoted by the *t* field and stores data from the general register denoted by the *r* field to memory. The effective memory reference address is formed by the addition of a displacement to a base value specified through the instruction. The entity being transferred can be a word, halfword, or a byte. The 14-bit byte displacement is in two's complement notation with the sign bit as its rightmost bit denoted by the *im14* field. The opcode specifies the particular data transfer to be performed.

The format of the load and store instructions is:

op	b	t/r	s	im14
6	5	5	2	14

When data translation is enabled, the effective space ID is the contents of the space register indicated by the s-field if the s-field is nonzero. If the s-field is 0, the effective space ID is the contents of the space register whose number is the sum of 4 and the two leftmost bits of GR *b*. In Level 0 systems or when data translation is disabled, the s-field of the instruction is ignored and the offset is directly used as the address.

The effective offset is the sum of the contents of GR *b* and the sign-extended displacement *d*.

The address calculation is shown in Figure 5-2 and Figure 5-3 in two parts: Figure 5-2 shows space identifier selection, while Figure 5-3 shows offset computation. In systems which support virtual addressing, the effective address is formed by the concatenation of the space identifier and the offset.

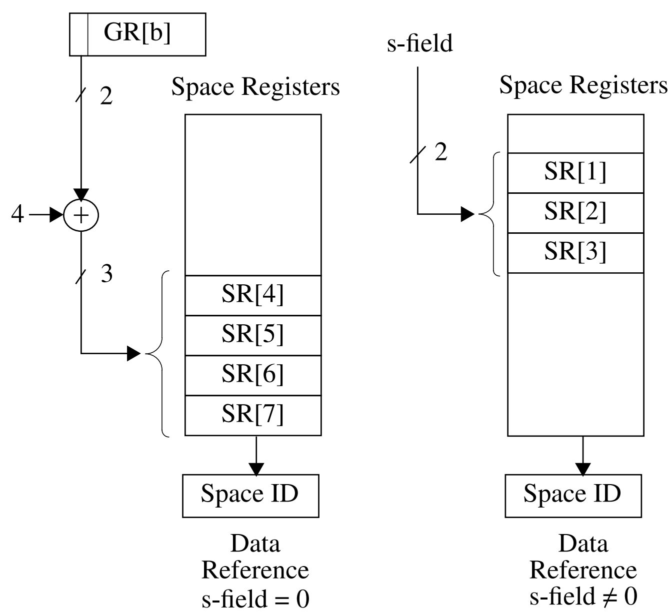


Figure 5-2. Space Identifier Selection

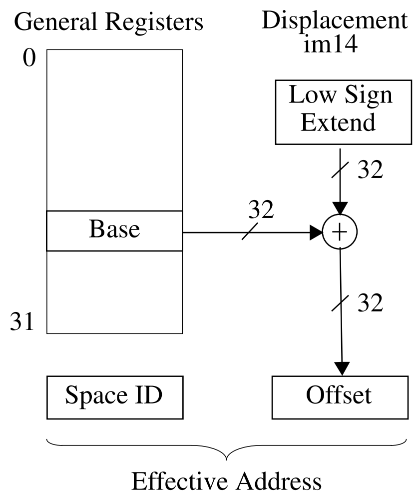


Figure 5-3. Loads and Stores

Loads and Stores with Base Register Modification

This class of instructions loads data from memory into the general register denoted by the *t* field and stores data from the general register denoted by the *r* field to memory. The effective memory reference address is formed by the addition of an immediate displacement to a base value. The entity being loaded is a word. The 14-bit byte displacement is in two's complement notation with the sign bit as its rightmost bit denoted by the *im14* field. In these instructions, base register modification always takes place.

The format of these instructions is:

op	b	t/r	s	im14
6	5	5	2	14

The calculation of the effective space ID for these instructions is the same as for the loads and stores described in the previous section. The effective offset, however, depends on the sign of the displacement *d*.

- Pre-decrement - if *d* is negative, its sign-extended value is added to GR *b* and the result is stored in GR *b*. The effective offset is the value stored in GR *b*.
- Post-increment - if *d* is positive, the effective offset is the original contents of GR *b*. The sum of the contents of GR *b* and the sign-extended value of *d* is stored in GR *b*.

The space identifier is computed like any other data memory reference (see Figure 5-2 on page 5-19). The calculation of the offset portion of the effective address for different completers is shown in Figure 5-4.

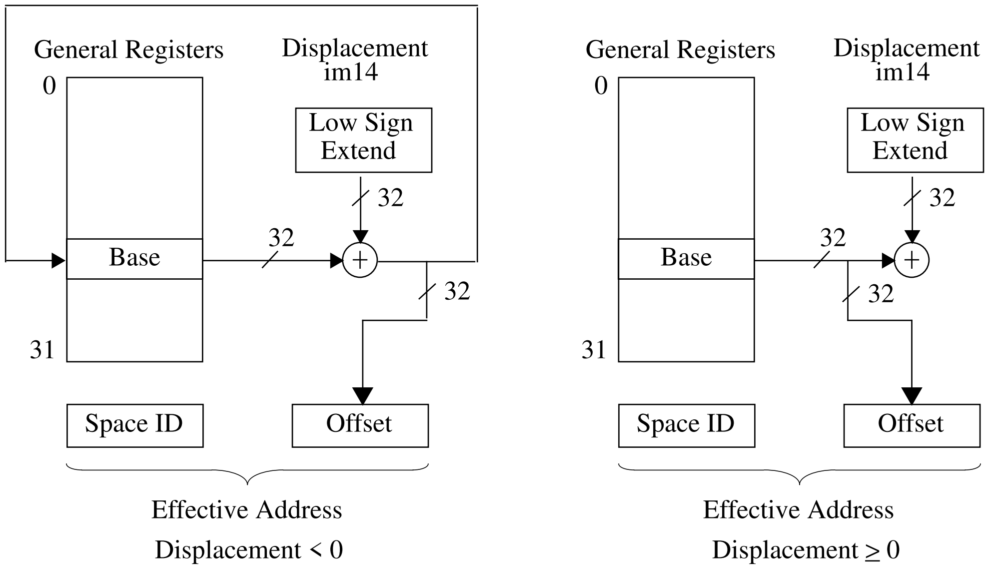


Figure 5-4. Load and Store Word Modify

Indexed Loads

This class of instructions loads data from memory into a general register, specified in the instruction, where the effective memory reference address is formed by the addition of an index value to a base value specified in the instruction. The entity being loaded can be a word, halfword, or a byte. This class also includes the LOAD AND CLEAR WORD INDEXED instruction.

The format of the indexed load instructions is:

03	b	x	s	u	0	cc	ext4	m	t
6	5	5	2	1	1	2	4	1	5

The *u* and *m* fields specify the actual function as follows:

u = 0	index register.
= 1	index register shifted by data size.
m = 0	no base register modification.
= 1	base register modification.

Index shift by data size means that the index value (contents of general register *x*) is multiplied by the size of the data item being loaded - 1 if it is a byte load, 2 for a halfword load, and 4 for a word load (these correspond to shifts by 0, 1, and 2 bits, respectively). Base register modification also results in the contents of GR *b* being replaced by the sum of the index value and the previous contents of GR *b*.

The *cc* field specifies the cache control hint (see Table 5-8 on page 5-17 and Table 5-10 on page 5-18).

In the instruction descriptions on the following pages, the term *cmplt* is used to denote the completer field which encodes the *u* and *m* fields. The list of completers and the address formation functions they specify appear in Table 5-11.

Table 5-11. Indexed Load Completers

cmplt	Description	u	m
<none>	no index shift, don't modify base register	0	0
M	no index shift, modify base register	0	1
S	Shift index by data size, don't modify base register	1	0
SM or S,M	Shift index by data size, modify base register	1	1

In the above table, *cmplt* is in assembly language format and *u* and *m* are in machine language format.

The space identifier is computed like any other data memory reference (see Figure 5-2 on page 5-19). The calculation of the offset portion of the effective address for different completers is shown in Figure 5-5.

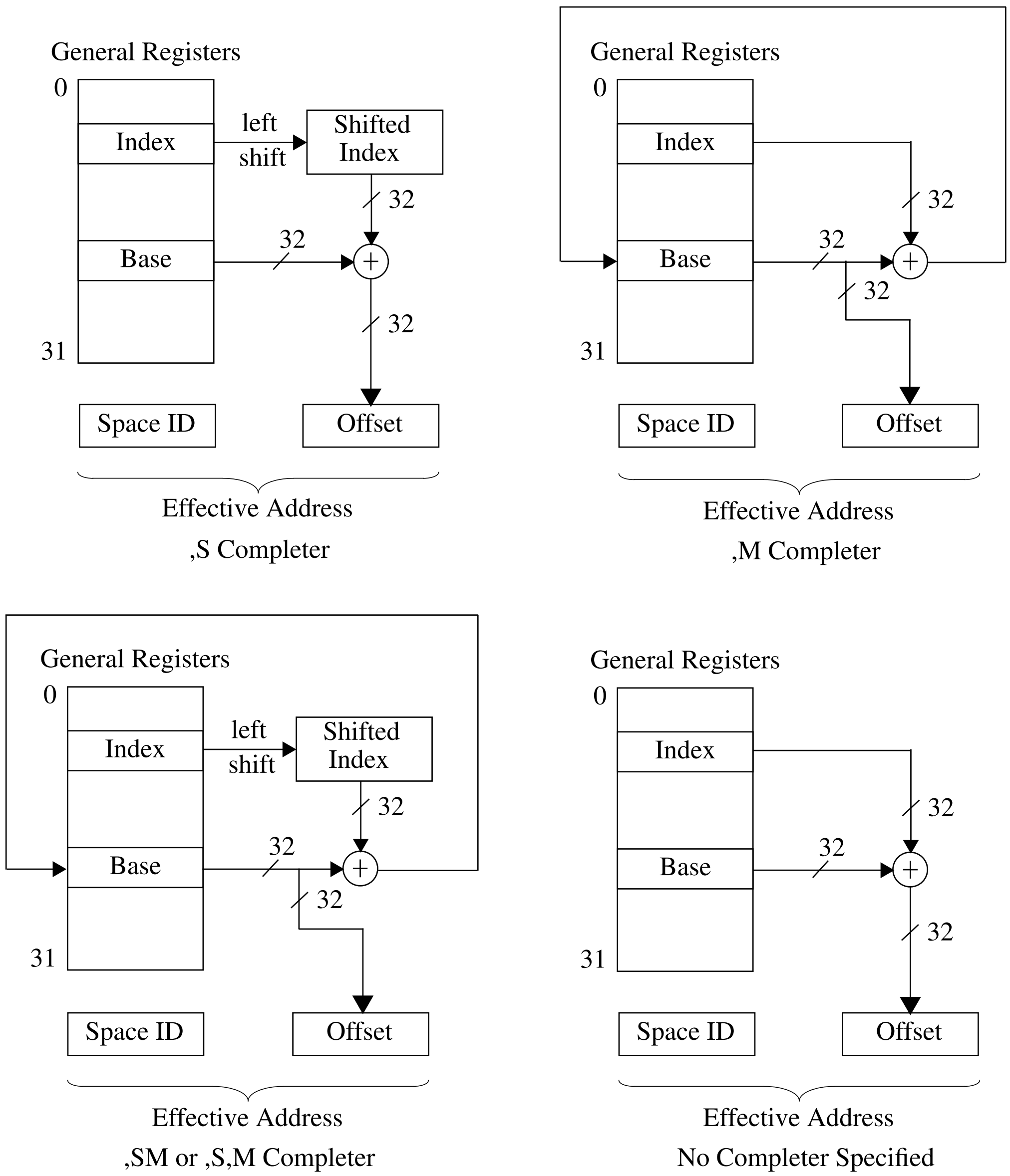


Figure 5-5. Indexed Loads

Short Displacement Loads and Stores

This set of instructions uses a short 5-bit displacement to load and store data values from and to memory. The effective address is formed by the addition of the low sign extended displacement to a base register. The sign bit of the short displacement is the rightmost bit of the 5-bit field, which is in two's complement notation. The entities being loaded or stored can be words, halfwords, or bytes. This class also includes the LOAD AND CLEAR WORD SHORT instruction.

The format of the short displacement load instructions is:

03	b	im5	s	a	1	cc	ext4	m	t
6	5	5	2	1	1	2	4	1	5

and that of the short displacement stores is:

03	b	r	s	a	1	cc	ext4	m	im5
6	5	5	2	1	1	2	4	1	5

The *ext4* field in the instruction format above specifies a load or a store and the data size. The *a* and *m* fields specify the following functions:

a = 0 modify after if m = 1.
= 1 modify before if m = 1.

m = 0 no address modification.
= 1 address modification.

The *cc* field specifies the cache control hint (see Table 5-8 on page 5-17, Table 5-9 on page 5-18, and Table 5-10 on page 5-18).

In the instruction descriptions that follow, some information is coded into the instruction names and the remainder is coded in the completer field (denoted by *cmplt* in the descriptions). Table 5-12 lists the assembly language syntax of the completer, the functions performed, and the values coded into the *a* and *m* bit fields of the instruction.

Table 5-12. Short Displacement Load and Store Completers

cmplt	Description	a	m
<none>	don't modify base register	x	0
MA	Modify base register After	0	1
MB	Modify base register Before	1	1
Notes: x indicates don't care.			

In the above table, *cmplt* is in assembly language format and *a* and *m* are in machine language format.

The space identifier is computed like any other data memory reference (see Figure 5-2 on page 5-19). The calculation of the offset portion of the address for different completers is shown in Figure 5-6.

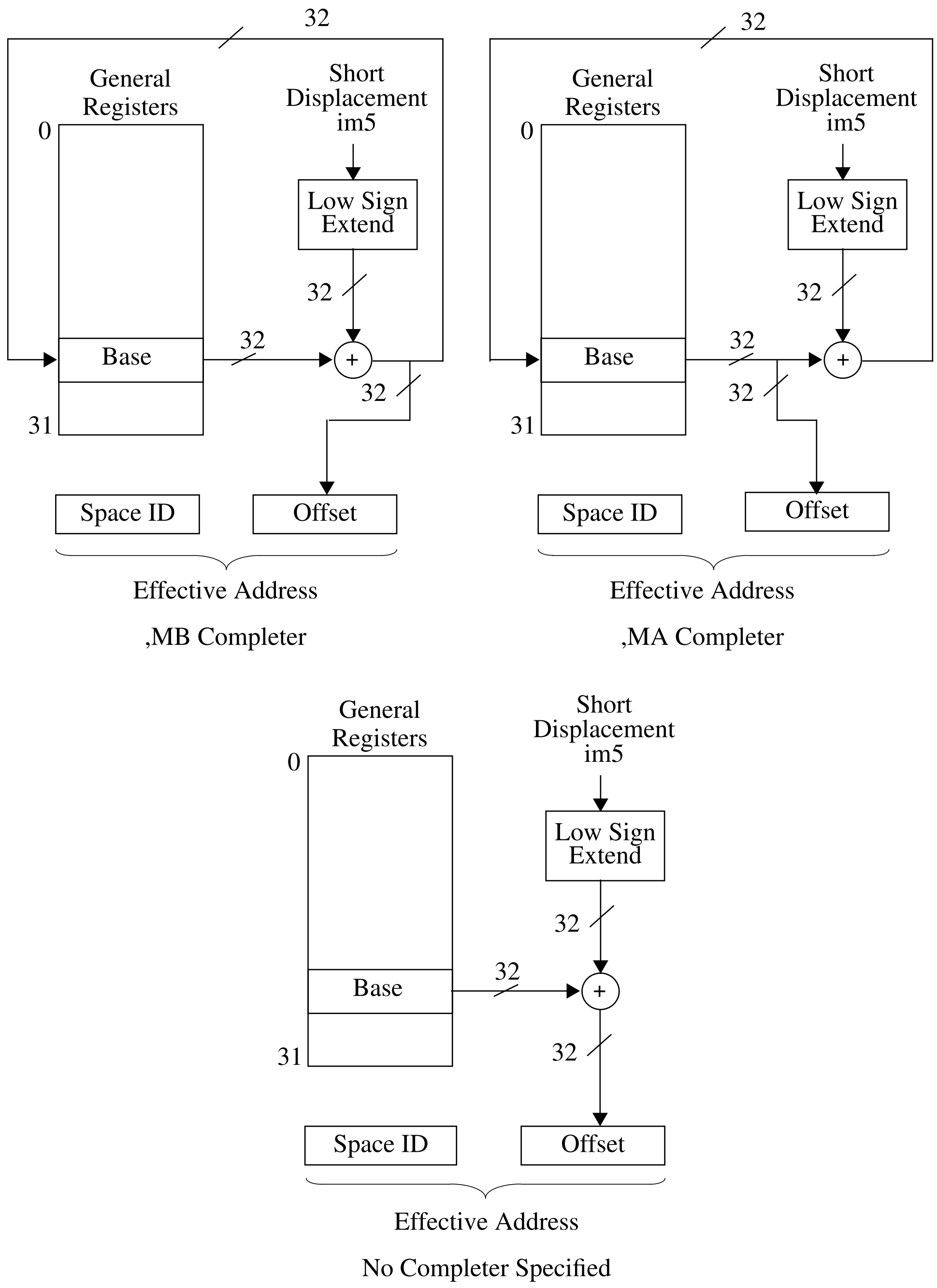


Figure 5-6. Short Displacement Loads and Stores

Store Bytes Short Instruction

STORE BYTES SHORT provides the means for doing unaligned byte moves efficiently. It uses a short 5-bit displacement to store bytes to unaligned destinations. The short displacement field is in two's complement notation. The sign bit is the rightmost bit of the field; the remaining bits are in the usual order.

The format of the STORE BYTES SHORT instruction is:

03	b	r	s	a	1	cc	C	m	im5
6	5	5	2	1	1	2	4	1	5

The *a* and *m* fields specify the following functions:

a = 0 store bytes beginning at the effective byte address in the word.
= 1 store bytes ending at the effective byte address in the word.

m = 0 no address modification.
= 1 address modification.

The *cc* field specifies the cache control hint (see Table 5-9 on page 5-18).

In the instruction descriptions that follow, some information is coded into the instruction names and the remainder is coded in the completer field (denoted by *cmplt* in the descriptions). Table 5-13 lists the assembly language syntax of the completer, the functions performed, and the values coded into the *a* and *m* bit fields of the instruction.

Table 5-13. Store Bytes Short Completers

cmplt	Description	a	m
<none> or B	Beginning case, don't modify base register	0	0
B,M	Beginning case, Modify base register	0	1
E	Ending case, don't modify base register	1	0
E,M	Ending case, Modify base register	1	1

In the above table, *cmplt* is in assembly language format and *a* and *m* are in machine language format.

The space identifier is computed like any other data memory reference (see Figure 5-2 on page 5-19). The calculation of the offset portion of the address for different completers is shown in Figure 5-7.

The actual offset and modified address involves some alignment and other considerations. Refer to the instruction description pages for an exact definition.

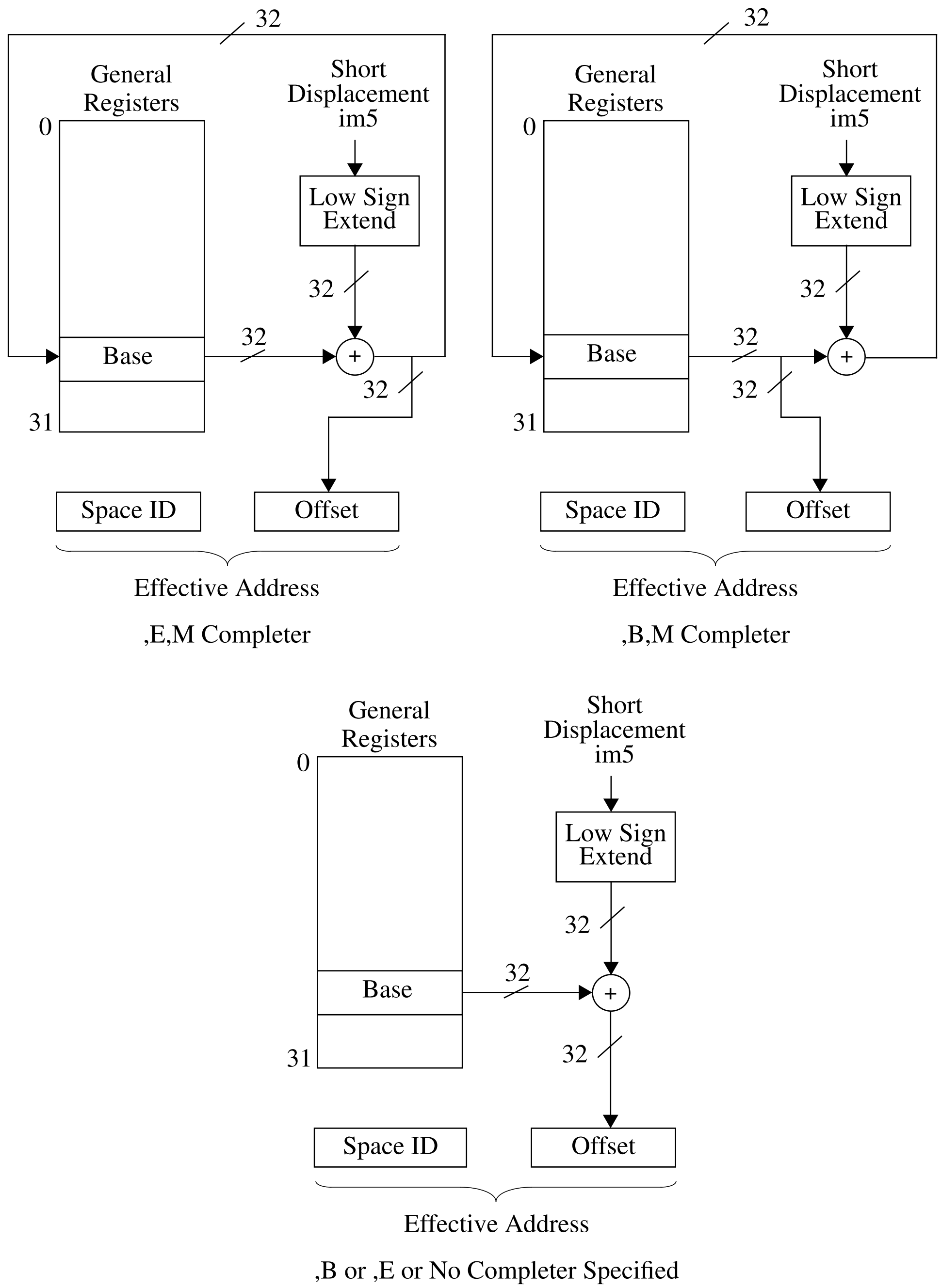


Figure 5-7. Store Bytes Short

LOAD WORD LDW

Format: LDW d(s,b),t

12	b	t	s	im14
6	5	5	2	14

Purpose: To load a word into a general register.

Description: The aligned word at the effective address is loaded into GR *t* from the effective address. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:

offset ← GR[b] + low_sign_ext(im14,14);
space ← space_select(s,GR[b]);
GR[t] ← mem_load(space,offset,0,31,NO_HINT);

Exceptions:

Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap

LOAD HALFWORD LDH

Format: LDH d(s,b),t

11	b	t	s	im14
6	5	5	2	14

Purpose: To load a halfword into a general register.

Description: The aligned halfword at the effective address is zero-extended and loaded into GR *t* from the effective address. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:

offset $\leftarrow$ GR[b] + low_sign_ext(im14,14);
space $\leftarrow$ space_select(s,GR[b]);
GR[t] $\leftarrow$ zero_ext(mem_load(space,offset,0,15,NO_HINT),16);

Exceptions:

Data TLB miss fault/data page fault	Unaligned data reference trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap

LOAD BYTE — LDB

Format: LDB d(s,b),t

10	b	t	s	im14
6	5	5	2	14

Purpose: To load a byte into a general register.

Description: The byte at the effective address is zero-extended and loaded into GR *t*. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:

```
offset ← GR[b] + low_sign_ext(im14,14);
space ← space_select(s,GR[b]);
GR[t] ← zero_ext(mem_load(space,offset,0,7,NO_HINT),8);
```

Exceptions:

Data TLB miss fault/data page fault	Page reference trap
Data memory access rights trap	Data debug trap
Data memory protection ID trap	

STORE WORD STW

Format: STW r,d(s,b)

1A	b	r	s	im14
6	5	5	2	14

Purpose: To store a word from a general register.

Description: GR *r* is stored in the aligned word at the effective address. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:
offset ← GR[b] + low_sign_ext(im14,14);
space ← space_select(s,GR[b]);
mem_store(space,offset,0,31,NO_HINT,GR[r]);

Exceptions:
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap

STORE HALFWORD — STH

Format: STH r,d(s,b)

19	b	r	s	im14
6	5	5	2	14

Purpose: To store a halfword from a general register.

Description: The right half of GR *r* is stored in the aligned halfword at the effective address. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:

```
offset ← GR[b] + low_sign_ext(im14,14);
space ← space_select(s,GR[b]);
mem_store(space,offset,0,15,NO_HINT,GR[r]{16..31});
```

Exceptions:

Data TLB miss fault/data page fault	Data memory break trap
Data memory access rights trap	TLB dirty bit trap
Data memory protection ID trap	Page reference trap
Unaligned data reference trap	Data debug trap

STORE BYTE STB

Format: STB r,d(s,b)

18	b	r	s	im14
6	5	5	2	14

Purpose: To store a byte from a general register.

Description: The rightmost byte of GR *r* is stored in the byte at the effective address. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:
offset ← GR[b] + low_sign_ext(im14,14);
space ← space_select(s,GR[b]);
mem_store(space,offset,0,7,NO_HINT,GR[r]{24..31})

Exceptions:
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap

LOAD WORD AND MODIFY **LDWM**

Format: LDWM d(s,b),t

13	b	t	s	im14
6	5	5	2	14

Purpose: To load a word into a general register and perform base register modification.

Description: The aligned word at the effective address is loaded into GR *t* from the effective address. The offset is either the base register, *b*, (positive displacement) or the base register plus the displacement, *d*, (negative displacement). The displacement is encoded into the *im14* field. Base register modification always occurs. If *b* = *t*, the value loaded is the aligned word at the effective address.

Operation:

```
if (low_sign_ext(im14,14) < 0)
      offset ← GR[b] + low_sign_ext(im14,14);
else
      offset ← GR[b];
space ← space_select(s,GR[b]);
GR[b] ← GR[b] + low_sign_ext(im14,14);
GR[t] ← mem_load(space,offset,0,31,NO_HINT);
```

Exceptions:

Data TLB miss fault/data page fault	Unaligned data reference trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap

STORE WORD AND MODIFY STWM

Format: STWM r,d(s,b)

1B	b	r	s	im14
6	5	5	2	14

Purpose: To store a word from a general register and perform base register modification.

Description: GR *r* is stored in the aligned word at the effective address. The offset is either the base register, *b*, (positive displacement) or the base register plus the displacement, *d*, (negative displacement). The displacement is encoded into the *im14* field. Base register modification always occurs. If *b* = *r*, the value stored at the effective address is the word from the source register before modification.

Operation:

```
if (low_sign_ext(im14,14) < 0)
    offset ← GR[b] + low_sign_ext(im14,14);
else
    offset ← GR[b];
space ← space_select(s,GR[b]);
mem_store(space,offset,0,31,NO_HINT,GR[r]);
GR[b] ← GR[b] + low_sign_ext(im14,14);
```

Exceptions:

Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap

LOAD WORD INDEXED LDWX

Format: LDWX,cmplt,cc x(s,b),t

03	b	x	s	u	0	cc	2	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a word into a general register.

Description: The aligned word at the effective address is loaded into GR *t*. The base register, *b*, and the index register, *x*, are combined to form the address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 2. This completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned word at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],2);         /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                           /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                           /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],2);
                break;
    default:    offset ← GR[b] + GR[x];                   /*u=0, m=0*/
                break;
}
GR[t] ← mem_load(space,offset,0,31,cc);
```

Exceptions:

Data TLB miss fault/data page fault	Unaligned data reference trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap

LOAD HALFWORD INDEXED LDHX

Format: LDHX,cmplt,cc x(s,b),t

03	b	x	s	u	0	cc	1	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a halfword into a general register.

Description: The aligned halfword at the effective address is zero-extended and loaded into GR *t*. The base register, *b*, and the index register, *x*, are combined to form the address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 1. This completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned halfword at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],1);        /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                          /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                          /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],1);
                break;
    default:    offset ← GR[b] + GR[x];                  /*u=0, m=0*/
                break;
}
GR[t] ← zero_ext(mem_load(space,offset,0,15,cc),16);
```

Exceptions:

Data TLB miss fault/data page fault	Unaligned data reference trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap

LOAD BYTE INDEXED LDBX

Format: LDBX,cmplt,cc x(s,b),t

03	b	x	s	u	0	cc	0	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a byte into a general register.

Description: The byte at the effective address is zero-extended and loaded into GR *t*. The base register, *b*, and the index register, *x*, are combined to form the address offset. The completer, *cmplt*, determines if the offset is the base register or the base register plus the index register. This completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the byte at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + GR[x];          /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                  /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                  /*u=1, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    default:    offset ← GR[b] + GR[x];          /*u=0, m=0*/
                break;
}
GR[t] ← zero_ext(mem_load(space,offset,0,7,cc),8);
```

Exceptions:
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Page reference trap
Data debug trap

LOAD WORD ABSOLUTE INDEXED — LDWAX

Format: LDWAX,cmplt,cc x(b),t

03	b	x	0	u	0	cc	6	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a word into a general register from an absolute address.

Description: The aligned word at the effective absolute address is loaded into GR *t* from the effective absolute address. The base register, *b*, and the index register, *x*, are combined to form the address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 2. This completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. This operation is only defined if the address is aligned on a 4 byte boundary. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned word at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Protection is not checked when this instruction is executed.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    switch (cmplt) {
        case S:     offset ← GR[b] + lshift(GR[x],2);          /*u=1, m=0*/
                    break;
        case M:     offset ← GR[b];                             /*u=0, m=1*/
                    GR[b] ← GR[b] + GR[x];
                    break;
        case SM:    offset ← GR[b];                             /*u=1, m=1*/
                    GR[b] ← GR[b] + lshift(GR[x],2);
                    break;
        default:    offset ← GR[b] + GR[x];                     /*u=0, m=0*/
                    break;
    }
    GR[t] ← phys_mem_load(offset,0,31,cc);
}
```

Exceptions: Privileged operation trap; Data debug trap

Restrictions: This instruction may be executed only at the most privileged level.

Level 0: Except for the privilege level restriction, this instruction functions identically to LOAD WORD INDEXED.

Format: LDCWX,cmplt,cc x(s,b),t

03	b	x	s	u	0	cc	7	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To read and lock a semaphore in main memory.

Description: The effective address is calculated. The base register, *b*, and the index register, *x*, are combined to form the address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 3. This completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned word at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-10 on page 5-18).

The address must be 16-byte aligned. If the address is unaligned, the operation of the instruction is undefined.

The remaining steps of the instruction are indivisible and non-interruptible. If a cache control hint is not specified, the instruction is performed as follows:

- If the cache line containing the effective address is not present in the cache or is present but not dirty, and the system is not fully coherent, the line is flushed, the addressed word is copied into GR *t*, and then set to zero in memory. If the line is retained in the cache, it must not be marked as dirty.
- If the cache line containing the effective address is present in the cache and is dirty, or the system is fully coherent, the semaphore operation may be handled as above or may be optimized by copying the addressed word into GR *t* and then setting the addressed word to zero in the cache.

If a cache control hint is specified, the semaphore operation may be handled as if a cache control hint had not been specified, or, preferably, the addressed word is copied into GR *t* and then the addressed word is set to zero in the cache. The cleared word need not be flushed to memory.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],3);        /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                          /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                          /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],3);
```

```
                    break;
     default:       offset ← GR[b] + GR[x];                            /*u=0, m=0*/
                    break;
}
(indivisible)
if (cache line is present and dirty || coherent_system || cc != 0) {
   GR[t] ← mem_load(space,offset,0,31,NO_HINT);
   mem_store(space,offset,0,31,NO_HINT,0);
} else {
   flush_data_cache(space, offset);
   GR[t] ← mem_load(space,offset,0,31,NO_HINT);
   store_in_memory(space,offset,0,31,NO_HINT,0);
}
```

Exceptions: Data TLB miss fault/data page fault; Data memory access rights trap; Data memory protection ID trap; Data memory break trap; TLB dirty bit trap; Page reference trap; Data debug trap

Restrictions: All software users of a semaphore must access the semaphore using the same cache control hint. Sharing a semaphore using different cache control hints is undefined.

Notes: Note that the “index shift” option for this instruction shifts by three, not two.

LOAD WORD SHORT **LDWS**

Format: LDWS,cmplt,cc d(s,b),t

03	b	im5	s	a	1	cc	2	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a word into a general register.

Description: The aligned word at the effective address is loaded into GR *t*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned word at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                                 /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);            /*m=0*/
                break;
}
GR[t] ← mem_load(space,offset,0,31,cc);
```

Exceptions:

Data TLB miss fault/data page fault	Unaligned data reference trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap

LOAD HALFWORD SHORT **LDHS**

Format: LDHS,cmplt,cc d(s,b),t

03	b	im5	s	a	1	cc	1	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a halfword into a general register.

Description: The aligned halfword at the effective address is zero-extended and loaded into GR *t*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned halfword at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                                /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);          /*m=0*/
                break;
}
GR[t] ← zero_ext(mem_load(space,offset,0,15,cc),16);
```

Exceptions:

Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap

LOAD BYTE SHORT LDBS

Format: LDBS,cmplt,cc d(s,b),t

03	b	im5	s	a	1	cc	0	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a byte into a general register.

Description: The byte at the effective address is zero-extended and loaded into GR *t*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the byte at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                                /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);               /*m=0*/
                break;
}
GR[t] ← zero_ext(mem_load(space,offset,0,7,cc),8);
```

Exceptions:

Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Page reference trap
Data debug trap

LOAD WORD ABSOLUTE SHORT — LDWAS

Format: LDWAS,cmplt,cc d(b),t

03	b	im5	0	a	1	cc	6	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To load a word into a general register from an absolute address.

Description: The aligned word at the effective absolute address is loaded into GR *t*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. The operation is only defined if the address is aligned on a 4-byte boundary. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned word at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Protection is not checked when this instruction is executed.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    switch (cmplt) {
        case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                    GR[b] ← GR[b] + low_sign_ext(im5,5);
                    break;
        case MA:    offset ← GR[b];                                /*a=0, m=1*/
                    GR[b] ← GR[b] + low_sign_ext(im5,5);
                    break;
        default:    offset ← GR[b] + low_sign_ext(im5,5);              /*m=0*/
                    break;
    }
    GR[t] ← phys_mem_load(offset,0,31,cc);
}
```

Exceptions: Privileged operation trap; Data debug trap

Restrictions: This instruction may be executed only at the most privileged level.

Level 0: Except for the privilege level restriction, this instruction functions identically to LOAD WORD SHORT.

LOAD AND CLEAR WORD SHORT **LDCWS**

Format: LDCWS,cmplt,cc d(s,b),t

03	b	im5	s	a	1	cc	7	m	t
6	5	5	2	1	1	2	4	1	5

Purpose: To read and lock a semaphore in main memory.

Description: The effective address is calculated. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. This completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *t*, the value loaded is the aligned word at the effective address.

The completer, *cc*, specifies the cache control hint (see Table 5-10 on page 5-18).

The address must be 16-byte aligned. If the address is unaligned, the operation of the instruction is undefined.

The remaining steps of the instruction are indivisible and non-interruptible. If a cache control hint is not specified, the instruction is performed as follows:

- If the cache line containing the effective address is not present in the cache or is present but not dirty, and the system is not fully coherent, the line is flushed, the addressed word is copied into GR *t*, and then set to zero in memory. If the line is retained in the cache, it must not be marked as dirty.
- If the cache line containing the effective address is present in the cache and is dirty, or the system is fully coherent, the semaphore operation may be handled as above or may be optimized by copying the addressed word into GR *t* and then setting the addressed word to zero in the cache.

If a cache control hint is specified, the semaphore operation may be handled as if a cache control hint had not been specified, or, preferably, the addressed word is copied into GR *t* and then the addressed word is set to zero in the cache. The cleared word need not be flushed to memory.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                              /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);        /*m=0*/
                break;
```

```
}
(indivisible)
if (cache line is present and dirty || coherent_system || cc != 0) {
  GR[t] ← mem_load(space,offset,0,31,NO_HINT);
  mem_store(space,offset,0,31,NO_HINT,0);
} else {
  flush_data_cache(space, offset);
  GR[t] ← mem_load(space,offset,0,31,NO_HINT);
  store_in_memory(space,offset,0,31,NO_HINT,0);
}
```

Exceptions: Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap

Restrictions: All software users of a semaphore must access the semaphore using the same cache control hint. Sharing a semaphore using different cache control hints is undefined.

STORE WORD SHORT — STWS

Format: STWS,cmplt,cc r,d(s,b)

03	b	r	s	a	1	cc	A	m	im5
6	5	5	2	1	1	2	4	1	5

Purpose: To store a word from a general register.

Description: GR *r* is stored in the aligned word at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *r*, the value stored at the effective address is the word from the source register before modification.

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:   offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
               mem_store(space,offset,0,31,cc,GR[r]);
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    case MA:   offset ← GR[b];                                 /*a=0, m=1*/
               mem_store(space,offset,0,31,cc,GR[r]);
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    default:   offset ← GR[b] + low_sign_ext(im5,5);          /*m=0*/
               mem_store(space,offset,0,31,cc,GR[r]);
               break;
}
```

Exceptions:

Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap

STORE HALFWORD SHORT STHS

Format: STHS,cmplt,cc r,d(s,b)

03	b	r	s	a	1	cc	9	m	im5
6	5	5	2	1	1	2	4	1	5

Purpose: To store a halfword from a general register.

Description: The right half of GR *r* is stored in the aligned halfword at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *r*, the value stored at the effective address is the rightmost halfword from the source register before modification.

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:   offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
               mem_store(space,offset,0,15,cc,GR[r]{16..31});
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    case MA:   offset ← GR[b];                                /*a=0, m=1*/
               mem_store(space,offset,0,15,cc,GR[r]{16..31});
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    default:   offset ← GR[b] + low_sign_ext(im5,5);              /*m=0*/
               mem_store(space,offset,0,15,cc,GR[r]{16..31});
               break;
}
```

Exceptions:

Data TLB miss fault/data page fault	Data memory break trap
Data memory access rights trap	TLB dirty bit trap
Data memory protection ID trap	Page reference trap
Unaligned data reference trap	Data debug trap

STORE BYTE SHORT **STBS**

Format: STBS,cmplt,cc r,d(s,b)

03	b	r	s	a	1	cc	8	m	im5
6	5	5	2	1	1	2	4	1	5

Purpose: To store a byte from a general register.

Description: The rightmost byte of GR *r* is stored in the byte at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *r*, the value stored at the effective address is the rightmost byte from the source register before modification.

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                mem_store(space,offset,0,7,cc,GR[r]{24..31});
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                                /*a=0, m=1*/
                mem_store(space,offset,0,7,cc,GR[r]{24..31});
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);              /*m=0*/
                mem_store(space,offset,0,7,cc,GR[r]{24..31});
                break;
}
```

Exceptions:

Data TLB miss fault/data page fault	TLB dirty bit trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap
Data memory break trap	

STORE WORD ABSOLUTE SHORT — STWAS

Format: STWAS,cmplt,cc r,d(b)

03	b	r	0	a	1	cc	E	m	im5
6	5	5	2	1	1	2	4	1	5

Purpose: To store a word from a general register to an absolute address.

Description: GR *r* is stored in the aligned word at the effective absolute address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. The operation is only defined if the address is aligned on a 4-byte boundary. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *r*, the value stored at the effective address is the word from the source register before modification.

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Protection is not checked when this instruction is executed.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    switch (cmplt) {
        case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                    phys_mem_store(offset,0,31,cc,GR[r]);
                    GR[b] ← GR[b] + low_sign_ext(im5,5);
                    break;
        case MA:    offset ← GR[b];                                /*a=0, m=1*/
                    phys_mem_store(offset,0,31,cc,GR[r]);
                    GR[b] ← GR[b] + low_sign_ext(im5,5);
                    break;
        default:    offset ← GR[b] + low_sign_ext(im5,5);          /*m=0*/
                    phys_mem_store(offset,0,31,cc,GR[r]);
                    break;
    }
}
```

Exceptions: Privileged operation trap; Data debug trap

Restrictions: This instruction may be executed only at the most privileged level.

Level 0: Except for the privilege level restriction, this instruction functions identically to STORE WORD SHORT.

STORE BYTES SHORT — STBYS

Format: STBYS,cmplt,cc r,d(s,b)

03	b	r	s	a	1	cc	C	m	im5
6	5	5	2	1	1	2	4	1	5

Purpose: To implement the beginning, middle, and ending cases for fast byte moves with unaligned sources and destinations.

Description: If the PSW[E] bit is 0 and begin (modifier ",B" corresponding to *a* = 0) is specified, the rightmost bytes of GR *r* are stored in memory starting at the byte whose address is given by the effective address. The number of bytes stored is sufficient to fill out the word containing the byte addressed by the effective address.

If the PSW[E] bit is 0 and end (modifier ",E" corresponding to *a* = 1) is specified, the leftmost bytes of GR *r* are stored in memory starting at the leftmost byte in the word specified by the effective address, and continuing until (but not including) the byte specified by the effective address. When the effective address specifies the leftmost byte in a word, nothing is stored, but protection is checked and the cache line is marked as *dirty.*

If the PSW[E] bit is 1 and begin (modifier ",B" corresponding to *a* = 0) is specified, the leftmost bytes of GR *r* are stored in memory starting at the byte whose address is given by the effective address. The number of bytes stored is sufficient to fill out the word containing the byte addressed by the effective address.

If the PSW[E] bit is 1 and end (modifier ",E" corresponding to *a* = 1) is specified, the rightmost bytes of GR *r* are stored in memory starting at the leftmost byte in the word specified by the effective address, and continuing until (but not including) the byte specified by the effective address. When the effective address specifies the leftmost byte in a word, nothing is stored, but protection is checked and the cache line is marked as *dirty.*

If base register modification is specified through completer ",M", GR *b* is updated and then truncated to a word address. (See Table 5-13 on page 5-26 for the assembly language completer mnemonics.) If base register modification is specified and *b* = *r*, the value stored at the effective address is the bytes from the source register before modification.

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18). If the first byte of the addressed cache line is not written to, the processor must perform the store as if the cache control hint had not been specified.

Operation:

```
space ← space_select(s,GR[b]);
if (cmplt == B,M)                                         /*a=0, m=1*/
     offset ← GR[b];
else
     offset ← GR[b] + low_sign_ext(im5,5);
pos ← 8*(offset & 0x3);
offset ← offset & 0xFFFFFFFC;
```

```
switch (cmplt) {
    case B:                                                          /*a=0, m=0*/
                if (PSW[E] == 0)
                    mem_store(space,offset,pos,31,cc,GR[r]{pos..31});
                else
                    mem_store(space,offset,pos,31,cc,GR[r]{0..31-pos});
                break;
    case E:                                                          /*a=1, m=0*/
                if (PSW[E] == 0)
                    mem_store(space,offset,0,pos-1,cc,GR[r]{0..pos-1});
                else
                    mem_store(space,offset,0,pos-1,cc,GR[r]{32-pos..31});
                break;
    case B,M:                                                        /*a=0, m=1*/
                if (PSW[E] == 0)
                    mem_store(space,offset,pos,31,cc,GR[r]{pos..31});
                else
                    mem_store(space,offset,pos,31,cc,GR[r]{0..31-pos});
                GR[b] ← (GR[b] + low_sign_ext(im5,5)) & 0xFFFFFFFC;
                break;
    case E,M:                                                        /*a=1, m=1*/
                if (PSW[E] == 0)
                    mem_store(space,offset,0,pos-1,cc,GR[r]{0..pos-1});
                else
                    mem_store(space,offset,0,pos-1,cc,GR[r]{32-pos..31});
                GR[b] ← (GR[b] + low_sign_ext(im5,5)) & 0xFFFFFFFC;
                break;
}
```

Exceptions:

Data TLB miss fault/data page fault	TLB dirty bit trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap
Data memory break trap	

Notes: All 32 bits of the original virtual offset are saved to IOR (CR21) if this instruction traps.

For this instruction, the low 2 bits of the virtual offset are masked to 0 when comparing against the contents of the data breakpoint address offset registers.

PROGRAMMING NOTE

The STBYS instruction with the ',E' completer and the effective address specifying the leftmost byte of the word may be used to implement a memory scrubbing operation. This is possible because the line is marked *dirty* but the contents are not modified.

Immediate Instructions

The immediate instructions do not reference memory. They compute values either from a shifted long immediate (21 bits long), from a shifted long immediate and a source register, or from a base register plus a 14-bit displacement. This computed value is then stored in another general register. These instructions are typically used to compute the values of addresses of data items. The LOAD OFFSET instruction can also be used to simply load a 14-bit immediate into a register.

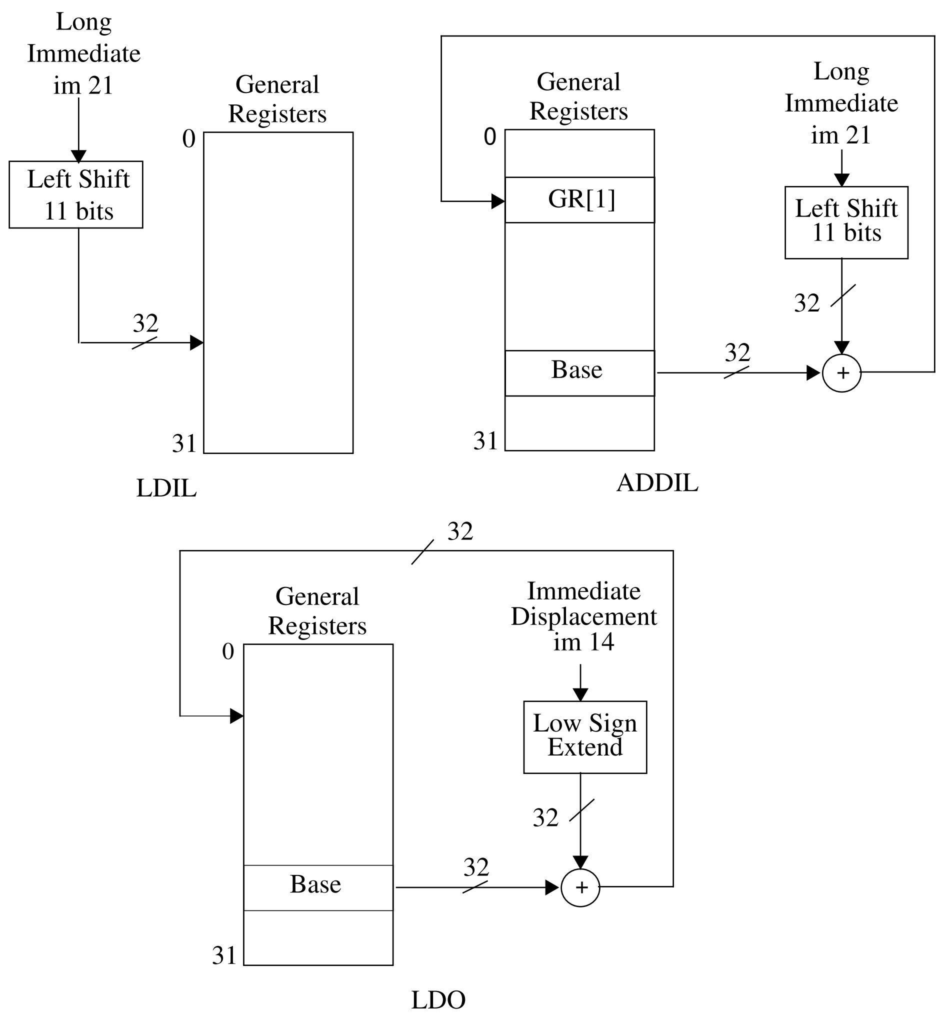


Figure 5-8. Immediate Instructions

LOAD OFFSET LDO

Format: LDO d(b),t

0D	b	t	rv	im14
6	5	5	2	14

Purpose: To load an offset into a general register.

Description: The effective address is calculated, and its offset part is loaded into GR *t*. The displacement *d* is encoded into the *im14* field.

Operation: GR[t] ← GR[b] + low_sign_ext(im14,14);

Exceptions: None

Notes: Memory is not referenced. The LDI i,t pseudo-operation generates an LDO i(0),t instruction to load a 14-bit immediate value into a register.

LOAD IMMEDIATE LEFT — LDIL

Format: LDIL i,t

08	t	im21
6	5	21

Purpose: To load an immediate value into the left part of a general register.

Description: The 21-bit immediate value, *i*, is assembled, padded on the right with 11 zero bits, and loaded into GR *t*.

Operation: GR[t] $\leftarrow$ lshift(assemble_21(im21),11);

Exceptions: None

Notes: Memory is not referenced.

PROGRAMMING NOTE

LOAD IMMEDIATE LEFT can be used to generate a 32-bit literal in an arbitrary general register *t* by the following sequence of assembly language code:

```
LDIL    l%literal,GRt
LDO     r%literal(GRt),GRt
```

ADD IMMEDIATE LEFT **ADDIL**

Format: ADDIL i,r

0A	r	im21
6	5	21

Purpose: To add the left part of a long displacement to a general register.

Description: The 21-bit immediate value, *i*, from *im21* is assembled, padded on the right with 11 zero bits, and added to GR *r*. The result is placed in GR 1. Overflow, if it occurs, is ignored. The immediate value is encoded into the *im21* field.

Operation: GR[1] $\leftarrow$ lshift(assemble_21(im21),11) + GR[r];

Exceptions: None

PROGRAMMING NOTE

ADD IMMEDIATE LEFT can be used to perform a load or store with a 32-bit displacement. For example, to load a word from memory into general register *t* with a 32-bit displacement, the following sequence of assembly language code could be used:

```
ADDIL   l%literal,GRb
LDW     r%literal(0,GR1),GRt
```

Branch Instructions

Branch instructions are classified into three major categories: unconditional local branches, unconditional external branches, and conditional local branches. Within these categories there is sub-classification based on how the target address is computed, whether or not a return address is saved, and whether or not privilege changes can occur. Not all of the options are available for each category. The following sections describe the types of branches. The operation of each branch instruction is detailed in the instruction description sections in this chapter.

Unconditional Local Branches

The unconditional local branch instructions are used for intraspace control transfers, procedure calls, and procedure returns. Three types of relative addressing are provided:

1. IA relative branches with static displacement use the IAOQ_Front plus a 17-bit signed word displacement. This allows a branch target range of up to plus or minus 256 Kbytes within a space.
2. IA relative branches with dynamic displacement use the IAOQ_Front plus a shifted index register.
3. Base relative branches with dynamic displacement use the value in a base register plus a shifted index register.

BRANCH AND LINK is used for procedure calls. The branch target address is IA relative with a static displacement. It places the offset of the return point in the specified GR. The return point is the location four bytes beyond the address of the instruction which executes after the BRANCH AND LINK. BRANCH AND LINK also satisfies most requirements for unconditional branching when GR 0 is specified as the link register.

GATEWAY is used for intraspace branching with a process privilege level promotion. The branch target address is IA relative with a static displacement.

BRANCH AND LINK REGISTER is used for intraspace procedure calls in which the branch target is outside the range for BRANCH AND LINK or when a dynamic target displacement is needed. The branch target address is IA relative with a dynamic displacement. Link handling is performed the same way as for the BRANCH AND LINK instruction.

BRANCH VECTORED is used for intraspace branching through a table and for procedure returns. The branch target is base relative with a dynamic displacement. The process privilege level may be demoted.

Unconditional External Branches

The unconditional external branch instructions are used for interspace control transfers, procedure calls, and procedure returns. All unconditional external branch instructions use base-relative addressing with static displacements and may demote the process privilege level based on the rightmost bits of the base register. The target address is the value in a base register plus a 17-bit signed word displacement. This allows for a plus or minus 256 Kbyte branch range across space boundaries.

BRANCH AND LINK EXTERNAL is used for interspace procedure calls. It places the offset of the return point in GR 31 and copies the space ID into SR 0. The return point is the location four bytes beyond the address of the instruction which executes after the branch.

BRANCH EXTERNAL is used for interspace branching and procedure returns. The return address is not saved in this instruction.

In Level 0 systems, unconditional external branch instructions are executed as in non-Level 0 systems, except that the IASQ and SR 0 are nonexistent registers and updating them has no effect.

Conditional Local Branches

The conditional local branch instructions are used to perform an operation and then branch if the condition specified is satisfied. All conditional local branch instructions use IA relative addresses with static displacements. The target address is the current IAOQ_Front plus a 12-bit signed word displacement. This allows for a plus or minus 8-Kbyte branch target range within a space.

There are four categories of conditional local branch instructions: move and branch, compare and branch, add and branch, and branch on bits. The branch may be taken if the condition specified is true or false. There are two forms of each instruction, the two-register form and the register plus 5-bit immediate form. The 5-bit immediate operand provides data values in the range from -16 to +15.

Branch Characteristics

Figure 5-9 categorizes the characteristics of the branch instructions.

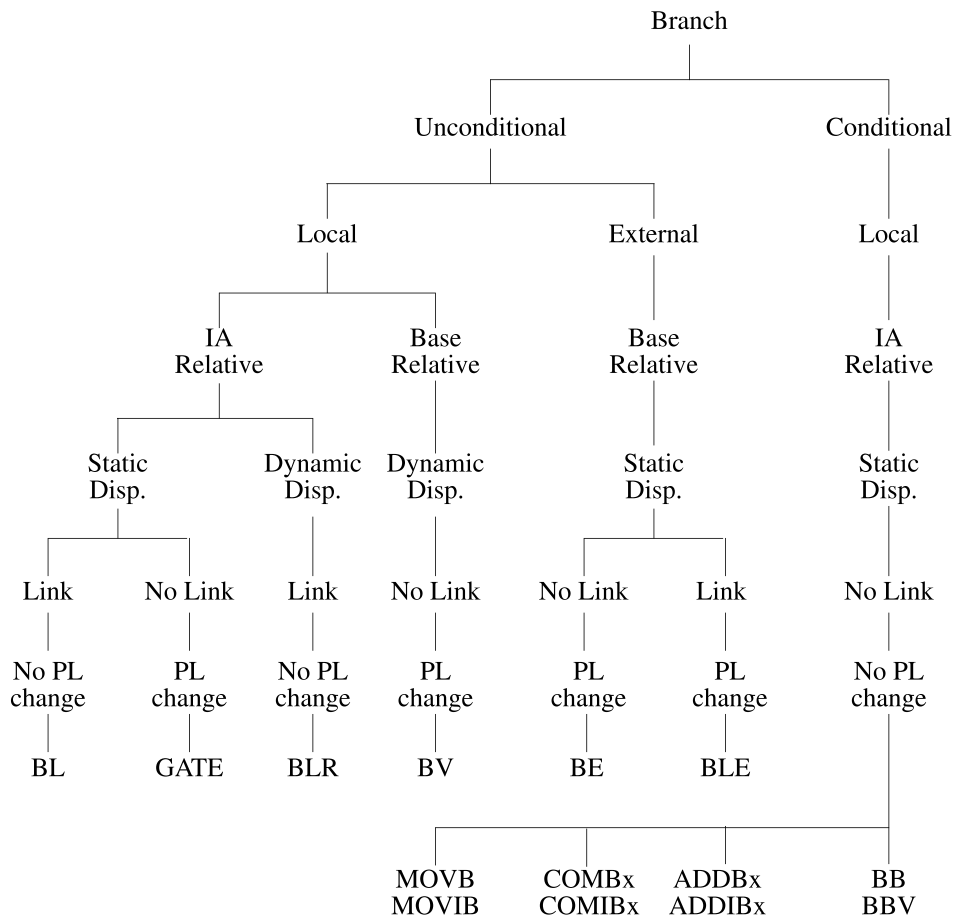


Figure 5-9. Classification of Branch Instructions

PROGRAMMING NOTE

Example instruction sequences which perform the different types of procedure calls are shown below. The examples illustrate ways to use offsets of different lengths. The simplest case is that of intraspace calls which can be done by any of the following code sequences, assuming that the convention that SR 4 tracks IASQ is observed:

```
call:    BL      target,rp        or    LDIL    l%target,rp
                                        BLE     r%target(SR4,rp)
                                        OR      GR31,0,rp

return:  BV      0(rp)            or    BE      0(SR0,rp)
```

Making interspace calls which might decrease privilege level is shown below:

```
call:    LDW     space_id,GR1           ; space id for target
         MTSP    GR1,SR4
         LDIL    l%target,rp
         BLE     target(SR4,rp)
         OR      GR31,0,rp

return:  BE      0(SR0,rp)
```

Calling a procedure which must run with less privilege than the caller requires setting the desired privilege level in the two rightmost bits of the base register used in the BRANCH AND LINK EXTERNAL. Returning to the more-privileged routine will succeed only if the return point is a GATEWAY instruction.

BRANCH AND LINK BL

Format: BL,n target,t

3A	t	w1	0	w2	n	w
6	5	5	3	11	1	1

Purpose: To do IA relative branches and procedure calls with a static displacement.

Description: The word displacement is assembled from the *w*, *w1*, and *w2* fields in the instruction. The displacement is sign extended, and the result plus 8 is added to the offset of the current instruction to form the target offset. The offset of the return point is placed in GR *t*. The return point is 4 bytes beyond the following instruction.

The instruction following the BRANCH AND LINK will be executed unless nullification is requested. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w, w1,* and *w2* fields.

Operation:
```
disp ← lshift(sign_ext(assemble_17(w1,w2,w),17),2);
IAOQ_Next ← IAOQ_Front + disp + 8;
GR[t] ← IAOQ_Back + 4;
if (n) PSW[N] ← 1;
```

Exceptions: Taken branch trap

Notes: To perform an unconditional branch without saving a link, the B pseudo-operation allows the coding of BRANCH AND LINK with GR 0 as the link register.

GATEWAY

Format: GATE,n target,t

3A	t	w1	1	w2	n	w
6	5	5	3	11	1	1

Purpose: To change privilege level and do an IA relative branch with a static displacement.

Description: The word displacement is assembled from the *w*, *w1*, and *w2* fields in the instruction. The displacement is sign extended and the result plus 8 is added to the offset of the current instruction to form the target offset. The instruction following the GATEWAY instruction will be executed unless nullification is requested. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w*, *w1*, *w2* fields.

If the PSW C-bit is 1, the privilege level is changed to that given by the two rightmost bits of the type field in the TLB entry for the page (when the type field is greater than 3) from which the GATEWAY instruction is fetched if that results in a higher privilege. If privilege is not increased, then the current privilege is used at the target. In all cases, the privilege level of the GATEWAY instruction is deposited into bits 30..31 of GR *t*. The privilege change must occur for the target of the GATEWAY, but the delay slot may be fetched and/or executed at either privilege. If the PSW C-bit is 0, the privilege level is changed to 0.

An illegal instruction trap is taken if a GATEWAY instruction is attempted and the PSW B-bit is 1.

Operation:

```
if (PSW[B])
    illegal_instruction_trap;
else {
    disp ← lshift(sign_ext(assemble_17(w1,w2,w),17),2);
    GR[t] ← cat(GR[t]{0..29},IAOQ_Front{30..31});
    if (PSW[C] && !level_0) {
        search_ITLB(IASQ_Front,IAOQ_Front,&entry);
        if (ITLB[entry].ACC_RIGHTS{0..2} <= 3)
            priv ← IAOQ_Front{30..31};
        else
            priv ← min(IAOQ_Front{30..31},
                        ITLB[entry].ACC_RIGHTS{1..2});
    } else
        priv ← 0;
    IAOQ_Next{0..29} ← (IAOQ_Front + disp + 8){0..29};
    IAOQ_Next{30..31} ← priv;
    if (n) PSW[N] ← 1;
}
```

Exceptions: Illegal instruction trap
Taken branch trap

Notes: The privilege information must be captured when the TLB is read for instruction fetch and that information kept for the determination of the new execution privilege.

PROGRAMMING NOTE

The privilege level checking for fetching and executing the instruction following a GATEWAY (which might be in a different page from the GATEWAY itself) may be done against either the old or the new privilege level. Software should ensure that both checks are equally valid.

It is possible for a GATEWAY to promote the privilege level so that the process cannot continue executing on that page (because it violates PL2 of the TLB access rights field). In that case, software should ensure that the GATEWAY nullifies execution of the following instruction and its target should be on a page whose range of execute levels includes the new privilege level. Otherwise, an instruction memory protection trap may result.

Level 0: This instruction promotes the privilege level to 0.

BRANCH AND LINK REGISTER BLR

Format: BLR,n x,t

3A	t	x	2	0	n	0
6	5	5	3	11	1	1

Purpose: To do IA relative branches with a dynamic displacement and store a return link.

Description: The index from GR *x* is shifted left 3 bits and the result plus 8 is added to the offset of the current instruction to form the target offset. The offset of the return point is placed in GR *t*. The return point is 4 bytes beyond the following instruction.

The instruction following the BRANCH AND LINK REGISTER instruction will be executed unless nullification is requested.

Operation: IAOQ_Next ← IAOQ_Front + lshift(GR[x],3) + 8;
GR[t] ← IAOQ_Back + 4;
if (n) PSW[N] ← 1;

Exceptions: Taken branch trap

PROGRAMMING NOTE

BRANCH AND LINK REGISTER with GR 0 as the link register does a IA relative branch without saving a link. Jump tables based on the index value can be constructed using this instruction. When the jump table begins at the instruction which is located at the BLR plus 8 bytes, an index value of 0 can be used to branch to the first entry of the table.

BRANCH VECTORED BV

Format: BV,n x(b)

3A	b	x	6	0	n	0
6	5	5	3	11	1	1

Purpose: To do base-relative branches with a dynamic displacement in the same space.

Description: The index from GR *x* is shifted left by 3 bits. The result is added to GR *b* and the sum becomes the new offset of the target instruction.

The instruction following the BRANCH VECTORED instruction will be executed unless nullification is specified.

If the two rightmost bits of GR *b* designate a lower privilege level than the current privilege level, then the privilege level of the target is set to that specified by the rightmost bits of GR *b*. The decrease in privilege level takes effect at the branch target, not for the instruction in the delay slot.

Operation:

```
IAOQ_Next{0..29} ← (GR[b] + lshift(GR[x],3)){0..29};
if (IAOQ_Front{30..31} < GR[b]{30..31})
    IAOQ_Next{30..31} ← GR[b]{30..31};
else
    IAOQ_Next{30..31} ← IAOQ_Front{30..31};
if (n) PSW[N] ← 1;
```

Exceptions: Taken branch trap

Level 0: This instruction demotes the privilege level to any nonzero value, if it changes it.

BRANCH EXTERNAL BE

Format: BE,n wd(sr,b)

38	b	w1	s	w2	n	w
6	5	5	3	11	1	1

Purpose: To do branches and returns to another space.

Description: The word displacement, *wd*, is assembled from the *w*, *w1*, and *w2* fields in the instruction and sign extended. The result is added to GR *b* and the sum becomes the offset of the target instruction. SR *sr* (which is assembled from the *s* field of the instruction) becomes the space of the target instruction.

If the two rightmost bits of GR *b* designate a less privileged level than the current instruction, the privilege level of the target is set to that specified by the rightmost bits of GR *b*. The decrease in privilege level takes effect at the branch target, not for the instruction in the delay slot. When a BRANCH EXTERNAL is executed with the PSW C-bit 0 (code address translation is disabled) the effect on IASQ is not defined.

Operation:

```
disp ← lshift(sign_ext(assemble_17(w1,w2,w),17),2);
IAOQ_Next{0..29} ← (GR[b] + disp){0..29};
if (IAOQ_Front{30..31} < GR[b]{30..31})
    IAOQ_Next{30..31} ← GR[b]{30..31};
else
    IAOQ_Next{30..31} ← IAOQ_Front{30..31};
IASQ_Next ← SR[assemble_3(s)];
if (n) PSW[N] ← 1;
```

Exceptions: Taken branch trap

Level 0: This instruction executes as usual, except that the IASQ is a nonexistent register and updating it has no effect. Also, the privilege level is demoted to any nonzero value, if the instruction changes it.

PROGRAMMING NOTE

If a taken local branch is executed following a BRANCH EXTERNAL instruction, the target's address is computed based on the value of the IASQ set by the BRANCH EXTERNAL instruction. This results in a transfer of control to possibly a meaningless location in the new space.

BRANCH AND LINK EXTERNAL BLE

Format: BLE,n wd(sr,b)

39	b	w1	s	w2	n	w
6	5	5	3	11	1	1

Purpose: To do procedure calls to another space.

Description: The word displacement, *wd*, is assembled from the *w*, *w1*, and *w2* fields in the instruction and sign extended. The result is added to GR *b* and the sum becomes the offset of the target instruction. SR *sr* (which is assembled from the *s* field of the instruction) becomes the space of the target instruction. The offset of the return point is placed in GR 31 and the space portion of the following instruction's address is placed in SR 0. The return point is 4 bytes beyond the following instruction.

If the two rightmost bits of GR *b* designate a less privileged level than the current instruction, the privilege level of the target is set to that specified by the rightmost bits of GR *b*. The decrease in privilege level takes effect at the branch target, not for the instruction in the delay slot. When a BRANCH AND LINK EXTERNAL is executed with the PSW C-bit 0 (code address translation is disabled) the effects on IASQ and SR 0 is not defined.

Operation:

```
disp ← lshift(sign_ext(assemble_17(w1,w2,w),17),2);
IAOQ_Next{0..29} ← (GR[b] + disp){0..29};
if (IAOQ_Front{30..31} < GR[b]{30..31})
    IAOQ_Next{30..31} ← GR[b]{30..31};
else
    IAOQ_Next{30..31} ← IAOQ_Front{30..31};
IASQ_Next ← SR[assemble_3(s)];
GR[31] ← IAOQ_Back + 4;
SR[0] ← IASQ_Back;
if (n) PSW[N] ← 1;
```

Exceptions: Taken branch trap

Level 0: This instruction executes as usual, except that the IASQ and SR 0 are nonexistent registers and updating them has no effect. Also, the privilege level is demoted to any nonzero value, if the instruction changes it.

MOVE AND BRANCH MOVB

Format: MOVB,cond,n r1,r2,target

32	r2	r1	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To copy one register to another and perform an IA relative branch conditionally based on the value moved.

Description: GR *r1* is copied into GR *r2*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied by the value moved, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, then the following instruction is not nullified. If nullification is specified, then the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the extract/deposit conditions shown in Table 5-7 on page 5-7 (never, =, <, OD, TR, <>, >=, EV). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the value moved satisfies the specified condition and set to 0 otherwise.

Operation:

```
GR[r2] ← GR[r1];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

MOVE IMMEDIATE AND BRANCH — MOVIB

Format: MOVIB,cond,n i,r,target

33	r	im5	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To copy an immediate value into a register and perform an IA relative branch conditionally based on the value moved.

Description: The immediate value *im5* is sign extended and copied into GR *r*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied by the value moved, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the extract/deposit conditions shown in Table 5-7 on page 5-7 (never, =, <, OD, TR, <>, >=, EV). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the value moved satisfies the specified condition and set to 0 otherwise.

Operation:

```
GR[r] ← low_sign_ext(im5,5);
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

PROGRAMMING NOTE

Since *i* is known at the time a MOVE IMMEDIATE AND BRANCH instruction is written, conditions other than always and never (the ",TR" and <none> completers) are of no use.

COMPARE AND BRANCH IF TRUE — COMBT

Format: COMBT,cond,n r1,r2,target

20	r2	r1	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To compare two values and perform an IA relative branch conditionally based on the values compared.

Description: GR *r1* is compared with GR *r2*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied by the values compared, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated compare or subtract conditions shown in Table 5-3 on page 5-5 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:

```
GR[r1] + ~GR[r2] + 1;
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The COMB pseudo-operation allows the coding of both true and false conditions and generates either a COMBT or COMBF instruction.

COMPARE AND BRANCH IF FALSE COMBF

Format: COMBF,cond,n r1,r2,target

22	r2	r1	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To compare two values and perform an IA relative branch conditionally based on the values compared.

Description: GR *r1* is compared with GR *r2*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is not satisfied by the values compared, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated compare or subtract conditions shown in Table 5-3 on page 5-5 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:

```
GR[r1] + ~GR[r2] + 1;
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (!cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← cond_satisfied;
    else
        PSW[N] ← !cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The COMB pseudo-operation allows the coding of both true and false conditions and generates either a COMBT or COMBF instruction.

Format: COMIBT,cond,n i,r,target

21	r	im5	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To compare two values and perform an IA relative branch conditionally based on the values compared.

Description: The sign-extended immediate value *im5* is compared with GR *r*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied by the values compared, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated compare or subtract conditions shown in Table 5-3 on page 5-5 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:

```
low_sign_ext(im5,5) + ~GR[r] + 1;
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The COMIB pseudo-operation allows the coding of both true and false conditions and generates either a COMIBT or COMIBF instruction.

COMPARE IMMEDIATE AND BRANCH IF FALSE — COMIBF

Format: COMIBF,cond,n i,r,target

23	r	im5	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To compare two values and perform an IA relative branch conditionally based on the values compared.

Description: The sign-extended immediate value *im5* is compared with GR *r*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is not satisfied by the values compared, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated compare or subtract conditions shown in Table 5-3 on page 5-5 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:

```
low_sign_ext(im5,5) + ~GR[r] + 1;
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (!cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← cond_satisfied;
    else
        PSW[N] ← !cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The COMIB pseudo-operation allows the coding of both true and false conditions and generates either a COMIBT or COMIBF instruction.

ADD AND BRANCH IF TRUE — ADDBT

Format: ADDBT,cond,n r1,r2,target

28	r2	r1	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To add two values and perform an IA relative branch conditionally based on the values added.

Description: GR *r1* and GR *r2* are added and the result is stored in GR *r2*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied by the values added, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated add conditions shown in Table 5-4 on page 5-5 (never, =, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values added satisfy the specified condition and set to 0 otherwise.

Operation:

```
GR[r2] ← GR[r1] + GR[r2];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The ADDB pseudo-operation allows the coding of both true and false conditions and generates either a ADDBT or ADDBF instruction.

ADD AND BRANCH IF FALSE ADDBF

Format: ADDBF,cond,n r1,r2,target

2A	r2	r1	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To add two values and perform an IA relative branch conditionally based on the values added.

Description: GR *r1* and GR *r2* are added and the result is stored in GR *r2*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is not satisfied by the values added, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated add conditions shown in Table 5-4 on page 5-5 (never, =, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:

```
GR[r2] ← GR[r1] + GR[r2];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (!cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← cond_satisfied;
    else
        PSW[N] ← !cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The ADDB pseudo-operation allows the coding of both true and false conditions and generates either a ADDBT or ADDBF instruction.

ADD IMMEDIATE AND BRANCH IF TRUE — ADDIBT

Format: ADDIBT,cond,n i,r,target

29	r	im5	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To add two values and perform an IA relative branch conditionally based on the values added.

Description: The sign-extended immediate value *im5* is added to GR *r*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied by the values added, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated add conditions shown in Table 5-4 on page 5-5 (never,=, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values added satisfy the specified condition and set to 0 otherwise.

Operation:

```
GR[r] ← low_sign_ext(im5,5) + GR[r];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The ADDIB pseudo-operation allows the coding of both true and false conditions and generates either a ADDIBT or ADDIBF instruction.

ADD IMMEDIATE AND BRANCH IF FALSE — ADDIBF

Format: ADDIBF,cond,n i,r,target

2B	r	im5	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To add two values and perform an IA relative branch conditionally based on the values added.

Description: The sign-extended immediate value *im5* is added to GR *r*. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is not satisfied by the values added, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is any of the non-negated add conditions shown in Table 5-4 on page 5-5 (never, =, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:
```
GR[r] ← low_sign_ext(im5,5) + GR[r];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (!cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← cond_satisfied;
    else
        PSW[N] ← !cond_satisfied;
```

Exceptions: Taken branch trap

Notes: The ADDIB pseudo-operation allows the coding of both true and false conditions and generates either a ADDIBT or ADDIBF instruction.

BRANCH ON VARIABLE BIT — BVB

Format: BVB,cond,n r,target

30	0	r	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To test a bit at a variable position in a register and perform an IA relative branch if the condition is satisfied.

Description: If the bit in GR *r*, specified by the Shift Amount Register (CR 11), satisfies the condition, *cond*, the word displacement is assembled from the *w* and *w1* fields of the instruction, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is either "<" (bit is 1), or ">=" (bit is 0) from the extract/deposit conditions (Table 5-7 on page 5-7). Use of other conditions is an undefined operation. The boolean variable "cond_satisfied" in the operation section is set to 1 when the bit tested satisfies the specified condition and set to 0 otherwise.

Operation:

```
lshift(GR[r], CR[11]);
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

BRANCH ON BIT **BB**

Format: BB,cond,n r,p,target

31	p	r	c	w1	n	w
6	5	5	3	11	1	1

Purpose: To test a bit at a fixed position in a register and perform an IA relative branch if the test condition is satisfied.

Description: If the bit in GR *r* specified by *p* satisfies the condition, *cond*, the word displacement is assembled from the *w* and *w1* fields of the instruction, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ",n" completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: The condition, *cond*, is either "<" (bit is 1), or ">=" (bit is 0) from the extract/deposit conditions (Table 5-7 on page 5-7). Use of other conditions is an undefined operation. The boolean variable "cond_satisfied" in the operation section is set to 1 when the bit tested satisfies the specified condition and set to 0 otherwise.

Operation:

```
lshift(GR[r],p);
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
    IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Computation Instructions

Computation instructions are comprised of the arithmetic, logical, shift, extract, and deposit instructions. The two 5-bit fields following the 6-bit opcode field could consist of the following combinations:

1. Two source registers.
2. A source register and a target register.
3. A source register and a 5-bit immediate.
4. A target register and a 5-bit immediate.

The three register arithmetic and logical instructions take two source arguments from two general registers. These source registers are specified by the two 5-bit fields following the opcode specifier. The rightmost 5-bit field specifies the target register.

Some of the computation instructions have a signed immediate argument which is either five bits or eleven bits in length. The 5-bit immediate is encoded in the second 5-bit field following the opcode field and the target specifier in the first 5-bit field following the opcode field. The 11-bit immediate is encoded in the rightmost 11-bit field, and the target specifier in the second 5-bit field following the opcode specifier.

Any computation instruction may nullify the instruction following, given the correct conditions. Most computation instructions encode the condition completers in the 3-bit *c*-field and 1-bit *f*-field of the instructions. The exceptions are the instructions that use the extract/deposit conditions. The computation instructions that use the extract/deposit conditions encode the condition completers in only the 3-bit *c*-field. The condition completers are used to determine if the instruction following is nullified, based on the contents of the source operands and the operation performed.

Three-Register Arithmetic and Logical Instructions

These instructions perform arithmetic and logical operations between two operands in registers and store the result into a register. Each arithmetic/logical instruction also specifies the conditional occurrence of either a skip or a trap, based on its opcode and the condition field. Not all options are available on every instruction. Only those operations and options considered useful are defined.

Immediate Arithmetic Operations

The immediate arithmetic instructions operate between a sign-extended 11-bit immediate and the contents of a register. The result is stored in a register. Immediate operations may optionally trap on overflow. In addition, immediate adds may trap on a specific condition.

The 11-bit immediate field has the sign bit in the rightmost position, but the other 10 bits are in the usual order. The 1-bit opcode extension field determines whether overflow causes a trap.

Shift Double, Extract, and Deposit Instructions

The double shift operations allow for a concatenation of two registers followed by a shift of 0 to 31 bit positions. The rightmost 32 bits are stored in a general register. Depending on the choice of the source registers, this operation allows the user to perform right or left shifts, rotates, bit field extractions when the bit field crosses word boundaries, unaligned byte moves, and so on.

Extract instructions take a field from a source register and insert it right-justified into the target register. This field is either zero extended or sign extended. This way, the extract instructions support both logical and arithmetic shift operations.

Deposits either set the target to zero or leave it unchanged (merge operation). The deposit instructions then take a right-justified field from a source and deposit it into any portion of the target. The source can be either a register or a 5-bit signed immediate value. The 5-bit immediate field has the sign bit in the rightmost position, but the other 4 bits are in the usual order. Deposit instructions support left shift operations and simple multiplication by powers of two.

Format: ADD,cond r1,r2,t

02	r2	r1	c	f	18	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer addition and conditionally nullify the following instruction.

Description: GR *r1* and GR *r2* are added and the result is placed in GR *t*. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:
GR[t] $\leftarrow$ GR[r1] + GR[r2];
PSW[C/B] $\leftarrow$ carry_borrows;
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

ADD LOGICAL — ADDL

Format: ADDL,cond r1,r2,t

02	r2	r1	c	f	28	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer addition without affecting the PSW C/B-bits and conditionally nullify the following instruction.

Description: GR *r1* and GR *r2* are added and the result is placed in GR *t*. The carry/borrow bits in the PSW are not updated.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] $\leftarrow$ GR[r1] + GR[r2];
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

Format: ADDO,cond r1,r2,t

02	r2	r1	c	f	38	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer addition, conditionally nullify the next instruction, and trap on overflow.

Description: GR *r1* and GR *r2* are added. If signed overflow does not occur, the results are placed in GR *t* and the carry/borrow bits in the PSW are updated; if signed overflow occurs, an overflow trap is taken. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the c and f fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← GR[r1] + GR[r2];
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

ADD WITH CARRY ADDC

Format: ADDC,cond r1,r2,t

02	r2	r1	c	f	1C	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer addition with carry and conditionally nullify the following instruction.

Description: GR *r1* and GR *r2* are added with the leftmost carry/borrow bit from the PSW and the result is placed in GR *t*. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] ← GR[r1] + GR[r2] + PSW[C/B]{0};
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

ADD WITH CARRY AND TRAP ON OVERFLOW — ADDCO

Format: ADDCO,cond r1,r2,t

02	r2	r1	c	f	3C	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer addition with carry, conditionally nullify the following instruction, and trap on overflow.

Description: GR *r1* and GR *r2* are added with the leftmost carry/borrow bit from the PSW. If signed overflow does not occur, the result is placed in GR *t*; if signed overflow occurs, an overflow trap is taken instead. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← GR[r1] + GR[r2] + PSW[C/B]{0};
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

SHIFT ONE AND ADD SH1ADD

Format: SH1ADD,cond r1,r2,t

02	r2	r1	c	f	19	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication.

Description: GR *r1* is shifted left one bit position and added to GR *r2*. The result is placed in GR *t* and the carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that either the bit shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that the bit shifted out is 1 or an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

GR[t] ← lshift(GR[r1], 1) + GR[r2];
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

SHIFT ONE AND ADD LOGICAL — SH1ADDL

Format: SH1ADDL,cond r1,r2,t

02	r2	r1	c	f	29	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication without affecting the carry/borrow bits.

Description: GR *r1* is shifted left one bit position and added to GR *r2*. The result is placed in GR *t*. The carry/borrow bits in the PSW are not affected. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that either the bit shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that the bit shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] $\leftarrow$ lshift(GR[r1], 1) + GR[r2];
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

SHIFT ONE, ADD AND TRAP ON OVERFLOW — SH1ADDO

Format: SH1ADDO,cond r1,r2,t

02	r2	r1	c	f	39	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication and trap on overflow.

Description: GR *r1* is shifted left one bit position and added to GR *r2*. If signed overflow does not occur, the result is placed in GR *t*; if signed overflow occurs, an overflow trap is taken instead. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

For this instruction, signed overflow condition means that either the bit shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that the bit shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← lshift(GR[r1], 1) + GR[r2];
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

SHIFT TWO AND ADD — SH2ADD

Format: SH2ADD,cond r1,r2,t

02	r2	r1	c	f	1A	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication.

Description: GR *r1* is shifted left two bit positions and added to GR *r2*. The result is placed in GR *t* and the carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that any of the bits shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that at least one of the bits shifted out is 1 or an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] ← lshift(GR[r1],2) + GR[r2];
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

SHIFT TWO AND ADD LOGICAL — SH2ADDL

Format: SH2ADDL,cond r1,r2,t

02	r2	r1	c	f	2A	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation multiplication without affecting the carry/borrow bits.

Description: GR *r1* is shifted left two bit positions and added to GR *r2*. The result is placed in GR *t*. The carry/borrow bits in the PSW are not updated. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that any of the bits shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that at least one of the bits shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] $\leftarrow$ lshift(GR[r1],2) + GR[r2];
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

SHIFT TWO, ADD AND TRAP ON OVERFLOW — SH2ADDO

Format: SH2ADDO,cond r1,r2,t

02	r2	r1	c	f	3A	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation multiplication and trap on overflow.

Description: GR *r1* is shifted left two bit positions and added to GR *r2*. If signed overflow does not occur, the result is placed in GR *t*; if signed overflow occurs, an overflow trap is taken instead. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

For this instruction, signed overflow condition means that any of the bits shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that at least one of the bits shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← lshift(GR[r1],2) + GR[r2];
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

SHIFT THREE AND ADD SH3ADD

Format: SH3ADD,cond r1,r2,t

02	r2	r1	c	f	1B	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication.

Description: GR *r1* is shifted left 3 bit positions and added to GR *r2*. The result is placed in GR *t* and the carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that any of the bits shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that at least one of the bits shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] $\leftarrow$ lshift(GR[r1],3) + GR[r2];
PSW[C/B] $\leftarrow$ carry_borrows;
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

SHIFT THREE AND ADD LOGICAL — SH3ADDL

Format: SH3ADDL,cond r1,r2,t

02	r2	r1	c	f	2B	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication.

Description: GR *r1* is shifted left 3 bit positions and added to GR *r2*. The result is placed in GR *t*. The carry/borrow bits in the PSW are not updated. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that any of the bits shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that at least one of the bits shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] ← lshift(GR[r1],3) + GR[r2];
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

SHIFT THREE, ADD AND TRAP ON OVERFLOW — SH3ADDO

Format: SH3ADDO,cond r1,r2,t

02	r2	r1	c	f	3B	0	t
6	5	5	3	1	6	1	5

Purpose: To provide a primitive operation for multiplication and trap on overflow.

Description: GR *r1* is shifted left 3 bit positions and added to GR *r2*. If signed overflow does not occur, the result is placed in GR *t*; if signed overflow occurs, an overflow trap is taken instead. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

For this instruction, signed overflow condition means that any of the bits shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that at least one of the bits shifted out is 1 or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← lshift(GR[r1],3) + GR[r2];
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

SUBTRACT SUB

Format: SUB,cond r1,r2,t

02	r2	r1	c	f	10	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer subtraction, and conditionally nullify the following instruction.

Description: GR *r2* is subtracted from GR *r1* and the result is placed in GR *t*. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit borrows resulting from the subtract operation.

The following instruction is nullified if the values subtracted satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions shown in Table 5-3 on page 5-5. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:
```
GR[t] ← GR[r1] + ~GR[r2] + 1;
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

SUBTRACT AND TRAP ON OVERFLOW — SUBO

Format: SUBO,cond r1,r2,t

02	r2	r1	c	f	30	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer subtraction, conditionally nullify the following instruction, and trap on overflow.

Description: GR *r2* is subtracted from GR *r1*. If signed overflow does not occur, the result is placed in GR *t*; if signed overflow occurs, an overflow trap is taken instead. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit borrows resulting from the subtract operation.

The following instruction is nullified if the values subtracted satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the compare or subtract conditions shown in Table 5-3 on page 5-5. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
res ← GR[r1] + ~GR[r2] + 1;
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

SUBTRACT WITH BORROW — SUBB

Format: SUBB,cond r1,r2,t

02	r2	r1	c	f	14	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer subtraction with borrow and conditionally nullify the following instruction.

Description: GR *r2* is subtracted from GR *r1* with the leftmost carry/borrow bit from the PSW and the result is placed in GR *t*. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit borrows resulting from the subtract operation.

The following instruction is nullified if the values subtracted satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions shown in Table 5-3 on page 5-5. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation: GR[t] ← GR[r1] + ~GR[r2] + PSW[C/B]{0};
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

SUBTRACT WITH BORROW AND TRAP ON OVERFLOW — SUBBO

Format: SUBBO,cond r1,r2,t

02	r2	r1	c	f	34	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer subtraction with borrow, conditionally nullify the following instruction, and trap on overflow.

Description: GR *r2* is subtracted from GR *r1* with the leftmost carry/borrow bit from the PSW. If signed overflow does not occur, the result is placed in GR *t*. If signed overflow occurs, an overflow trap is taken instead. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit borrows resulting from the subtract operation.

The following instruction is nullified if the values subtracted satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the compare or subtract conditions shown in Table 5-3 on page 5-5. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
res ← GR[r1] + ~GR[r2] + PSW[C/B]{0};
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

SUBTRACT AND TRAP ON CONDITION **SUBT**

Format: SUBT,cond r1,r2,t

02	r2	r1	c	f	13	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer subtraction and trap on a condition.

Description: GR *r2* is subtracted from GR *r1*. If the values subtracted do not satisfy the condition specified, the result is placed in GR *t*. If the values subtracted satisfy the condition specified, a conditional trap is taken. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit borrows resulting from the subtract operation. The condition, *cond*, is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions shown in Table 5-3 on page 5-5. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
res ← GR[r1] + ~GR[r2] + 1;
if (cond_satisfied)
    conditional_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
}
```

Exceptions: Conditional trap

SUBTRACT AND TRAP ON CONDITION OR OVERFLOW — SUBTO

Format: SUBTO,cond r1,r2,t

02	r2	r1	c	f	33	0	t
6	5	5	3	1	6	1	5

Purpose: To do 32-bit integer subtraction and trap on a condition or on overflow.

Description: GR *r2* is subtracted from GR *r1*. If signed overflow occurs, an overflow trap is taken; if signed overflow does not occur and the condition specified is satisfied, a conditional trap occurs. If neither trap occurs, the result is stored in GR *t*.

The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit borrows resulting from the subtract operation.

The condition, *cond*, is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the compare or subtract conditions (Table 5-3 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
res ← GR[r1] + ~GR[r2] + 1;
if (overflow)
    overflow_trap;
else if (cond_satisfied)
    conditional_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
}
```

Exceptions: Overflow trap
Conditional trap

DIVIDE STEP DS

Format: DS,cond r1,r2,t

02	r2	r1	c	f	11	0	t
6	5	5	3	1	6	1	5

Purpose: To provide the primitive operation for integer division.

Description: This instruction performs a single-bit non-restoring divide step and produces a set of result conditions. It calculates one bit of the quotient when GR *r1* is divided by GR *r2* and leaves the partial remainder in GR *t*. The quotient bit is the leftmost carry/borrow bit of the PSW. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the single-bit divide operation.

The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

For this instruction, signed overflow condition means that the bit shifted out differs from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition or subtraction. Unsigned overflow means that the bit shifted out is 1 or that an ordinary unsigned overflow occurred during the addition or subtraction. The conditions take on special interpretations since the shift operation participates in overflow determination.

Conditions: The condition is any of the compare or subtract conditions (Table 5-3 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
if (PSW[V])
    GR[t] ← cat(lshift(GR[r1],1),PSW[C/B]{0}) + ~GR[r2] + 1;
else
    GR[t] ← cat(lshift(GR[r1],1),PSW[C/B]{0}) + GR[r2];
PSW[C/B] ← carry_borrows;
PSW[V] ← xor(carry_borrows{0},GR[r2]{0});
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

COMPARE AND CLEAR — COMCLR

Format: COMCLR,cond r1,r2,t

02	r2	r1	c	f	22	0	t
6	5	5	3	1	6	1	5

Purpose: To compare two registers, set a register to 0, and conditionally nullify the following instruction, based on the result of the comparison.

Description: GR *r1* and GR *r2* are compared and GR *t* is set to zero. The carry/borrow bits in the PSW are not updated.

The following instruction is nullified if the values compared satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions (Table 5-3 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values compared satisfy the specified condition.

Operation:

```
GR[r1] + ~GR[r2] + 1;
GR[t] ← 0;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

PROGRAMMING NOTE

COMPARE AND CLEAR can be used to produce the logical value of the result of a comparison (assuming false is represented by 0 and true by 1) in a register. The following example will set *ra* to 1 if *rb* and *rc* are equal, and to 0 if they are not equal:

```
COMCLR,<>    rb,rc,ra
LDO          1(0),ra
```

INCLUSIVE OR

Format: OR,cond r1,r2,t

02	r2	r1	c	f	09	0	t
6	5	5	3	1	6	1	5

Purpose: To do a 32-bit, bitwise inclusive OR.

Description: GR *r1* and GR *r2* are ORed and the result is placed in GR *t*. The following instruction is nullified if the values ORed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the logical conditions (Table 5-5 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ORed satisfy the specified condition.

Operation: GR[t] $\leftarrow$ GR[r1] | GR[r2];
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

Notes: The COPY pseudo-operation allows for the movement of data from one register to another by generating the instruction OR r,0,t. The NOP pseudo-operation generates the instruction OR 0,0,0.

EXCLUSIVE OR XOR

Format: XOR,cond r1,r2,t

02	r2	r1	c	f	0A	0	t
6	5	5	3	1	6	1	5

Purpose: To do a 32-bit, bitwise exclusive OR.

Description: GR *r1* and GR *r2* are XORed and the result is placed in GR *t*. The following instruction is nullified if the values XORed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the logical conditions (Table 5-5 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values XORed satisfy the specified condition.

Operation: GR[t] $\leftarrow$ xor(GR[r1], GR[r2]);
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

Format: AND,cond r1,r2,t

02	r2	r1	c	f	08	0	t
6	5	5	3	1	6	1	5

Purpose: To do a 32-bit, bitwise AND.

Description: GR *r1* and GR *r2* are ANDed and the result is placed in GR *t*. The following instruction is nullified if the values ANDed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the logical conditions (Table 5-5 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ANDed satisfy the specified condition.

Operation: GR[t] $\leftarrow$ GR[r1] & GR[r2];
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

AND COMPLEMENT ANDCM

Format: ANDCM,cond r1,r2,t

02	r2	r1	c	f	00	0	t
6	5	5	3	1	6	1	5

Purpose: To do a 32-bit bitwise AND with complement.

Description: GR *r1* is ANDed with the one's complement of GR *r2* and the result is placed in GR *t*. The following instruction is nullified if the values ANDed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the logical conditions (Table 5-5 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ANDed satisfy the specified condition.

Operation: GR[t] $\leftarrow$ GR[r1] & ~GR[r2];
if (cond_satisfied) PSW[N] $\leftarrow$ 1;

Exceptions: None

UNIT XOR UXOR

Format: UXOR,cond   r1,r2,t

02	r2	r1	c	f	0E	0	t
6	5	5	3	1	6	1	5

Purpose: To individually compare corresponding sub-units of two words for equality.

Description: GR *r1* and GR *r2* are XORed and the result is placed in GR *t*. This instruction generates unit conditions unlike XOR which generates logical conditions. The following instruction is nullified if the values XORed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition, *cond*, is any of the unit conditions not involving carries shown in Table 5-6 on page 5-6 ("never", SBZ, SHZ, TR, NBZ, NHZ). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values XORed satisfy the specified condition.

Operation: GR[t] ← xor(GR[r1], GR[r2]);
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

UNIT ADD COMPLEMENT — UADDCM

Format: UADDCM,cond r1,r2,t

02	r2	r1	c	f	26	0	t
6	5	5	3	1	6	1	5

Purpose: To individually compare corresponding sub-units of a word for a greater-than or less-than-or-equal relation and to prepare for decimal operations.

Description: GR *r1* is added to the one's complemented of GR *r2* and the result is stored in GR *t*. The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition *cond* is any of the unit conditions (Table 5-6 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] ← GR[r1] + ~GR[r2];
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

PROGRAMMING NOTE

UNIT ADD COMPLEMENT can be used to perform a logical NOT operation when coded as follows:

```
UADDCM    0,r,t    /* GR[t] ← ~GR[r] */
```

UNIT ADD COMPLEMENT AND TRAP ON CONDITION — UADDCMT

Format: UADDCMT,cond r1,r2,t

02	r2	r1	c	f	27	0	t
6	5	5	3	1	6	1	5

Purpose: To individually compare corresponding sub-units of a word for a greater-than or less-than-or-equal a relation and trap if the specified condition is satisfied by any sub-unit.

Description: GR *r1* is added to the one's complement of GR *r2*. If the condition, *cond*, is satisfied by the values added a conditional trap is taken; otherwise the result is stored in GR *t*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition *cond* is any of the unit conditions (Table 5-6 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← GR[r1] + ~GR[r2];
if (cond_satisfied)
     conditional_trap;
else
     GR[t] ← res;
```

Exceptions: Conditional trap

PROGRAMMING NOTE

UNIT ADD COMPLEMENT AND TRAP ON CONDITION can be used to check decimal validity and to pre-bias decimal numbers. *ra* contains the number to be checked and *rt* will contain the number plus the bias as result of the UADDCMT operation.

```
NINES     .equ           X'99999999
          LDIL           l%NINES,nines
          LDO            r%NINES(nines),nines
          UADDCMT,SDC    ra,nines,rt
```

DECIMAL CORRECT DCOR

Format: DCOR,cond r,t

02	r	0	c	f	2E	0	t
6	5	5	3	1	6	1	5

Purpose: To separately correct the eight BCD digits of the result of an addition or subtraction.

Description: Every digit of GR *r* corresponding to a bit which is 0 in PSW C/B-bits has 6 subtracted from it. The result is stored in GR *t*. The following instruction is nullified if the result of the operation satisfies the specified condition *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition *cond* is any of the unit conditions (Table 5-6 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:
```
GR[t] ← GR[r] - cat(
      0x6*(1 - PSW[C/B]{0}), 0x6*(1 - PSW[C/B]{1}),
      0x6*(1 - PSW[C/B]{2}), 0x6*(1 - PSW[C/B]{3}),
      0x6*(1 - PSW[C/B]{4}), 0x6*(1 - PSW[C/B]{5}),
      0x6*(1 - PSW[C/B]{6}), 0x6*(1 - PSW[C/B]{7}));
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

PROGRAMMING NOTE

DECIMAL CORRECT can be used to take the sum of 32-bit BCD values. *ra*, *rb*, *rc*, and *rd* each contain a 32-bit BCD value and *rt* will hold the result at the end of the sequence. The UADDCM operation is used to pre-bias the value in *ra* in order to perform BCD arithmetic. The IDCOR operations between the ADD operations are used to re-adjust the BCD bias of the result. The final DCOR operation is used to remove the bias and leave the value in *rt* in BCD format.

```
NINES   .equ     0x99999999
        LDIL     l%NINES,nines
        LDO      r%NINES(nines),nines
        UADDCM   ra,nines,rt            ; pre-bias first operand
        ADD      rt,rb,rt               ; add in the next value
        IDCOR    rt,rt                  ; correct result, retaining bias
        ADD      rt,rc,rt               ; add in the next value
        IDCOR    rt,rt                  ; correct result, retaining bias
        ADD      rt,rd,rt               ; add in the next value
        DCOR     rt,rt                  ; final correction
```

INTERMEDIATE DECIMAL CORRECT IDCOR

Format: IDCOR,cond r,t

02	r	0	c	f	2F	0	t
6	5	5	3	1	6	1	5

Purpose: To separately correct the eight BCD digits of the result of an addition or subtraction.

Description: Every digit of GR *r* corresponding to a bit which is on in PSW C/B-bits has 6 added to it. The result is stored in GR *t*. The following instruction is nullified if the result of the operation satisfies the specified condition *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition *cond* is any of the unit conditions (Table 5-6 on page 5-6). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
GR[t] ← GR[r] + cat(
    0x6*PSW[C/B]{0}, 0x6*PSW[C/B]{1},
    0x6*PSW[C/B]{2}, 0x6*PSW[C/B]{3},
    0x6*PSW[C/B]{4}, 0x6*PSW[C/B]{5},
    0x6*PSW[C/B]{6}, 0x6*PSW[C/B]{7});
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

ADD TO IMMEDIATE   ADDI

Format: ADDI,cond   i,r,t

2D	r	t	c	f	0	im11
6	5	5	3	1	1	11

Purpose: To add an immediate value to a register and conditionally nullify the following instruction.

Description: The sign-extended immediate value *i* is added to GR *r* and the result is stored in GR *t*. The immediate value is encoded into the *im11* field. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation: GR[t] ← low_sign_ext(im11,11) + GR[r];
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

ADD TO IMMEDIATE AND TRAP ON OVERFLOW — ADDIO

Format: ADDIO,cond i,r,t

2D	r	t	c	f	1	im11
6	5	5	3	1	1	11

Purpose: To add an immediate value to a register, conditionally nullify the following instruction, and trap on overflow.

Description: The sign-extended immediate value *i* and GR *r* are added. The immediate value is encoded into the *im11* field. If signed overflow does not occur, the result is stored in GR *t*, and the carry/borrow bits in the PSW are updated. If signed overflow occurs, an overflow trap is taken instead. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation.

The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← low_sign_ext(im11,11) + GR[r];
if (overflow)
     overflow_trap;
else {
     GR[t] ← res;
     PSW[C/B] ← carry_borrows;
     if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

ADD TO IMMEDIATE AND TRAP ON CONDITION — ADDIT

Format: ADDIT,cond i,r,t

2C	r	t	c	f	0	im11
6	5	5	3	1	1	11

Purpose: To add an immediate value to a register and trap on a condition.

Description: The sign-extended immediate value *i* and GR *r* are added. The immediate value is encoded into the *im11* field. If the specified condition, *cond*, is satisfied by the values added, a conditional trap occurs; otherwise, the result is stored in GR *t* and the carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← low_sign_ext(im11,11) + GR[r];
if (cond_satisfied)
    conditional_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
}
```

Exceptions: Conditional trap

ADD TO IMMEDIATE AND TRAP ON CONDITION OR OVERFLOW — ADDITO

Format: ADDITO,cond i,r,t

2C	r	t	c	f	1	im11
6	5	5	3	1	1	11

Purpose: To add an immediate value to a register and trap on a condition or on overflow.

Description: The sign-extended immediate value *i* and GR *r* are added. The immediate value is encoded into the *im11* field. If signed overflow occurs, an overflow trap is taken. If signed overflow does not occur and the specified condition, *cond*, is satisfied, a conditional trap occurs. If overflow does not occur and the specified condition is not satisfied, the result is stored in GR *t*, and the carry/borrow bits in the PSW are updated.

The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The boolean variable "overflow" in the operation section is set if the operation results in a signed overflow.

Conditions: The condition is any of the add conditions (Table 5-4 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← low_sign_ext(im11,11) + GR[r];
if (overflow)
    overflow_trap;
else if (cond_satisfied)
    conditional_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
}
```

Exceptions: Overflow trap
Conditional trap

Format: SUBI,cond i,r,t

25	r	t	c	f	0	im11
6	5	5	3	1	1	11

Purpose: To subtract a register from an immediate value and conditionally nullify the following instruction.

Description: GR *r* is subtracted from the sign-extended immediate value *i* and the result is stored in GR *t*. The immediate value is encoded into the *im11* field. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the subtract operation. The following instruction is nullified if the values subtracted satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions (Table 5-3 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
GR[t] ← low_sign_ext(im11,11) + ~GR[r] + 1;
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

PROGRAMMING NOTE

SUBTRACT FROM IMMEDIATE can be used to perform a logical NOT operation when coded as follows:

```
SUBI    -1,r,t        /* GR[t] ← ~GR[r]
                         all PSW[C/B] are set to ones */
```

SUBTRACT FROM IMMEDIATE AND TRAP ON OVERFLOW — SUBIO

Format: SUBIO,cond i,r,t

25	r	t	c	f	1	im11
6	5	5	3	1	1	11

Purpose: To subtract a register from an immediate value, conditionally nullify the following instruction, and trap on overflow.

Description: GR *r* is subtracted from the sign-extended immediate value *i*. The immediate value is encoded into the *im11* field. If signed overflow does not occur, the result is stored in GR *t*, and the carry/borrow bits in the PSW are updated. If signed overflow occurs, an overflow trap is taken instead. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the subtract operation. The following instruction is nullified if the values subtracted satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions (Table 5-3 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
res ← low_sign_ext(im11,11) + ~GR[r] + 1;
if (overflow)
    overflow_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}
```

Exceptions: Overflow trap

COMPARE IMMEDIATE AND CLEAR — COMICLR

Format: COMICLR,cond   i,r,t

24	r	t	c	f	0	im11
6	5	5	3	1	1	11

Purpose: To compare an immediate value with the contents of a register, set a register to 0, and conditionally nullify the following instruction.

Description: The sign-extended immediate and GR *r* are compared and GR *t* is set to zero. The immediate value is encoded into the *im11* field. The following instruction is nullified if the values compared satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the compare or subtract conditions (Table 5-3 on page 5-5). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values compared satisfy the specified condition.

Operation:

```
low_sign_ext(im11,11) + ~GR[r] + 1;
GR[t] ← 0;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

VARIABLE SHIFT DOUBLE VSHD

Format: VSHD,cond r1,r2,t

34	r2	r1	c	0	0	t
6	5	5	3	3	5	5

Purpose: To shift a pair of registers by a variable amount and conditionally nullify the following instructions.

Description: The rightmost 31 bits of GR *r1* are concatenated with the 32 bits of GR *r2* and shifted right the number of bits given by the Shift Amount Register (CR 11). The rightmost 32 bits of the result are stored in GR *t*. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation: GR[t] ← rshift(cat(GR[r1]{1..31},GR[r2]),CR[11]){32..63};
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

PROGRAMMING NOTE

A logical right shift of GR *r* by a variable amount contained in GR *p* leaving the result in GR *t* may be done by the following sequence:

```
MTSAR   p
VSHD    0,r,t
```

An arithmetic right shift can be done with an extract instruction. See VARIABLE EXTRACT SIGNED on page 5-125 for an example.

If *r1* and *r2* name the same register, its contents are rotated and placed in GR *t*. See SHIFT DOUBLE on page 5-123 for an example.

Format: SHD,cond r1,r2,p,t

34	r2	r1	c	2	cp	t
6	5	5	3	3	5	5

Purpose: To shift a pair of registers by a fixed amount and conditionally nullify the following instruction.

Description: The rightmost 31 bits of GR *r1* are concatenated with the 32 bits of GR *r2* and shifted right *p* bits. The rightmost 32 bits of the result are stored in GR *t*. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The shift count *p* in the assembly language format is represented by *cp* in the machine instruction, whose value is 31-*p*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the condition.

Operation: GR[t] ← rshift(cat(GR[r1]{1..31},GR[r2]),p){32..63};
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

PROGRAMMING NOTE

A rotate operation is possible if the two source registers are the same. For example, the following rotates the contents of *ra* right by 8 bits:

```
SHD     ra,ra,8,ra
```

VARIABLE EXTRACT UNSIGNED — VEXTRU

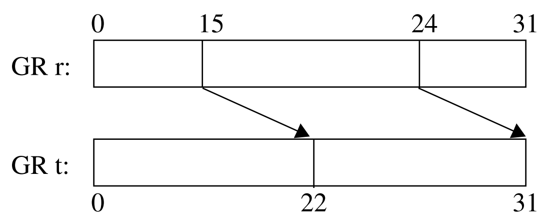
Format: VEXTRU,cond r,len,t

34	r	t	c	4	0	clen
6	5	5	3	3	5	5

Purpose: To extract any 32-bit or shorter field from a variable position, and conditionally nullify the following instruction.

Description: A field is extracted, from GR *r*, zero extended and placed right-justified in GR *t*. This field is of length *len*. It begins at the bit position given by the Shift Amount Register (CR 11) and extends to the left. If the field extends beyond the leftmost bit, it is zero extended. The following diagram illustrates an extract of a 10-bit field when the Shift Amount Register contains the value 24.

The instruction is: VEXTRU r,10,t.



The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
shct ← 1 + CR[11];
tmp ← lshift(zero_ext_64(GR[r],32),shct){0..31};
GR[t] ← zero_ext(tmp{32-len..31},len);
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

VARIABLE EXTRACT SIGNED — VEXTRS

Format: VEXTRS,cond r,len,t

34	r	t	c	5	0	clen
6	5	5	3	3	5	5

Purpose: To extract any signed 32-bit or shorter field from a variable position, and conditionally nullify the following instruction.

Description: A field is extracted, from GR *r*, sign extended and placed right-justified in GR *t*. This field is of length *len*. It begins at the bit position given by the Shift Amount Register (CR 11) and extends to the left. If the field extends beyond the leftmost bit, it is sign extended. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
shct ← 1 + CR[11];
tmp ← lshift(sign_ext_64(GR[r],32), shct){0..31};
GR[t] ← sign_ext(tmp{32-len..31},len);
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

PROGRAMMING NOTE

An arithmetic right shift of GR *r* by a variable amount contained in GR *p* leaving the result in GR *t* may be done by the following sequence:

```
SUBI    31,p,t
MTSAR   t
VEXTRS  r,32,t
```

EXTRACT UNSIGNED EXTRU

Format: EXTRU,cond r,p,len,t

34	r	t	c	6	p	clen
6	5	5	3	3	5	5

Purpose: To extract any 32-bit or shorter field, and conditionally nullify the following instruction.

Description: A field is extracted, from GR *r*, zero extended and placed right-justified in GR *t*. This field is of length *len*. It begins at bit position *p* and extends to the left. Since the field is fully specified by the instruction, it cannot extend beyond the leftmost bit. Doing so is an undefined operation. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
if (p >= len-1) {
    tmp ← lshift(zero_ext_64(GR[r],32),(1+p)){0..31};
    GR[t] ← zero_ext(tmp{32-len..31},len);
    if (cond_satisfied) PSW[N] ← 1;
} else
    undefined;
```

Exceptions: None

Format: EXTRS,cond r,p,len,t

34	r	t	c	7	p	clen
6	5	5	3	3	5	5

Purpose: To extract any signed 32-bit or shorter field, and conditionally nullify the following instruction.

Description: A field is extracted, from GR *r*, sign extended, and placed right-justified in GR *t*. This field is of length *len*. It begins at bit position *p* and extends to the left. Since the field is fully specified by the instruction, it cannot extend beyond the leftmost bit. Doing so is an undefined operation. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
if (p >= len-1) {
    tmp ← lshift(sign_ext_64(GR[r],32),(1+p)){0..31};
    GR[t] ← sign_ext(tmp{32-len..31},len);
    if (cond_satisfied) PSW[N] ← 1;
} else
    undefined;
```

Exceptions: None

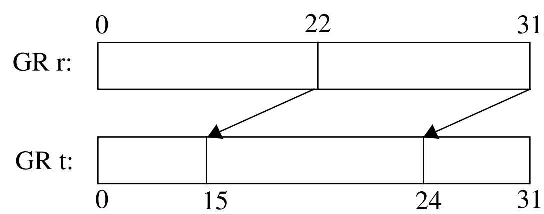
Format: VDEP,cond r,len,t

35	t	r	c	1	0	clen
6	5	5	3	3	5	5

Purpose: To deposit a value into a register at a variable position, and conditionally nullify the following instruction.

Description: A right-justified field from GR *r* is deposited (merged) in GR *t*. This field is of length *len*. It begins at the bit position given by the Shift Amount Register (CR 11) and extends to the left. If the field extends beyond the leftmost bit, the field is truncated and the higher bits are ignored. The remainder of GR *t* is unchanged. The following diagram illustrates a deposit of a 10-bit field when the Shift Amount Register contains the value 24.

The instruction is: VDEP r,10,t.



The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction. The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
tpos ← CR[11];
if (tpos < len-1)
    GR[t]{0..tpos} ← GR[r]{31-tpos..31};
else
    GR[t]{tpos-len+1..tpos} ← GR[r]{32-len..31};
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

DEPOSIT DEP

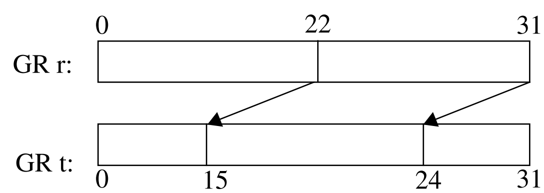
Format: DEP,cond r,p,len,t

35	t	r	c	3	cp	clen
6	5	5	3	3	5	5

Purpose: To deposit a value into a register at a constant position, and conditionally nullify the following instruction.

Description: A right-justified field from GR *r* is deposited (merged) in GR *t*. This field is of length *len*. It begins at the bit position *p* and extends to the left. Since the field is fully specified by the instruction, it cannot extend beyond the leftmost bit. Doing so is an undefined operation. The remainder of GR *t* is unchanged. The following diagram illustrates a deposit of a 10-bit field when *p* specifies the value 24.

The instruction is: DEP r,24,10,t.



The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction. The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*. The bit position *p* in the assembly language format is represented by *cp* in the machine instruction, whose value is 31-*p*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
if (p >= len-1) {
     GR[t]{p-len+1..p} ← GR[r]{32-len..31};
     if (cond_satisfied) PSW[N] ← 1;
} else
     undefined;
```

Exceptions: None

VARIABLE DEPOSIT IMMEDIATE VDEPI

Format: VDEPI,cond i,len,t

35	t	im5	c	5	0	clen
6	5	5	3	3	5	5

Purpose: To deposit an immediate value into a register at a variable position, and conditionally nullify the following instruction.

Description: A right-justified field from the sign-extended immediate *i* is deposited (merged) in GR *t*. This field is of length *len*. It begins at the bit position given by the Shift Amount Register (CR 11), and extends to the left. If the field extends beyond the leftmost bit, the field is truncated and the higher bits are ignored. The remainder of GR *t* is unchanged. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction. The immediate is encoded in the *im5* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
tpos ← CR[11];
ival ← low_sign_ext(im5,5);
if (tpos < len-1)
    GR[t] ← cat(ival{31-tpos..31},GR[t]{tpos+1..31});
else
    GR[t] ← cat(GR[t]{0..tpos-len},ival{32-len..31},GR[t]{tpos+1..31});
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

DEPOSIT IMMEDIATE DEPI

Format: DEPI,cond i,p,len,t

35	t	im5	c	7	cp	clen
6	5	5	3	3	5	5

Purpose: To deposit an immediate value into a register at a constant position, and conditionally nullify the following instruction.

Description: A right-justified field from the sign-extended immediate *i* is deposited (merged) in GR *t*. This field is of length *len*. It begins at the bit position p and extends to the left. Since the field is fully specified by the instruction, it cannot extend beyond the leftmost bit. Doing so is an undefined operation. The remainder of GR *t* is unchanged. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction. The immediate is encoded in the *im5* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*. The bit position *p* in the assembly language format is represented by *cp* in the machine instruction, whose value is 31-*p*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
ival ← low_sign_ext(im5,5);
if (p >= len-1) {
    GR[t] ← cat(GR[t]{0..p-len},ival{32-len..31},GR[t]{p+1..31});
    if (cond_satisfied) PSW[N] ← 1;
} else
    undefined;
```

Exceptions: None

ZERO AND VARIABLE DEPOSIT — ZVDEP

Format: ZVDEP,cond r,len,t

35	t	r	c	0	0	clen
6	5	5	3	3	5	5

Purpose: To set a register to zero, deposit a value into it at a variable position, and conditionally nullify the following instruction.

Description: GR *t* is set to zero and a right-justified field from GR *r* is deposited in it. This field is of length *len*. It begins at the bit position given by the Shift Amount Register (CR 11), and extends to the left. If the field extends beyond the leftmost bit, the field is truncated and the higher bits are ignored. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
tpos ← CR[11];
if (tpos < len-1)
    GR[t] ← cat(GR[r]{31-tpos..31},0{tpos+1..31});
else
    GR[t] ← cat(0{0..tpos-len},GR[r]{32-len..31},0{tpos+1..31});
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

PROGRAMMING NOTE

This sequence left shifts the value in *ra* by a variable amount given by *count* leaving the result in *rt*. Note that this provides no indication of a possible overflow.

```
SUBI    31,count,rt    ; adjust the shift count
MTCTL   rt,%sar        ; move shift amount to SAR
ZVDEP   ra,32,rt       ; deposit all 32 bits of ra
```

ZERO AND DEPOSIT — ZDEP

Format: ZDEP,cond r,p,len,t

35	t	r	c	2	cp	clen
6	5	5	3	3	5	5

Purpose: To set a register to zero and deposit a value into it at a constant position.

Description: GR *t* is set to zero and a right-justified field from GR *r* is deposited in it. This field is of length *len*. It begins at the bit position *p* and extends to the left. Since the field is fully specified by the instruction, it cannot extend beyond the leftmost bit. Doing so is an undefined operation. The following instruction is nullified if the result of the operation satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*. The bit position *p* in the assembly language format is represented by *cp* in the machine instruction, whose value is 31-*p*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
if (p >= len-1) {
    GR[t] ← cat(0{0..p-len},GR[r]{32-len..31},0{p+1..31});
    if (cond_satisfied) PSW[N] ← 1;
} else
    undefined;
```

Exceptions: None

ZERO AND VARIABLE DEPOSIT IMMEDIATE — ZVDEPI

Format: ZVDEPI,cond i,len,t

35	t	im5	c	4	0	clen
6	5	5	3	3	5	5

Purpose: To set a register to zero and deposit an immediate value into it at a variable position.

Description: GR *t* is set to zero and a right-justified field from the sign-extended immediate *i* is deposited in it. This field is of length *len*. It begins at the bit position given by the Shift Amount Register (CR 11), and extends to the left. If the field extends beyond the leftmost bit, the field is truncated and the higher bits are ignored. The following instruction is nullified if the result of the condition satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction. The immediate is encoded in the *im5* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
tpos ← CR[11];
ival ← low_sign_ext(im5,5);
if (tpos < len-1)
    GR[t] ← cat(ival{31-tpos..31},0{tpos+1..31});
else
    GR[t] ← cat(0{0..tpos-len},ival{32-len..31},0{tpos+1..31});
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

ZERO AND DEPOSIT IMMEDIATE ZDEPI

Format: ZDEPI,cond i,p,len,t

35	t	im5	c	6	cp	clen
6	5	5	3	3	5	5

Purpose: To set a register to zero and deposit an immediate value into it at a constant position.

Description: GR *t* is set to zero and a right-justified field from the sign-extended immediate *i* is deposited in it. This field is of length *len*. It begins at the bit position *p* and extends to the left. Since the field is fully specified by the instruction, it cannot extend beyond the leftmost bit. Doing so is an undefined operation. The following instruction is nullified if the result of the condition satisfies the specified condition, *cond*. The condition is encoded in the *c* field of the instruction. The immediate is encoded in the *im5* field of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*. The bit position *p* in the assembly language format is represented by *cp* in the machine instruction, whose value is 31-*p*.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7 on page 5-7). When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

```
ival ← low_sign_ext(im5,5);
if (p >= len-1) {
    GR[t] ← cat(0{0..p-len},ival{32-len..31},0{p+1..31});
    if (cond_satisfied) PSW[N] ← 1;
} else
    undefined;
```

Exceptions: None

System Control Instructions

The system control instructions provide special register moves, system mask control, return from interruption, probe access rights, memory management operations, and implementation-dependent functions.

Memory management instructions generate instruction and data addresses. Address formation is similar to that of the indexed load instructions. The only difference is that the index register is never shifted before adding to the base register. The address formation, the completers, and the bit field encodings are shown in Table 5-14.

Table 5-14. System Control Completers

cmplt	Description	m
<none>	don't modify base register	0
M	Modify base register	1

In the above table, *cmplt* is in assembly language format and *m* is in machine language format. The effective address computation for the insert TLB address and insert TLB protection instructions concatenates a space register with a general register.

The probe instructions use the two rightmost bits of the index value or immediate to indicate the privilege level for which access is to be validated. The probe instructions do not perform address modification.

Memory management instructions behave either as data reference instructions or instruction reference instructions in terms of space identifier selection as shown in Figure 5-10. The calculation of the offset portion of the address is shown in Figure 5-11.

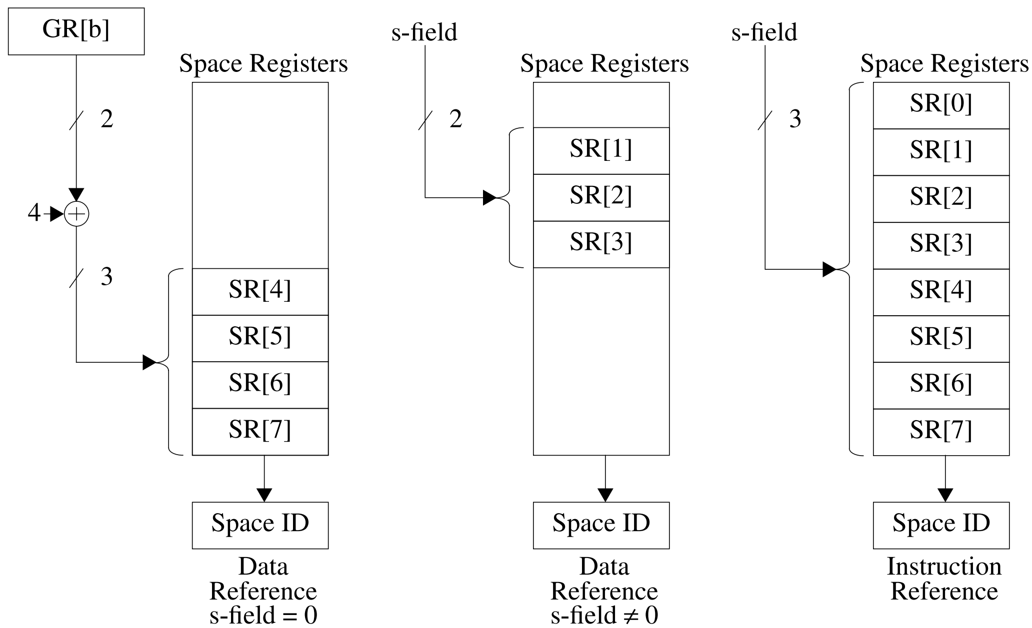


Figure 5-10. Space Identifier Selection

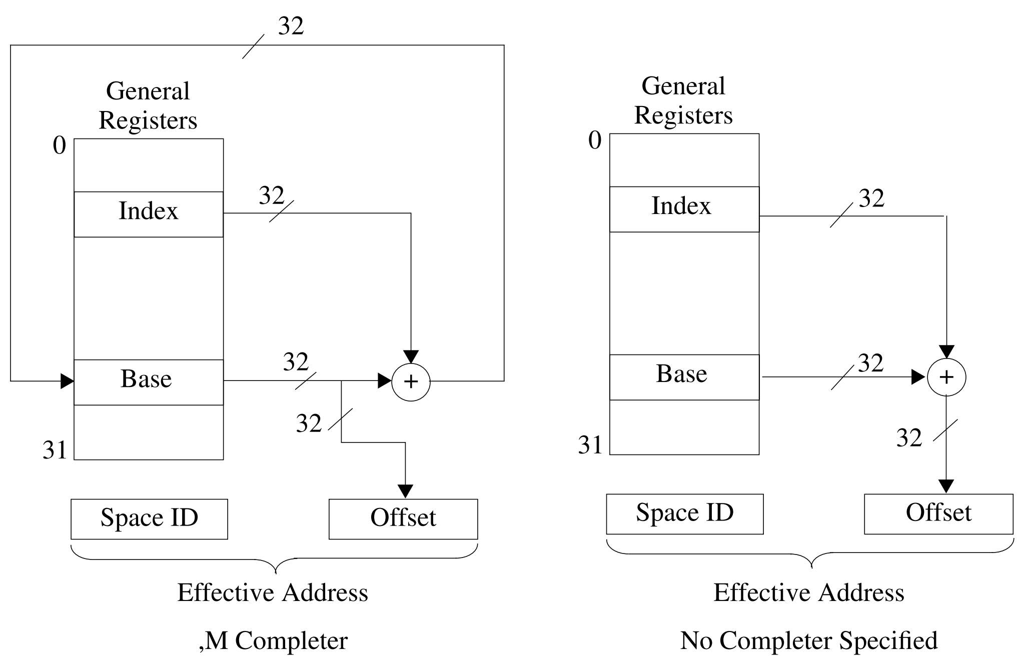


Figure 5-11. System Operations

Format: BREAK im5,im13

00	im13	00	im5
6	13	8	5

Purpose: To cause a break instruction trap for debugging purposes.

Description: A break instruction trap occurs when this instruction is executed.

Operation: break_instruction_trap;

Exceptions: None

Notes: *im5* and *im13* can be used as parameters to the "BREAK" processing code.

RETURN FROM INTERRUPTION RFI

Format: RFI

00	rv	rv	rv	60	0
6	5	5	3	8	5

Purpose: To restore processor state and restart execution of an interrupted instruction stream.

Description: The PSW register contents are restored from the IPSW register but are not modified by this instruction. The IA queues are restored from the IIA queues. Execution continues at the locations loaded into the IA queues.

Execution of an RFI with the IPSW Q-bit equal to 0 returns to the location specified by the IIA queues, but leaves the IIAOQ, IIASQ, and IPRs undefined. Software is responsible for avoiding interruptions during the execution of an RFI. Execution of an RFI instruction when any of the PSW Q, I, or R bits are ones is an undefined operation. Execution of an RFI instruction when the PSW L bit is a one is an undefined operation if the new privilege level after execution of the RFI is non-zero. This instruction and the RFIR instruction are the only instructions that can set the PSW Q-bit to 1.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    PSW          ←  IPSW;
    IAOQ_Back    ←  IIAOQ_Back;                    /* CR[18] */
    IAOQ_Front   ←  IIAOQ_Front;                   /* CR[18] */
    if (!level_0) {
        IASQ_Back  ←  IIASQ_Back;                  /* CR[17] */
        IASQ_Front ←  IIASQ_Front;                 /* CR[17] */
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Because this instruction restores the state of the execution pipeline, it is possible for software to place the processor in states which could not result from the execution of any sequence of instructions not involving interruptions. For example, it could set the PSW B-bit to 0 even though the addresses in the IA queues are not contiguous. The operation of the machine is undefined in such cases, and it is the responsibility of software to avoid them.

Some machines promote privilege at the target of a GATEWAY instruction. To avoid improper processor states, software must not set the PSW B-bit to 0 with different privilege levels in the IAOQ unless the executing processor promotes privilege in the delay slot of a GATEWAY.

Notes: When this instruction returns to an instruction which executes at a lower privilege level, a lower-privilege transfer trap is not taken.

Level 0: This instruction executes as usual except that the IASQ is nonexistent and updating it has no effect.

RETURN FROM INTERRUPTION AND RESTORE RFIR

Format: RFIR

00	rv	rv	rv	65	0
6	5	5	3	8	5

Purpose: To restore processor state, restore GRs 1, 8, 9, 16, 17, 24, and 25 from the shadow registers, and restart execution of an interrupted instruction stream.

Description: The contents of GRs 1, 8, 9, 16, 17, 24, and 25 are restored from their shadow registers. The PSW register contents are restored from the IPSW register but are not modified by this instruction. The IA queues are restored from the IIA queues. Execution continues at the locations loaded into the IA queues.

Execution of an RFIR with the IPSW Q-bit equal to 0 returns to the location specified by the IIA queues, but leaves the IIAOQ, IIASQ and IPRs undefined. Software is responsible for avoiding interruptions during the execution of an RFIR. Execution of an RFIR instruction when any of the PSW Q, I, or R bits are ones is an undefined operation. Execution of an RFIR instruction when the PSW L bit is a one is an undefined operation if the new privilege level after execution of the RFIR is non-zero. This instruction and the RFI instruction are the only instructions that can set the PSW Q-bit to 1.

Execution of an RFIR instruction when the contents of the shadow registers are undefined leaves the contents of GRs 1, 8, 9, 16, 17, 24, and 25 undefined. After execution of an RFIR instruction, the SHRs are undefined.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    GR[1]        ←  SHR[0];
    GR[8]        ←  SHR[1];
    GR[9]        ←  SHR[2];
    GR[16]       ←  SHR[3];
    GR[17]       ←  SHR[4];
    GR[24]       ←  SHR[5];
    GR[25]       ←  SHR[6];
    PSW          ←  IPSW;
    IAOQ_Back    ←  IIAOQ_Back;                /* CR[18] */
    IAOQ_Front   ←  IIAOQ_Front;               /* CR[18] */
    if (!level_0) {
        IASQ_Front ←  IIASQ_Front;             /* CR[17] */
        IASQ_Back  ←  IIASQ_Back;              /* CR[17] */
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Because this instruction restores the state of the execution pipeline, it is possible for software to place the processor in states which could not result from the execution of any sequence of instructions not involving interruptions. For example, it could set the PSW B-bit to 0 even though the addresses in the IA queues are not contiguous. The operation of the machine is undefined in such cases, and it is the responsibility of software to avoid them.

Some machines promote privilege at the target of a GATEWAY instruction. To avoid improper processor states, software must not set the PSW B-bit to 0 with different privilege levels in the IAOQ unless the executing processor promotes privilege in the delay slot of a GATEWAY.

Notes: When this instruction returns to an instruction which executes at a lower privilege level, a lower-privilege transfer trap is not taken.

Level 0: This instruction executes as usual except that the IASQ is nonexistent and updating it has no effect.

SET SYSTEM MASK SSM

Format: SSM i,t

00	rv	i	0	6B	t
6	3	7	3	8	5

Purpose: To selectively set bits in the system mask to 1.

Description: The current value of the system mask, PSW{25..31}, is saved in GR *t* and then the immediate value *i* is ORed with the system mask. Setting the PSW Q-bit, PSW{28}, to 1 with this instruction, if it was not already 1, is an undefined operation.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    if ((PSW[Q] == 0) && (i{3}))
        undefined;
    else {
        GR[t]    ←  cat(0{0..24},PSW{25..31});
        PSW[G] ←  PSW[G] | i{0};
        PSW[F] ←  PSW[F] | i{1};
        PSW[R] ←  PSW[R] | i{2};
        PSW[P] ←  PSW[P] | i{4};
        PSW[D] ←  PSW[D] | i{5};
        PSW[I] ←  PSW[I] | i{6};
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

RESET SYSTEM MASK RSM

Format: RSM i,t

00	rv	i	0	73	t
6	3	7	3	8	5

Purpose: To selectively reset bits in the system mask to 0.

Description: The current value of the system mask, PSW{25..31}, is saved in GR *t* and then the complement of the immediate value *i* is ANDed with the system mask.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    GR[t]   ←  cat(0{0..24},PSW{25..31});
    PSW[G] ←  PSW[G] & (~i{0});
    PSW[F] ←  PSW[F] & (~i{1});
    PSW[R] ←  PSW[R] & (~i{2});
    PSW[Q] ←  PSW[Q] & (~i{3});
    PSW[P] ←  PSW[P] & (~i{4});
    PSW[D] ←  PSW[D] & (~i{5});
    PSW[I]  ←  PSW[I] & (~i{6});
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: The state of the IPRs, IIA queues, and the IPSW is undefined when this instruction is used to set the Q-bit to 0, if it was not already 0.

MOVE TO SYSTEM MASK MTSM

Format: MTSM r

00	0	r	0	C3	0
6	5	5	3	8	5

Purpose: To set PSW system mask bits to a value from a register.

Description: The seven rightmost bits of GR *r* replace the system mask, PSW{25..31}. Setting the PSW Q-bit, PSW{28}, to 1 with this instruction, if it was not already 1, is an undefined operation.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    if ((PSW[Q] == 0) && (GR[r]{28} == 1))
        undefined;
    else {
        PSW[G] ←  GR[r]{25};
        PSW[F] ←  GR[r]{26};
        PSW[R] ←  GR[r]{27};
        PSW[Q] ←  GR[r]{28};
        PSW[P] ←  GR[r]{29};
        PSW[D] ←  GR[r]{30};
        PSW[I] ←  GR[r]{31};
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: The state of the IPRs, IIA queues, and the IPSW is undefined when this instruction is used to set the Q-bit to 0, if it was not already 0.

LOAD SPACE IDENTIFIER LDSID

Format: LDSID (s,b),t

00	b	rv	s	0	85	t
6	5	5	2	1	8	5

Purpose: To calculate the space register number referenced by a short pointer and copy the space register into a general register.

Description: If *s* is zero, the space identifier referenced by the leftmost two bits of GR *b* is copied into GR *t*. If *s* is not zero, SR *s* is copied into GR *t*.

Operation: GR[t] $\leftarrow$ space_select(s,GR[b]);

Exceptions: None

Notes: In Level 1 systems, this instruction may set the leftmost 16 bits of the specified GR either to zeros or to the leftmost 16 bits of the value last moved into the specified SR.

In Level 1.5 systems, this instruction may set the leftmost 8 bits of the specified GR either to zeros or to the leftmost 8 bits of the value last moved into the specified SR.

Level 0: The value 0 is written into the specified GR.

MOVE TO SPACE REGISTER — MTSP

Format: MTSP r,sr

00	rv	r	s	C1	0
6	5	5	3	8	5

Purpose: To move a value from a general register to a space register.

Description: GR *r* is copied into SR *sr* (which is assembled from the *s* field in the instruction).

Operation:

```
if (!level_0) {
      sr ← assemble_3(s);
      if (sr >= 5 && priv != 0)
            privileged_register_trap;
      else
            SR[sr] ← GR[r];
}
```

Exceptions: Privileged register trap

Restrictions: SRs 5, 6 and 7 may be changed only by software running at the most privileged level.

Notes: In Level 1 systems, the leftmost 16 bits are nonexistent bits.

In Level 1.5 systems, the leftmost 8 bits are nonexistent bits.

Level 0: This instruction executes as a null instruction.

MOVE TO CONTROL REGISTER MTCTL

Format: MTCTL r,t

00	t	r	rv	C2	0
6	5	5	3	8	5

Purpose: To move a value from a general register to a control register.

Description: GR *r* is copied into CR *t*. If CR 23 is specified, then the value is first complemented and ANDed with the original value.

Operation:

```
if (t >= 1 && t <= 7)
      undefined;
else if (level_0 && (t == 8 || t == 9 || t == 12 || t == 13 || t == 17 || t == 20))
      ; /* null instruction */
else if (t != 11 && priv != 0)
      privileged_register_trap;
else
      switch(t) {
            case 0:       if (PSW[R])
                                undefined;
                          else
                                CR[t] ← GR[r];
                          break;
            case 14: case 15: case 16: case 24: case 25: case 26:
            case 27: case 28: case 29: case 30: case 31:
                          CR[t] ← GR[r];
                          break;
            case 17: case 18: case 22:
                          if (PSW[Q])
                                undefined;
                          else
                                CR[t] ← GR[r];
                          break;
            case 23:CR[23] ← CR[23] & ~GR[r];
                          break;
            case 10:CR[10] ← GR[r]{16..31};
                          break;
            case 11:CR[11] ← GR[r]{27..31};
                          break;
            case 8: case 9: case 12: case 13:
                          CR[t] ← GR[r]{16..31};
                          break;
            case 19: case 20: case 21:
                          undefined;
```

```
                    break;
        }
```

Exceptions: Privileged register trap

Restrictions: System control registers other than the Shift Amount Register (CR 11) may be written only at the most privileged level. CR 11 may be written at any privilege level. The Recovery Counter (CR 0) may be written reliably only when the PSW[R] bit is 0. Writing into the Interruption Parameter Registers (CRs 19, 20, and 21) is an undefined operation. Writing into the Interruption Instruction Address Queues (CRs 17 and 18) and Interruption Processor Status Word (CR 22) when the PSW[Q] bit is 1 is an undefined operation.

Notes: The MTSAR pseudo-operation generates an MTCTL r,CR11 to copy a general register to the Shift Amount Register (CR 11).

Level 0: If the target control register is CR 8, 9, 12, 13, 17, or 20, this instruction executes as a null instruction.

MOVE FROM SPACE REGISTER — MFSP

Format: MFSP sr,t

00	rv	0	s	25	t
6	5	5	3	8	5

Purpose: To move a value to a general register from a space register.

Description: SR *sr* (which is assembled from the *s* field in the instruction) is copied into GR *t*.

Operation:

```
if (level_0)
      GR[t] ← 0;
else {
      sr ← assemble_3(s);
      GR[t] ← SR[sr];
}
```

Exceptions: None

Notes: In Level 1 systems, this instruction may set the leftmost 16 bits of the specified general register either to zeros or to the leftmost 16 bits of the value last moved into the specified space register.

In Level 1.5 systems, this instruction may set the leftmost 8 bits of the specified general register either to zeros or to the leftmost 8 bits of the value last moved into the specified space register.

Level 0: The value 0 is written into the specified general register.

MOVE FROM CONTROL REGISTER MFCTL

Format: MFCTL r,t

00	r	0	rv	45	t
6	5	5	3	8	5

Purpose: To move a value to a general register from a control register.

Description: CR r is copied into GR *t*.

Operation:

```
if (r >= 1 && r <= 7)
    undefined;
else if (level_0 &&(r == 8 || r == 9 || r == 12 || r == 13 || r == 17 || r == 20))
    GR[t] ← 0;
else if (priv != 0 &&!(r == 11 || r == 26 || r == 27 || (r == 16 && !PSW[S])))
    privileged_register_trap;
else if (r >= 17 && r <= 22)
    if (PSW[Q])
        undefined;
    else
        GR[t] ← CR[r];
else if (r == 0)
    if (PSW[R])
        undefined;
    else
        GR[t] ← CR[r];
else if (r >= 8)
    GR[t] ← CR[r];
```

Exceptions: Privileged register trap

Restrictions: System control registers other than the Shift Amount Register (CR 11), the Interval Timer (CR 16), and temporary registers CR 26 and CR27, may be read only at the most privileged level. CR 11, CR 26, and CR 27 may be read at any privilege level. CR 16 may be read at any privilege level only if the PSW S-bit is 0; otherwise, CR 16 may be read only at the most privileged level. The Interruption Instruction Address Queues (CRs 17 and 18) and Interruption Parameter Registers (CRs 19, 20, and 21) and the Interruption Processor Status Word (CR 22) may be read reliably only when PSW[Q] bit is 0.

Level 0: If the source control register is CR 8, 9, 12, 13, 17, or 20, a 0 is written into the specified general register.

SYNCHRONIZE CACHES SYNC

Format: SYNC

00	rv	0	rv	0	20	0
6	5	1	4	3	8	5

Purpose: To enforce program order of instruction execution.

Description: Any load, store, semaphore, cache flush, or cache purge instructions that follow the SYNC instruction get executed only after all such instructions prior to the SYNC instruction have completed executing. On implementations which execute such instructions out of sequence, this instruction enforces program ordering.

Operation: Enforce program order of memory references

Exceptions: None

Notes: In systems in which all memory references are performed in order, this instruction executes as a null instruction.

PROGRAMMING NOTE

The minimum spacing that is guaranteed to work for "self-modifying code" is shown in the code segment below. Since instruction prefetching is permitted, any data cache flushes must be separated from any instruction cache flushes by a SYNC. This will ensure that the "new" instruction will be written to memory prior to any attempts at prefetching it as an instruction.

```
            LDIL    l%newinstr,rnew
            LDW     r%newinstr(0,rnew),temp
            LDIL    l%instr,rinstr
            STW     temp,r%instr(0,rinstr)
            FDC     r%instr(0,rinstr)
            SYNC
            FIC     r%instr(0,rinstr)
            SYNC
            (at least seven instructions)
instr       . . .
```

This sequence assumes a uniprocessor system. In a multiprocessor system, software must ensure no processor is executing code which is in the process of being modified.

SYNCHRONIZE DMA **SYNCDMA**

Format: SYNCDMA

00	rv	1	rv	0	20	0
6	5	1	4	3	8	5

Purpose: To enforce DMA completion order.

Description: On implementations which can signal DMA completion prior to achieving cache coherence, this instruction enforces ordering. All cache coherence actions which are outstanding as a consequence of prior DMA operations must be completed before the next memory access is performed.

Operation: Enforce DMA completion order

Exceptions: None

Notes: In systems in which all DMA operations are performed in order, this instruction executes as a null instruction.

PROBE READ ACCESS PROBER

Format: PROBER (s,b),r,t

01	b	r	s	46	0	t
6	5	5	2	8	1	5

Purpose: To determine whether read access to a given address is allowed.

Description: A test is performed to determine if read access to the address computed by the instruction is permitted at the privilege level given by the two rightmost bits of the GR *r*. GR *t* is set to 1 if the test succeeds and 0 otherwise.

This instruction checks the read access rights for the page. If the PSW P-bit is 1, the protection IDs are also checked. The instruction performs data address translation regardless of the state of the PSW D-bit.

Operation:

```
space ← space_select(s,GR[b]);
offset ← GR[b];
if (search_DTLB(space,offset,&entry))
    if (read_access_allowed(space,offset,GR[r]))
        GR[t] ← 1;
    else
        GR[t] ← 0;
else
    non-access_data_TLB_miss_fault();
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Notes: If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system's handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the IPSW[N] bit to 1, and return to the interrupting instruction.

Level 0: This instruction always sets the target general register to 1.

PROBE READ ACCESS IMMEDIATE — PROBERI

Format: PROBERI (s,b),i,t

01	b	i	s	C6	0	t
6	5	5	2	8	1	5

Purpose: To determine whether read access to a given address is allowed.

Description: A test is performed to determine if read access to the address computed by the instruction is permitted at the privilege level given by the two rightmost bits of the immediate value *i*. GR *t* is set to 1 if the test succeeds and 0 otherwise.

This instruction checks the read access rights for the page. If the PSW P-bit is 1, the protection IDs are also checked. This instruction performs data address translation regardless of the state of the PSW D-bit.

Operation:

```
space ← space_select(s,GR[b]);
offset ← GR[b];
if (search_DTLB(space,offset,&entry))
    if (read_access_allowed(space,offset,i))
        GR[t] ← 1;
    else
        GR[t] ← 0;
else
    non-access_data_TLB_miss_fault();
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Notes: If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system's handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the IPSW[N] bit to 1, and return to the interrupting instruction.

Level 0: This instruction always sets the target general register to 1.

PROBE WRITE ACCESS — PROBEW

Format: PROBEW (s,b),r,t

01	b	r	s	47	0	t
6	5	5	2	8	1	5

Purpose: To determine whether write access to a given address is allowed.

Description: A test is performed to determine if write access to the address computed by the instruction is permitted at the privilege level given by the two rightmost bits of the GR *r*. GR *t* is set to 1 if the test succeeds and 0 otherwise.

This instruction checks the write access rights for the page. If the PSW P-bit is 1, the protection IDs are also checked. This instruction performs data address translation regardless of the state of the PSW D-bit.

Operation:

```
space ← space_select(s,GR[b]);
offset ← GR[b];
if (search_DTLB(space,offset,&entry))
    if (write_access_allowed(space,offset,GR[r]))
        GR[t] ← 1;
    else
        GR[t] ← 0;
else
    non-access_data_TLB_miss_fault();
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Notes: If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system's handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the IPSW[N] bit to 1, and return to the interrupting instruction.

Level 0: This instruction always sets the target general register to 1.

PROBE WRITE ACCESS IMMEDIATE — PROBEWI

Format: PROBEWI (s,b),i,t

01	b	i	s	C7	0	t
6	5	5	2	8	1	5

Purpose: To determine whether write access to a given address is allowed.

Description: A test is performed to determine if write access to the address computed by the instruction is permitted at the privilege level given by the two rightmost bits of the immediate value *i*. GR *t* is set to 1 if the test succeeds and 0 otherwise.

This instruction checks the write access rights for the page. If the PSW P-bit is set, the protection IDs are also checked. This instruction performs data address translation regardless of the state of the PSW D-bit.

Operation:

```
space ← space_select(s,GR[b]);
offset ← GR[b];
if (search_DTLB(space,offset,&entry))
    if (write_access_allowed(space,offset,i))
        GR[t] ← 1;
    else
        GR[t] ← 0;
else
    non-access_data_TLB_miss_fault();
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Notes: If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system's handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the IPSW[N] bit to 1, and return to the interrupting instruction.

Level 0: This instruction always sets the target general register to 1.

LOAD PHYSICAL ADDRESS LPA

Format: LPA,cmplt x(s,b),t

01	b	x	s	4D	m	t
6	5	5	2	8	1	5

Purpose: To determine the absolute address of a mapped virtual page.

Description: The effective address is calculated. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus index register *x*. The completer, encoded in the *m* field of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) GR *t* receives the absolute address corresponding to the given virtual address. If the page is not present, GR *t* is set to 0. If base register modification is specified and *b* = *t*, the value loaded is the absolute address of the item indicated by the effective address.

In systems with separate data and instruction TLBs, the absolute address is obtained from the data TLB. This instruction performs data address translation regardless of the state of the PSW D-bit.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    space ← space_select(s,GR[b]);
    switch (cmplt) {
        case M:  offset ← GR[b];                              /*m=1*/
                 GR[b] ← GR[b] + GR[x];
                 break;
        default: offset ← GR[b] + GR[x];                      /*m=0*/
    }
    if (search_DTLB(space,offset,&entry))
        GR[t] ← absolute_address(space,offset);
    else
        non-access_data_TLB_miss_fault();
}
```

Exceptions: Non-access data TLB miss fault   Privileged operation trap

Restrictions: The result of LPA is ambiguous for an address which maps to absolute address 0.

This instruction may be executed only at the most privileged level.

Notes: If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system's handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the

IPSW[N] bit to 1, and return to the interrupting instruction.

Level 0: This instruction is an undefined operation.

LOAD COHERENCE INDEX **LCI**

Format: LCI x(s,b),t

01	b	x	s	4C	0	t
6	5	5	2	8	1	5

Purpose: To determine the coherence index corresponding to a virtual address.

Description: The effective address is calculated. GR *t* receives the coherence index corresponding to the given virtual address.

In systems with separate data and instruction caches, the coherence index is obtained from the data cache.

The coherence index function is independent of the state of the PSW D-bit.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else {
    space ← space_select(s,GR[b]);
    offset ← GR[b] + GR[x];
    GR[t] ← coherence_index(space,offset);
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: All addresses within a page have the same coherence index.

The coherence index corresponding to a physical address can be determined by performing LCI on the equivalently-mapped virtual address. Also, in order to allow I/O modules to have coherent access to equivalently-mapped addresses without knowing the coherence index, the coherence index for equivalently-mapped addresses must be an implementation-defined function of the physical address bits only.

Two virtual addresses having the same coherence index are not guaranteed to alias unless they also meet the virtual aliasing rules.

For systems that do not have a cache, the target register receives an undefined value.

For system that do not support coherent I/O, this instruction is undefined.

Level 0: This instruction is an undefined operation.

Format: PDTLB,cmplt x(s,b)

01	b	x	s	48	m	rv
6	5	5	2	8	1	5

Purpose: To invalidate a data TLB entry.

Description: The data or combined TLB entry (if any) for the page specified by the effective address generated by the instruction is invalidated, removed, or altered. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus index register *x*. The completer, encoded in the *m* field of the instruction, specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

In a multiprocessor system, a purge request is broadcast to all data and combined TLBs. The other processors must invalidate, remove, or alter the entry before the issuing processor continues.

Operation:

```
if (!level_0) {
      if (priv != 0)
            privileged_operation_trap;
      else {
            space ← space_select(s,GR[b]);
            switch (cmplt) {
                  case M:  offset ← GR[b];                          /*m=1*/
                           GR[b] ← GR[b] + GR[x];
                           break;
                  default: offset ← GR[b] + GR[x];                  /*m=0*/
            }
            if (search_DTLB(space, offset{0..19},&entry))
                  purge_DTLB(entry);
            broadcast_purge_DTLB(space,offset);
      }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries from a combined TLB.

Level 0: This instruction executes as a null instruction.

PURGE INSTRUCTION TLB PITLB

Format: PITLB,cmplt x(sr,b)

01	b	x	s	08	m	rv
6	5	5	3	7	1	5

Purpose: To invalidate an instruction TLB entry.

Description: The instruction or combined TLB entry (if any) for the page specified by the effective address generated by the instruction is invalidated, removed, or altered. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m* field of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

In a multiprocessor system, a purge request is broadcast to all instruction and combined TLBs. The other processors must invalidate, remove, or alter the entry before the issuing processor continues.

Operation:

```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← SR[assemble_3(s)];
        switch (cmplt) {
            case M:  offset ← GR[b];                       /*m=1*/
                     GR[b] ← GR[b] + GR[x];
                     break;
            default: offset ← GR[b] + GR[x];               /*m=0*/
        }
        if (search_ITLB(space,offset{0..19},&entry))
            purge_ITLB(entry);
        broadcast_purge_ITLB(space,offset);
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries from a combined TLB.

Level 0: This instruction executes as a null instruction.

Format: PDTLBE,cmplt x(s,b)

01	b	x	s	49	m	rv
6	5	5	2	8	1	5

Purpose: To invalidate a data TLB entry without matching the address portion.

Description: The data or combined TLB entries (if any) specified by an implementation-dependent function of the effective address generated by the instruction are invalidated or removed. All the fields of these entries may be changed to arbitrary values as long as these entries do not validate any subsequent accesses. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m* field of the instruction, specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

This is an implementation-dependent instruction that can be used to purge the entire data TLB without knowing the translations in the TLB. No broadcast occurs in a multiprocessor system.

Operation:

```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← space_select(s,GR[b]);
        switch (cmplt) {
            case M: offset ← GR[b];                      /*m=1*/
                    GR[b] ← GR[b] + GR[x];
                    break;
            default: offset ← GR[b] + GR[x];             /*m=0*/
        }
        entries ← select_DTLB_entries(space,offset);
        purge_DTLB_entry(entries);
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries from a combined TLB. This instruction does not necessarily purge the entry specified by “space” and “offset”.

Level 0: This instruction executes as a null instruction.

PURGE INSTRUCTION TLB ENTRY PITLBE

Format: PITLBE,cmplt x(sr,b)

01	b	x	s	09	m	rv
6	5	5	3	7	1	5

Purpose: To invalidate an instruction TLB entry without matching the address portion.

Description: The instruction or combined TLB entries (if any) specified by an implementation-dependent function of the effective address generated by the instruction are invalidated or removed. All the fields of these entries may be changed to arbitrary values as long as these entries do not validate any subsequent accesses. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m* field of the instruction, specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) The space register, *sr*, is encoded in the *s* field of the instruction.

This is an implementation-dependent instruction that can be used to purge the entire instruction TLB without knowing the translations in the TLB. No broadcast occurs in a multiprocessor system.

Operation:

```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← SR[assemble_3(s)];
        switch (cmplt) {
            case M: offset ← GR[b];                    /*m=1*/
                    GR[b] ← GR[b] + GR[x];
                    break;
            default: offset ← GR[b] + GR[x];           /*m=0*/
        }
        entries ← select_ITLB_entries(space,offset);
        purge_ITLB_entry(entries);
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries from a combined TLB. This instruction does not necessarily purge the entry specified by “space” and “offset”.

Level 0: This instruction executes as a null instruction.

INSERT DATA TLB ADDRESS IDTLBA

Format: IDTLBA r,(s,b)

01	b	r	s	41	0	0
6	5	5	2	8	1	5

Purpose: To begin adding an entry to the data TLB (INSERT DATA TLB PROTECTION completes the process).

Description: A slot is found in the data or combined TLB and the new translation is placed there in the invalid state. If the data or combined TLB already contains an entry with the virtual page number of the new translation, the entry must be removed. The base register, *b*, forms the offset portion of the address. The TLB tag and translation (constructed from the effective address and GR *r*) are loaded into that slot. The physical page number is obtained from GR *r*.

Operation:
```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← space_select(s,GR[b]);
        offset ← GR[b];
        if (!search_DTLB(space,offset,&entry))
            alloc_DTLB(space,offset,&entry);
        DTLB[entry].ENTRY_VALID ← false;
        DTLB[entry].VIRTUAL_ADDR ← cat(space,offset{0..19});
        DTLB[entry].PHY_PAGE_NO ← GR[r]{7..26};
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries into a combined TLB.

Level 0: This instruction executes as a null instruction.

INSERT INSTRUCTION TLB ADDRESS — IITLBA

Format: IITLBA r,(sr,b)

01	b	r	s	01	0	0
6	5	5	3	7	1	5

Purpose: To begin adding an entry to the instruction TLB (INSERT INSTRUCTION TLB PROTECTION completes the process).

Description: A slot is found in the instruction or combined TLB and the new translation is placed there in the invalid state. If the instruction or combined TLB already contains an entry with the virtual page number of the new translation, the entry must be removed. The base register, *b*, forms the offset portion of the address. The TLB tag and translation (constructed from the effective address and GR *r*) are loaded into that slot. The physical page number is obtained from GR *r*. The space register, *sr*, is encoded in the *s* field of the instruction.

Operation:

```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← SR[assemble_3(s)];
        offset ← GR[b];
        if (!search_ITLB(space,offset,&entry))
            alloc_ITLB(space,offset,&entry);
        ITLB[entry].ENTRY_VALID ← false;
        ITLB[entry].VIRTUAL_ADDR ← cat(space,offset{0..19});
        ITLB[entry].PHY_PAGE_NO ← GR[r]{7..26};
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries into a combined TLB.

Level 0: This instruction executes as a null instruction.

INSERT DATA TLB PROTECTION IDTLBP

Format: IDTLBP r,(s,b)

01	b	r	s	40	0	0
6	5	5	2	8	1	5

Purpose: To finish adding an entry to the data TLB (INSERT DATA TLB ADDRESS begins the process).

Description: The virtual page number computed from the instruction is checked for a match against entries in the data or combined TLB. The base register, *b*, forms the offset portion of the address. If a match is found, GR *r* is loaded into the TLB entry as the flags and access control, and the entry is marked valid. Otherwise, this instruction executes as a null instruction.

Operation:

```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← space_select(s,GR[b]);
        offset ← GR[b];
        if (search_DTLB(space,offset,&entry)) {
            DTLB[entry].U ← GR[r]{12};
            DTLB[entry].T ← GR[r]{2};
            DTLB[entry].D ← GR[r]{3};
            DTLB[entry].B ← GR[r]{4};
            DTLB[entry].ACCESS_RIGHTS ← GR[r]{5..11};
            DTLB[entry].ACCESS_ID ← GR[r]{13..30};
            DTLB[entry].ENTRY_VALID ← true;
        }
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries into a combined TLB.

If smaller than 18-bit access IDs are implemented, only the appropriate number of the rightmost bits of GR[r]{13..30} are stored in the TLB.

Level 0: This instruction executes as a null instruction.

INSERT INSTRUCTION TLB PROTECTION IITLBP

Format: IITLBP r,(sr,b)

01	b	r	s	00	0	0
6	5	5	3	7	1	5

Purpose: To finish adding an entry to the instruction TLB (INSERT INSTRUCTION TLB ADDRESS begins the process).

Description: The virtual page number computed from the instruction is checked for a match against entries in the instruction or combined TLB. The base register, *b*, forms the offset portion of the address. If a match is found, GR *r* is loaded into the TLB entry as the flags and access control, and the entry is marked valid. Otherwise, this instruction executes as a null instruction. The space register, *sr*, is encoded in the *s* field of the instruction.

Operation:

```
if (!level_0) {
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← SR[assemble_3(s)];
        offset ← GR[b];
        if (search_ITLB(space,offset,&entry)) {
            if (combined_TLB) {
                ITLB[entry].U ← GR[r]{12};
                ITLB[entry].T ← GR[r]{2};
                ITLB[entry].D ← GR[r]{3};
                ITLB[entry].B ← GR[r]{4};
            }
            ITLB[entry].ACCESS_RIGHTS ← GR[r]{5..11};
            ITLB[entry].ACCESS_ID ← GR[r]{13..30};
            ITLB[entry].ENTRY_VALID ← true;
        }
    }
}
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries into a combined TLB. The U, T, D, and B bits are set to the appropriate bits of GR *r* in that case.

If smaller than 18-bit access IDs are implemented, only the appropriate number of rightmost bits of GR[r]{13..30} are stored in the TLB.

Level 0: This instruction executes as a null instruction.

PURGE DATA CACHE PDC

Format: PDC,cmplt x(s,b)

01	b	x	s	4E	m	0
6	5	5	2	8	1	5

Purpose: To invalidate a data cache line.

Description: The cache line (if present) specified by the effective address generated by the instruction is invalidated from the data cache. If the privilege level is non-zero and the cache line is dirty then it is written back to memory before being invalidated. If the privilege level is zero and the line is dirty then the implementation may optionally write back the line to memory.

The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m* field of the instruction, specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

If a cache purge operation is performed, write access to the data is required. The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

In a multiprocessor system, a purge or flush request is broadcast to all data and combined caches.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case M:  offset ← GR[b];                                  /*m=1*/
             GR[b] ← GR[b] + GR[x];
             break;
    default: offset ← GR[b] + GR[x];                          /*m=0*/
}
if (priv != 0)
    flush_data_cache(space,offset);
else
    purge_or_flush_data_cache(space,offset);
```

Exceptions:

- Non-access data TLB miss fault
- Data memory access rights trap
- Data memory protection ID trap
- Data memory break trap
- Data debug trap

Notes: For systems that do not have a cache, this instruction executes as a null instruction.

At privilege level zero, implementations are encouraged to purge the cache line for performance reasons.

This instruction may be executed out of sequence but must satisfy the instruction ordering constraints. The SYNC instruction enforces program order with respect to the instructions following the SYNC.

A data debug trap is not taken if a PDC causes a cache flush operation; the trap is taken only if a cache purge operation is performed to an address matching a data breakpoint with its *w* (write access) bit enabled..

FLUSH DATA CACHE FDC

Format: FDC,cmplt x(s,b)

01	b	x	s	4A	m	rv
6	5	5	2	8	1	5

Purpose: To invalidate a data cache line and write it back to memory if it is dirty.

Description: The data cache line (if present) specified by the effective address generated by the instruction is written back to memory, if and only if it is dirty, and then invalidated from the data cache. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m*-field of the instruction, specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

A cache line is called *dirty* if any byte has been written to since it was read from memory or if a STBYS,E to the leftmost byte of a word has been performed.

In a multiprocessor system, a flush request is broadcast to all data and combined caches.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case M:  offset ← GR[b];                          /*m=1*/
             GR[b] ← GR[b] + GR[x];
             break;
    default: offset ← GR[b] + GR[x];                  /*m=0*/
}
flush_data_cache(space,offset);
```

Exceptions: Non-access data TLB miss fault

Notes: For systems that do not have a cache, this instruction executes as a null instruction.

In systems with a combined cache, this instruction may be used to flush both data and instruction lines from the cache.

This instruction may be executed out of sequence but must satisfy the instruction ordering constraints. The SYNC instruction enforces program order with respect to the instructions following the SYNC.

FLUSH INSTRUCTION CACHE FIC

Format: FIC,cmplt x(sr,b)

01	b	x	s	0A	m	rv
6	5	5	3	7	1	5

Purpose: To invalidate an instruction cache line.

Description: The instruction cache line (if any) specified by the effective address generated by the instruction is invalidated in the instruction cache. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m*-field of the instruction, specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) The space register, *sr*, is encoded in the *s* field of the instruction. The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

Either the instruction TLB or the data TLB can be used to perform the address translation for the address to be flushed. If the data TLB is used, a TLB miss fault is reported using a non-access data TLB miss fault.

In a multiprocessor system, a flush request is broadcast to all instruction and combined caches.

Operation:

```
space ← SR[assemble_3(s)];
switch (cmplt) {
    case M:  offset ← GR[b];                              /*m=1*/
             GR[b] ← GR[b] + GR[x];
             break;
    default: offset ← GR[b] + GR[x];                      /*m=0*/
}
flush_instruction_cache(space,offset);
```

Exceptions: Non-access instruction TLB miss fault
Non-access data TLB miss fault

Notes: For systems that do not have a cache, this instruction executes as a null instruction.

In systems with a combined cache, this instruction may be used to flush both instruction and data lines from the cache, including writing them back to main memory, if they are dirty.

This instruction may be executed out of sequence but must satisfy the instruction ordering constraints. The SYNC instruction enforces program order with respect to the instructions following the SYNC.

FLUSH DATA CACHE ENTRY FDCE

Format: FDCE,cmplt x(s,b)

01	b	x	s	4B	m	rv
6	5	5	2	8	1	5

Purpose: To provide for flushing the entire data or combined cache by causing zero or more cache lines to be invalidated.

Description: Zero or more cache lines specified by an implementation-dependent function of the effective address are written back to main memory, if and only if they are dirty, and are invalidated in the data or combined cache. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m*-field of the instruction, specifies base register modification. No address translation is performed. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

When this instruction is used in an architecturally defined cache flush loop, the entire data or combined cache will be flushed upon completion of the loop.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case M:  offset ← GR[b];                              /*m=1*/
             GR[b] ← GR[b] + GR[x];
             break;
    default: offset ← GR[b] + GR[x];                      /*m=0*/
}
entries ← select_data_cache_entries(space,offset);
flush_data_cache_entry(entries);
```

Exceptions: None

Notes: In a multiprocessor system, this instruction is not broadcast to other processors. This instruction does not necessarily flush the entry specified by “space” and “offset”.

For systems that do not have a cache, this instruction executes as a null instruction.

FLUSH INSTRUCTION CACHE ENTRY FICE

Format: FICE,cmplt x(sr,b)

01	b	x	s	0B	m	rv
6	5	5	3	7	1	5

Purpose: To provide for flushing the entire instruction or combined cache by causing zero or more cache lines to be invalidated.

Description: Zero or more cache lines specified by an implementation-dependent function of the effective address are invalidated in the instruction or combined cache. For implementations with a combined cache, the cache lines are written back to main memory, if and only if they are dirty, and are invalidated. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m*-field of the instruction, specifies base register modification. No address translation is performed. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.) The space register, *sr*, is encoded in the *s* field of the instruction.

When this instruction is used in an architecturally defined cache flush loop, the entire instruction or combined cache will be flushed upon completion of the loop (all the contents of the instruction cache, except the loop itself, prior to the beginning of the flush loop must be flushed).

Operation:

```
switch (cmplt) {
    case M:  offset ← GR[b];                               /*m=1*/
             GR[b] ← GR[b] + GR[x];
             break;
    default: offset ← GR[b] + GR[x];                       /*m=0*/
}
space ← SR[assemble_3(s)];
entries ← select_instruction_cache_entries(space,offset);
flush_instruction_cache_entry(entries);
```

Exceptions: None

Notes: In a multiprocessor system, this instruction is not broadcast to other processors. This instruction does not necessarily flush the entry specified by “space” and “offset”.

For systems which do not have a cache, this instruction executes as a null instruction.

DIAGNOSE DIAG

Format: DIAG i

05	im26
6	26

Purpose: To provide implementation-dependent operations for system initialization, reconfiguration, and diagnostic purposes.

Description: The immediate value in the assembly language is encoded in the *im26* field of the instruction. Refer to the hardware reference manual for the definition on a particular machine implementation.

Operation:

```
if (priv != 0)
    privileged_operation_trap;
else
    implementation_dependent;
```

Exceptions: Privileged operation trap
Implementation-dependent.

Restrictions: This instruction may be executed only at the most privileged level.

Notes: Since the DIAG instruction is privileged, a privileged operation trap will result from unprivileged diagnostic software executing DIAG. The trap could invoke an emulator which would allow the unprivileged software access to the required unprivileged implementation-dependent resources.

Assist Instructions

The PA-RISC design generally conforms to the concept of a simple instruction set implemented in cost effective hardware. Certain algorithms can benefit from substantial performance gains by dedicating specialized hardware to execute specialized instructions. Few algorithms rely solely upon the specialized hardware alone and it is advantageous to combine the processor with additional assist processors closely coupled to it.

In addition to the instructions executed by a central processor, the instruction set contains instructions to invoke the special, optional, hardware functions provided by the two types of assist processors: Special Function Units (SFUs) and coprocessors.

Special function units are closely coupled to the central processor and provide extensions to the instruction set. They use the general registers as operands and targets of operations.

Coprocessors provide functions that use either memory locations or coprocessor registers as operands and targets of operations. Coprocessors are less closely coupled to the central processor, and so, are more easily provided as configuration options for an implementation than special function units. Coprocessors may also directly pass double-word quantities to and from the coprocessor and memory. This is suited to the manipulation of quantities that are too large to be directly handled in general registers.

The special function unit and coprocessor instructions are intended to encapsulate all of the optional hardware features used for non-system level code. An emulation facility is provided that permits PA-RISC family members to execute code using the standard instruction set when optional hardware is not present. The emulation facility is provided by the assist emulation trap, which passes information in control registers, substantially reducing the instruction path length for emulation.

The assist exception trap permits partial implementations of standard "hardware" functions in a combination of hardware and software. This handles functions that are difficult or not cost-effective to implement fully in hardware.

Compatibility Among Implementations

The standard PA-RISC instruction set contains all defined instructions, including those for all defined assist processors. Particular implementations may choose to implement these instructions in hardware, software, or some combination of the two, using assist emulation traps and/or assist exception traps to complete the implementation. Thus, these instructions can be used by compilers and assemblers without sacrificing object-code portability. Software emulation of the extended functions is also used to permit execution of the object code in a degraded mode for high-availability systems.

Architecturally Defined Assist Processors

Some assist processors are defined and compilers are capable of generating instructions for them. Trap handlers can emulate these instructions if the hardware is nonexistent or incomplete.

These assist processors are standardized for software portability because most implementations will include them as either standard or optional hardware.

Special Function Unit (SFU) Instructions

The SFU mechanism is intended for certain architecturally defined instruction extensions, such as hardware fixed-point binary multiply/divide or encryption hardware, as well as for implementation-specific extensions, such as emulation assist processors or direct I/O controller connections.

SFUs are connected to the general register interface and are invoked by special operation instructions. These instructions cause the execution unit to perform any of several operations (determined by the opcode extension), which may use the contents of registers, or may write back a result. Some instructions conditionally nullify the next instruction.

Some special function operations overlap their execution with succeeding instructions. These operations require that the special function unit's state be saved and restored when a context switch is made. An interlock occurs if a special function result is requested before the operation has completed, or the special function unit is busy.

An SFU is not required to hold its state in addressable registers. Instead, SFU operations are used to save and restore the state, as well as to pass it operands and receive results from it.

Defined special function units will conform to the requirements of the defined SFU instructions, so that they may be implemented either as built-in or interfaced special function units. The assist emulation trap permits software implementation of any defined special operation instruction.

The processor must also provide the current privilege level to special function units. Privilege levels could be broadcast each time they change or could be transmitted with each SFU operation. Use of the privilege level by the SFU is specific to each of the units. The operation paragraph of each SFU instruction description specifies the necessary information that must be available to the SFU in the "sfu_operation" function.

There is one SFU instruction, the IDENTIFY SFU (SPOP1) instruction, that is defined for all SFUs. It must be implemented.

There are four instruction formats for the special function unit instructions as given below. The major opcode is 0x04 for SFU operations.

1. Special Operation Zero

04	sop1	0	sfu	n	sop2
6	15	2	3	1	5

2. Special Operation One

04	sop	1	sfu	n	t
6	15	2	3	1	5

3. Special Operation Two

04	r	sop1	2	sfu	n	sop2
6	5	10	2	3	1	5

4. Special Operation Three

04	r2	r1	sop1	3	sfu	n	sop2
6	5	5	5	2	3	1	5

The SCR (SFU Configuration Register) is an 8-bit control register, in CR 10 bits 16..23, that is used to indicate the presence and usability of a hardware implementation of an SFU. Bit 1 in the SCR corresponds to the debug SFU. For all other bits in the SCR, SCR{i} corresponds to an undefined SFU with a unit identifier that is the same as the bit position, that is the SFU with uid i.

When SCR{i} is 1, the SFU with uid i is implied to be present and usable. SFU instructions are passed to the SFU and the defined operation occurs. Exceptions resulting from the operation cause the instruction to be terminated with an assist exception trap. Assist emulation traps are not allowed to occur for the SFU with uid i when SCR{i} is 1. It is an undefined operation to set to 1 the SCR bit corresponding to a nonexistent SFU.

When SCR{i} is 0, it is not implied that the SFU with uid i is absent from the system, but rather that the SFU, if present, is not currently being used. When the SCR bit is 0, the SFU instruction is terminated with an assist emulation trap. Assist exception traps are not allowed to occur for the SFU with uid i when SCR{i} is 0.

Setting the SCR{i} bit to 0 must logically decouple the SFU with uid i. This must ensure that the state of the SFU with uid i is frozen just prior to the transition of SCR{i} from 1 to 0 and that the state does not change as long as SCR{i} is 0. When SCR{i} is 0, the SFU with uid i must not respond to any SFU operations for the SFU with uid i. The frozen state of an SFU, for example, could also be a state in which the SFU is left "armed" to trap any subsequent operations. For example, if the coprocessor with uid i is in an "armed-to-trap" state and SCR{i} is 0, any operation involving that SFU must not cause an assist exception trap.

The precedence of the interruptions that are applicable to operations for the SFU with uid i depends on the state of SCR{i}. The assist exception trap and assist emulation trap are always taken in the priority order as described in “Interruptions” on page 4-13.

NOTE

Logical decoupling may be accomplished in a variety of ways. Processors may use abort signals or other schemes to notify SFUs that the current instruction is to be ignored.

When the SCR bit is 0, logical decoupling suppresses any exception traps from a SFU and causes the emulation trap to occur (if it is the highest priority).

Coprocessor Instructions

The coprocessor mechanism is intended for special-purpose data manipulations, especially those which handle data larger than single words. The interconnection method allows for instruction set extensions with minimal effect on the instruction execution rate, while maintaining short data communication paths between the coprocessors and the rest of the system. Coprocessor instructions can be executed by the coprocessor hardware or emulated by software. Combinations of instructions implemented in hardware and emulated by software are possible even when the coprocessor hardware is present in a system.

When caches are implemented, coprocessors are connected to the CPU-cache interface. For systems that do not have a cache, coprocessors are connected to the CPU-memory bus interface. Coprocessors manipulate data in their own register sets, but use the data cache or memory bus and central processor's address generation logic. Under control of the CPU, coprocessor load instructions pass data from the data cache or memory bus to a coprocessor, and coprocessor store instructions from a coprocessor to the data cache or memory bus. Coprocessor operations use only the coprocessor's registers. Some coprocessor operations may nullify the next instruction.

Coprocessor operation, load, and store instructions may overlap their execution with following instructions. An interlock occurs if a coprocessor operation is requested before the coprocessor is able to perform it, and for loads and stores involving busy coprocessor registers.

The coprocessor load and store instructions contain a 5-bit field which normally specifies a coprocessor register, but may also be interpreted by coprocessors as a sub-operation field. Coprocessors keep their state in their registers, so that storing the coprocessor registers and reloading them is sufficient to save and restore the state of a coprocessor.

Some coprocessors are capable of supporting double-word load and store operations. These operations are implemented on all systems that support such coprocessors, even though they may require additional cycles for some machines. Coprocessor load and store instructions to I/O address spaces are undefined. Coprocessor load and store operations must be atomic.

The operation section of each coprocessor instruction description specifies the necessary information that must be available to the coprocessor in the *coprocessor_op* and *send_to_copr* functions. There is one coprocessor instruction, the IDENTIFY COPROCESSOR (COPR,0,0) instruction, that is defined for coprocessors with unit identifiers 4 through 7. Coprocessors with unit identifiers 0 and 3 have a mechanism to identify themselves that is individually defined.

There are five instruction formats for the coprocessor instructions, as given below. The major opcodes are 0x0C for coprocessor operations, 0x09 for single word loads and stores, and 0x0B for double word loads and stores.

1. Coprocessor Operation

0C	sop1	uid	n	sop2
6	17	3	1	5

2. Coprocessor Indexed Loads

09/0B	b	x	s	u	0	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

3. Coprocessor Indexed Stores

09/0B	b	x	s	u	0	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

4. Coprocessor Short Displacement Loads

09/0B	b	im5	s	a	1	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

5. Coprocessor Short Displacement Stores

09/0B	b	im5	s	a	1	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

NOTE

An unaligned data reference trap is taken if the appropriate number of rightmost bits of the effective virtual address are not zeros for the COPROCESSOR LOAD WORD INDEXED, COPROCESSOR LOAD DOUBLEWORD INDEXED, COPROCESSOR STORE WORD INDEXED, COPROCESSOR STORE DOUBLEWORD INDEXED, COPROCESSOR LOAD WORD SHORT, COPROCESSOR LOAD DOUBLEWORD SHORT, COPROCESSOR STORE WORD SHORT, and COPROCESSOR STORE DOUBLEWORD SHORT instructions. Absolute accesses to unaligned data are undefined operations.

The CCR (Coprocessor Configuration Register) is an 8-bit control register, in CR 10 bits 24..31, that is used to indicate the presence and usability of a hardware implementation of a coprocessor. Bits 0 and 1 in the CCR correspond to the floating-point coprocessor and Bit 2 in the CCR corresponds to the performance monitor coprocessor. For all other bits in the CCR, CCR{i} corresponds to an undefined coprocessor with a unit identifier that is the same as the bit position, that is the coprocessor with uid i.

Execution of any floating-point instruction with CCR{0} and CCR{1} not set to the same value is an undefined operation. Execution of a coprocessor operation instruction (major opcode 0x0C) with CCR{0}, CCR{1}, and the uid field in the instruction all set to 1 is an undefined operation.

When CCR{i} is 1, the coprocessor with uid i is implied to be present and usable. Coprocessor instructions are passed to the coprocessor and the defined operation occurs. Exceptions resulting from the operation cause the instruction to be terminated with an assist exception trap. Assist emulation traps are not allowed to occur for the coprocessor with uid i when CCR{i} is 1. It is an undefined operation to set to 1 the CCR bit corresponding to a nonexistent coprocessor.

When CCR{i} is 0, it is not implied that the coprocessor with uid i is absent from the system, but rather that the coprocessor, if present, is not currently being used. When the CCR bit is 0, the coprocessor instruction is terminated with an assist emulation trap. Assist exception traps are not allowed to occur for the coprocessor with uid i when CCR{i} is 0.

Setting the CCR{i} bit to 0 must logically decouple the coprocessor with uid i. This must ensure that the state of the coprocessor with uid i is frozen just prior to the transition of CCR{i} from 1 to 0 and that the state does not change as long as CCR{i} is 0. When CCR{i} is 0, the coprocessor with uid i must not respond to any coprocessor operations for the coprocessor with uid i. The frozen state of a coprocessor, for example, could also be a state in which the coprocessor is left "armed" to trap any subsequent operations. For example, if the coprocessor with uid i is in an "armed-to-trap" state and CCR{i} is 0, any operation involving that coprocessor must not cause an assist exception trap.

The precedence of the interruptions that are applicable to operations for the coprocessor with uid i depends on the state of CCR{i}. The assist exception trap and assist emulation trap are always taken in the priority order as described in “Interruptions” on page 4-13.

NOTE

Logical decoupling may be accomplished in a variety of ways. Processors may use abort signals or other schemes to notify coprocessors that the current instruction is to be ignored.

When the CCR bit is 0, logical decoupling suppresses any exception traps from a coprocessor and causes the emulation trap to occur (if it is the highest priority).

SPECIAL OPERATION ZERO — SPOP0

Format: SPOP0,sfu,sop,n

04	sop1	0	sfu	n	sop2
6	15	2	3	1	5

Purpose: To invoke a special function unit operation.

Description: The SFU identified by *sfu* is directed to perform the operation specified by the information supplied to it. If nullification is specified in the instruction, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

The *sop* field in the assembly language format is the concatenation of the *sop1* and *sop2* fields in the machine instruction, sop = cat(sop1,sop2).

Operation:

```
sfu_operation0(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31});
if (n && sfu_condition0(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31}))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

SPECIAL OPERATION ONE SPOP1

Format: SPOP1,sfu,sop,n t

04	sop	1	sfu	n	t
6	15	2	3	1	5

Purpose: To copy a special function unit register or a result to a general register.

Description: A single word is sent from the SFU identified by *sfu* to GR *t*. The SFU uses its internal state and the instruction fields supplied to it to compute or select the result. If nullification is specified in the instruction, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

Operation:

```
GR[t] ← sfu_operation1(cat(sop,sfu,n),IAOQ_Front{30..31});
if (n && sfu_condition1(cat(sop,sfu,n),IAOQ_Front{30..31}))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

Notes: The SPECIAL OPERATION ONE instruction is used to implement the IDENTIFY SFU pseudo-operation. This operation returns a 32-bit identification number from the special function unit *sfu* to general register *t*. The value returned is implementation dependent and is useful for configuration, diagnostics, and error recovery. The state of the SFU is undefined after this instruction.

Each implementation must choose an identification number that identifies the version of the SFU. The values all zeros and all ones are reserved. The assist emulation trap handler returns zero when executing this instruction. An assist exception trap is not allowed and this instruction must be implemented by all SFUs. The IDENTIFY SFU pseudo-operation is coded as: SPOP1,sfu,0 t

SPECIAL OPERATION TWO SPOP2

Format: SPOP2,sfu,sop,n r

04	r	sop1	2	sfu	n	sop2
6	5	10	2	3	1	5

Purpose: To perform a parameterized special function unit operation.

Description: GR *r* is passed to the SFU identified by *sfu*. The SFU uses its internal state, the contents of the register, and the instruction fields supplied to it to compute a result. If nullification is specified, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

The *sop* field in the assembly language format is the concatenation of the *sop1* and *sop2* fields in the machine instruction, sop = cat(sop1,sop2).

Operation:

```
sfu_operation2(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r]);
if (n && sfu_condition2(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r]))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

SPECIAL OPERATION THREE SPOP3

Format: SPOP3,sfu,sop,n r1,r2

04	r2	r1	sop1	3	sfu	n	sop2
6	5	5	5	2	3	1	5

Purpose: To perform a parameterized special function unit operation.

Description: GR *r1* and GR *r2* are passed to the SFU identified by *sfu*. The SFU uses its internal state, the contents of the two registers, and the instruction fields supplied to it to compute a result. If nullification is specified, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

The *sop* field in the assembly language format is the concatenation of the *sop1* and *sop2* fields in the machine instruction, sop = cat(sop1,sop2).

Operation:

```
sfu_operation3(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r1],GR[r2]);
if (n && sfu_condition3(cat(sop1,sfu,n,sop2),
        IAOQ_Front{30..31},GR[r1],GR[r2]))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

COPROCESSOR OPERATION — COPR

Format: COPR,uid,sop,n

0C	sop1	uid	n	sop2
6	17	3	1	5

Purpose: To invoke a coprocessor unit operation.

Description: The coprocessor operation code *sop* (assembled from the *sop1* and *sop2* fields) is sent to the coprocessor identified by *uid* and the indicated operation is performed. If nullification is specified and the coprocessor condition is satisfied, the following instruction is nullified.

Operation:

```
sop ← cat(sop1,sop2);
coprocessor_op(uid,sop,n,IAOQ_Front{30..31});
if (n && coprocessor_condition(uid,sop,n))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap   Assist exception trap

Notes: The COPROCESSOR OPERATION instruction is used to implement the IDENTIFY COPROCESSOR pseudo-operation. This operation places a identification number from the coprocessor uid into coprocessor register 0. This value is implementation dependent and is useful for configuration, diagnostic, and error recovery.

Each implementation must choose an identification number that identifies the version of the coprocessor. The values all zeros and all ones are reserved. An assist exception trap is not allowed and this instruction must be implemented by all coprocessors with unit identifiers 4 through 7. Unit identifiers 0 and 2 have a uid-specific sequence to obtain the identification number.

The format of the identification number for the floating-point coprocessor is described in “Status Register” on page 6-9.

The IDENTIFY COPROCESSOR pseudo-operation is coded as follows: COPR,uid,0

COPROCESSOR LOAD WORD INDEXED — CLDWX

Format: CLDWX,uid,cmplt,cc x(s,b),t

09	b	x	s	u	0	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a word into a coprocessor register.

Description: The aligned word, at the effective address, is loaded into register *t* of the coprocessor identified by *uid*. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 2. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],2);          /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                            /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                            /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],2);
                break;
    default:    offset ← GR[b] + GR[x];                    /*u=0, m=0*/
}
send_to_copr(uid,t);
CPR[uid][t] ← mem_load(space,offset,0,31,cc);
```

Exceptions:

- Assist exception trap
- Data TLB miss fault/data page fault
- Data memory access rights trap
- Data memory protection ID trap
- Unaligned data reference trap
- Page reference trap
- Data debug trap
- Assist emulation trap

COPROCESSOR LOAD DOUBLEWORD INDEXED — CLDDX

Format: CLDDX,uid,cmplt,cc x(s,b),t

0B	b	x	s	u	0	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a double word into a coprocessor register.

Description: The aligned double word, at the effective address, is loaded into register *t* of the coprocessor identified by *uid*. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 3. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:      offset ← GR[b] + lshift(GR[x],3);          /*u=1, m=0*/
                 break;
    case M:      offset ← GR[b];                            /*u=0, m=1*/
                 GR[b] ← GR[b] + GR[x];
                 break;
    case SM:     offset ← GR[b];                            /*u=1, m=1*/
                 GR[b] ← GR[b] + lshift(GR[x],3);
                 break;
    default:     offset ← GR[b] + GR[x];                    /*u=0, m=0*/
}
send_to_copr(uid,t);
CPR[uid][t] ← mem_load(space,offset,0,63,cc);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap
Assist emulation trap

COPROCESSOR STORE WORD INDEXED CSTWX

Format: CSTWX,uid,cmplt,cc r,x(s,b)

09	b	x	s	u	0	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

Purpose: To store a word from a coprocessor register.

Description: Register *r*, of the coprocessor identified by *uid*, is stored in the aligned word at the effective address. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 2. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],2);          /*u=1, m=0*/
                break;
    case M:     offset ← GR[b]                             /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                            /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],2);
                break;
    default:    offset ← GR[b] + GR[x];                    /*u=0, m=0*/
}
send_to_copr(uid,r);
mem_store(space,offset,0,31,cc,CPR[uid][r])
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap
Assist emulation trap

COPROCESSOR STORE DOUBLEWORD INDEXED CSTDX

Format: CSTDX,uid,cmplt,cc r,x(s,b)

0B	b	x	s	u	0	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

Purpose: To store a double word from a coprocessor register.

Description: Register *r*, of the coprocessor identified by *uid*, is stored in the aligned double word at the effective address. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 3. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],3);          /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                            /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                            /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],3);
                break;
    default:    offset ← GR[b] + GR[x];                    /*u=0, m=0*/
}
send_to_copr(uid,r);
mem_store(space,offset,0,63,cc,CPR[uid][r]);
```

Exceptions:

Assist exception trap	Data memory break trap
Data TLB miss fault/data page fault	TLB dirty bit trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap
Unaligned data reference trap	Assist emulation trap

COPROCESSOR LOAD WORD SHORT — CLDWS

Format: CLDWS,uid,cmplt,cc d(s,b),t

09	b	im5	s	a	1	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a word into a coprocessor register.

Description: The aligned word is loaded, from the effective address, into register *t* of the coprocessor identified by *uid*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);       /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                             /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);       /*m=0*/
}
send_to_copr(uid,t);
CPR[uid][t] ← mem_load(space,offset,0,31,cc);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap
Assist emulation trap

COPROCESSOR LOAD DOUBLEWORD SHORT CLDDS

Format: CLDDS,uid,cmplt,cc i(s,b),t

0B	b	im5	s	a	1	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a double word into a coprocessor register.

Description: The aligned double word is loaded, from the effective address, into register *t* of the coprocessor identified by *uid*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                              /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);        /*m=0*/
}
send_to_copr(uid,t);
CPR[uid][t] ← mem_load(space,offset,0,63,cc);
```

Exceptions:

Assist exception trap	Unaligned data reference trap
Data TLB miss fault/data page fault	Page reference trap
Data memory access rights trap	Data debug trap
Data memory protection ID trap	Assist emulation trap

COPROCESSOR STORE WORD SHORT — CSTWS

Format: CSTWS,uid,cmplt,cc r,d(s,b)

09	b	im5	s	a	1	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

Purpose: To store a word from a coprocessor register.

Description: Register *r*, of the coprocessor identified by *uid*, is stored in the aligned word at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:   offset ← GR[b] + low_sign_ext(im5,5);      /*a=1, m=1*/
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    case MA:   offset ← GR[b];                            /*a=0, m=1*/
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    default:   offset ← GR[b] + low_sign_ext(im5,5);      /*m=0*/
}
send_to_copr(uid,r);
mem_store(space,offset,0,31,cc,CPR[uid][r]);
```

Exceptions:

Assist exception trap	Data memory break trap
Data TLB miss fault/data page fault	TLB dirty bit trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap
Unaligned data reference trap	Assist emulation trap

COPROCESSOR STORE DOUBLEWORD SHORT — CSTDS

Format: CSTDS,uid,cmplt,cc r,d(s,b)

0B	b	im5	s	a	1	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

Purpose: To store a double word from a coprocessor register.

Description: Register *r*, of the coprocessor identified by *uid*, is stored in the aligned double word at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                                 /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);           /*m=0*/
}
send_to_copr(uid,r);
mem_store(space,offset,0,31,cc,CPR[uid][r]);
```

Exceptions:

Assist exception trap	Data memory break trap
Data TLB miss fault/data page fault	TLB dirty bit trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap
Unaligned data reference trap	Assist emulation trap

6 Floating-point Coprocessor

Introduction

This chapter describes the architecture for the floating-point coprocessor.

The PA-RISC floating-point coprocessor is an assist processor that is added to a system to improve the system's performance on floating-point operations. The floating-point coprocessor contains a register file which is independent of the processor's register file. The floating-point coprocessor executes floating-point instructions to perform arithmetic on this register file and to move data between the register file and memory. The architecture permits pipelined execution of floating-point instructions, further increasing the system's performance.

Floating-point instructions are implementations of the more general coprocessor instructions described previously in Chapter 5, "Instruction Set". The floating-point coprocessor responds to coprocessor instructions with a coprocessor id equal to 0 and 1.

While the floating-point coprocessor is not required to execute instructions sequentially, the coprocessor and processor must ensure that the instructions appear sequentially executed to the software. At any one time, the processor and coprocessor may be operating on a number of instructions. For purposes of this chapter, the **current instruction** is the instruction pointed to by the IA queues. The term **pending instructions** refers to instructions which have entered and left the IA queues, but which the coprocessor is still executing.

The IEEE Standard

When used in this chapter, the term **IEEE standard** or simply **the standard**, refers to the *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*. PA-RISC fully conforms to the requirements of the IEEE floating-point standard and permits implementation of all IEEE floating-point recommendations. Where hardware is unable to fully implement the standard, software completes the implementation.

Though this chapter uses quotes from the IEEE standard as architecture, knowledge of the standard is not necessary to understand the architecture. Whenever a quote of the standard contains a reference to another part of the standard, the quote also contains an equivalent reference to a section of this document. In these quotes, a reference to the IEEE standard is enclosed in parentheses, and the equivalent reference to this document is enclosed in square brackets.

The Instruction Set

The floating-point instruction set consists of load and store instructions, and operations. Floating-point load and store instructions copy both single-word and double-word data between memory and the floating-point registers. Floating-point operations do arithmetic on the floating-point registers and copy data between floating-point registers.

The floating-point coprocessor operates on single-word and double-word IEEE floating-point numbers, as well as quad-word numbers, which are an implementation of the IEEE double-extended format. Each type of floating-point number may represent one of the following: a normalized number, a denormalized number, a zero, an infinity, or a NaN (Not a Number). These floating-point formats consist of a sign bit, an exponent, and a fraction.

The instruction set also has operations that convert among the three formats of floating-point numbers and between floating-point numbers and single-word, double-word, and quad-word two's complement integers, as well as an instruction which multiplies two 32-bit unsigned integers with a 64-bit unsigned integer result.

Coprocessor Registers

The coprocessor contains thirty-two 64-bit floating-point registers. These same 32 locations can be used as sixty-four 32-bit locations or as sixteen 128-bit locations. Instructions executing at any privilege level may read or write the floating-point registers. Double-word load/store operations access the entire 64-bit register; single-word load/stores access either the left portion of a 64-bit register, bits 0 to 31, or the right portion of a 64-bit register, bits 32 to 63.

By convention, a 32-bit floating-point register is identified by appending a suffix to the identifier of the 64-bit register within which it is contained. The suffix for the left hand side 32-bit register is 'L'; the use of this suffix is optional. The suffix for the right hand side 32-bit register is 'R'; its use is **not** optional. Thus, for example, the left half of double-word register 13 (bits 0 to 31) would be referred to as either 13 or 13L; the right half of double-word register 13 (bits 32 to 63) would be referred to as 13R. The specification 'L' or 'R' for each register is encoded in the instructions that access these registers.

Table 6-1 illustrates the specification of single-word registers and Table 6-2 illustrates the specification of double-word registers.

Table 6-1. Single-Word Floating-Point Registers

Register	Purpose	
0	Status Register	Undefined
1	Undefined	Undefined
2	Undefined	Undefined
3	Undefined	Undefined
4	Floating-point register 4L	Floating-point register 4R
5	Floating-point register 5L	Floating-point register 5R
6	Floating-point register 6L	Floating-point register 6R
7	Floating-point register 7L	Floating-point register 7R
8	Floating-point register 8L	Floating-point register 8R
9	Floating-point register 9L	Floating-point register 9R
10	Floating-point register 10L	Floating-point register 10R
11	Floating-point register 11L	Floating-point register 11R
12	Floating-point register 12L	Floating-point register 12R
13	Floating-point register 13L	Floating-point register 13R
14	Floating-point register 14L	Floating-point register 14R
15	Floating-point register 15L	Floating-point register 15R
16	Floating-point register 16L	Floating-point register 16R
17	Floating-point register 17L	Floating-point register 17R
18	Floating-point register 18L	Floating-point register 18R
19	Floating-point register 19L	Floating-point register 19R
20	Floating-point register 20L	Floating-point register 20R
21	Floating-point register 21L	Floating-point register 21R
22	Floating-point register 22L	Floating-point register 22R
23	Floating-point register 23L	Floating-point register 23R
24	Floating-point register 24L	Floating-point register 24R
25	Floating-point register 25L	Floating-point register 25R
26	Floating-point register 26L	Floating-point register 26R
27	Floating-point register 27L	Floating-point register 27R
28	Floating-point register 28L	Floating-point register 28R
29	Floating-point register 29L	Floating-point register 29R
30	Floating-point register 30L	Floating-point register 30R
31	Floating-point register 31L	Floating-point register 31R

Table 6-2. Double-Word Floating-Point Registers

Register	Purpose	
0	Status Register	Exception Register 1
1	Exception Register 2	Exception Register 3
2	Exception Register 4	Exception Register 5
3	Exception Register 6	Exception Register 7
4	Floating-point register 4	
5	Floating-point register 5	
6	Floating-point register 6	
7	Floating-point register 7	
8	Floating-point register 8	
9	Floating-point register 9	
10	Floating-point register 10	
11	Floating-point register 11	
12	Floating-point register 12	
13	Floating-point register 13	
14	Floating-point register 14	
15	Floating-point register 15	
16	Floating-point register 16	
17	Floating-point register 17	
18	Floating-point register 18	
19	Floating-point register 19	
20	Floating-point register 20	
21	Floating-point register 21	
22	Floating-point register 22	
23	Floating-point register 23	
24	Floating-point register 24	
25	Floating-point register 25	
26	Floating-point register 26	
27	Floating-point register 27	
28	Floating-point register 28	
29	Floating-point register 29	
30	Floating-point register 30	
31	Floating-point register 31	

Single-word register 0 contains the Status Register. Double-word registers 0 - 3 contain the Status Register and the Exception Registers. Double-word registers 4 - 31 and single-word registers 4R - 31R and 4L - 31L are data registers.

Registers 0 - 3 are partitioned into eight 32-bit registers. Bits 0 to 31 of double-word register 0 contain the Status Register, which holds information on rounding, compares, and exceptions. Bits 32 to 63 of double-word register 0 contain Exception Register 1. Specifying Floating-point Register 0 in a non-load/store operation encodes a floating-point +0 or a fixed-point 0, whichever is appropriate, when used as a source and is an undefined operation when used as a destination.

Double-word registers 1 to 3 contain the remaining exception registers. The exception registers form a queue of instructions which could not normally complete and thus complete with a trapping exception. The exception registers are accessed using double-word load and store instructions. Single-word loads and stores of exception registers are undefined operations. Specifying an exception register as a source or destination of a non-load/store operation is undefined.

A special instruction sequence saves and restores the state of the coprocessor. This sequence ensures that context switches and other operations which affect the state of the coprocessor do not affect a process.

Exceptions

Floating-point instructions may cause an interruption in the processor, an exception in the coprocessor, or both. Interruptions are described in Chapter 4, “Flow Control and Interruptions” and always force the processor to branch to a location in the Interruption Vector Table. Floating-point coprocessor exceptions may or may not force the processor to trap (that is, force the processor to take an interruption). In this chapter, an instruction which causes a floating-point exception is called an **excepting instruction**.

Floating-point exceptions are divided into immediate trapping exceptions and delayed trapping exceptions. Immediate trapping exceptions always force the processor to trap. Delayed trapping exceptions are further divided into exceptions that always trap, and exceptions that will trap only when the corresponding trap is enabled. An immediate trapping exception forces the processor to signal an assist exception trap when the excepting instruction is the current instruction being executed. A delayed trapping exception forces the processor to signal an assist exception trap when the current instruction is a floating-point instruction, but the excepting instruction is a pending instruction.

The only immediate trapping exception is the reserved-op exception. This exception cannot be disabled.

The only delayed trapping exception that cannot be disabled is the unimplemented exception. The other delayed trapping exceptions are the IEEE exceptions. Each has a corresponding bit in the Status Register which enables and disables the delayed trap. The IEEE exceptions are the following: invalid operation, division-by-zero, overflow, underflow, and inexact.

Data Registers

Floating-point registers 4 - 31 contain the 64-bit data registers which instructions use as operands. Software may access these registers with single-word or double-word load and store instructions.

Each of the floating-point data registers may contain values in a number of formats. The fields in these formats are packed into words, double-words, or quad-words so that load and store operations do not require field-shuffling or tag bits.

Single-word formats occupy either the left half (bits 0 to 31, suffix ’L’), or the right half (bits 32 to 63, suffix ’R’) of a register as shown in Figure 6-1.

0 31	32 63
single-word number (suffix L)	single-word number (suffix R)
32	32

Figure 6-1. Single-word Data Format

Double-word formats fill one register as shown in Figure 6-2.

double-word format
64

Figure 6-2. Double-word Data Format

Quad-word formats (128 bits) are packed into adjacent even-odd pairs of registers. An instruction which references a register containing a quad-word value must name an even numbered register. An operation which specifies an odd numbered register for a quad-word format is an undefined operation. Quad formats are assembled in register pairs as shown in Figure 6-3.

even # register	most significant bits
odd # register	least significant bits
	64

Figure 6-3. Quad-word Data Format

Data Formats

Two data types are defined: floating-point formats and fixed-point formats.

Floating-point Formats

Numbers in the single, double, and quad binary floating-point formats are composed of three fields:

1. A 1-bit sign, *s*.
2. A biased exponent, $e = E + bias$.
3. A fraction, $f = .b_1 b_2 \ldots b_{p-1}$.

Note that adding the *bias* to the unbiased exponent *E* produces the biased exponent *e*. The number *e* is always non-negative. Also, *p* is the precision of the number, and is equal to one plus the number of fraction bits. Figure 6-4 shows the positions of these fields in the registers.

Single Binary Floating-point

s	e	fraction
1	8	23

Double Binary Floating-point

s	e	fraction
1	11	52

Quad Binary Floating-point

s	e	high fraction
1	15	48

low fraction
64

Figure 6-4. Floating-point Formats

For each floating-point format, a number may either be a normalized number, a denormalized number, an infinity, a zero, or a NaN (Not a Number). Each representable nonzero numerical value has just one encoding. Use the format parameters listed in Table 6-3 and the equations which follow to determine the representation and value, *v*, of a floating-point number.

Zero:	If $E = E_{min} - 1$ and $f = 0$, then $v = (-1)^s 0$.
Denormalized:	If $E = E_{min} - 1$ and $f \neq 0$, then $v = (-1)^s 2^{E_{min}} (0.f)$.
Normalized:	If $E_{min} \leq E \leq E_{max}$, then $v = (-1)^s 2^E (1.f)$.
Infinity:	If $E = E_{max} + 1$ and $f = 0$, then $v = (-1)^s \infty$.
NaN:	If $E = E_{max} + 1$ and $f \neq 0$, then *v* is a NaN, regardless of *s*.

If the number is a NaN, then the leftmost bit in the fraction, b_1, determines whether the NaN is signaling or quiet. If b_1 is 1, the NaN is a signaling NaN. If b_1 is 0, it is a quiet NaN.

Table 6-3. Floating-Point Format Parameters

Parameter	Format		
	Single	Double	Quad
p (precision)	24	53	113
E_{max}	+127	+1023	+16383
E_{min}	−126	−1022	−16382
exponent *bias*	+127	+1023	+16383
under/overflow *bias*-adjustment	192	1536	24576
exponent width in bits	8	11	15
format width in bits	32	64	128

Table 6-4 shows the hexadecimal ranges of floating-point numbers.

Table 6-4. Hexadecimal Ranges of Floating-Point Representations

Type	Sign	Single	Double	Quad
Signaling NaN	none	`7FFFFFFF –` `7FC00000`	`7FFFFFFF FFFFFFFF –` `7FF80000 00000000`	`7FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF –` `7FF80000 00000000 00000000 00000000`
Quiet NaN		`7FBFFFFF –` `7F800001`	`7FF7FFFF FFFFFFFF –` `7FF00000 00000001`	`7FFF7FFF FFFFFFFF FFFFFFFF FFFFFFFF –` `7FFF0000 00000000 00000000 00000001`
Infinity	+	`7F800000`	`7FF00000 00000000`	`7FFF0000 00000000 00000000 00000000`
Normalized		`7F7FFFFF –` `00800000`	`7FEFFFFF FFFFFFFF –` `00100000 00000000`	`7FFEFFFF FFFFFFFF FFFFFFFF FFFFFFFF –` `00010000 00000000 00000000 00000000`
Denorm		`007FFFFF –` `00000001`	`000FFFFF FFFFFFFF –` `00000000 00000001`	`0000FFFF FFFFFFFF FFFFFFFF FFFFFFFF –` `00000000 00000000 00000000 00000001`
Zero		`00000000`	`00000000 00000000`	`00000000 00000000 00000000 00000000`
Zero	–	`80000000`	`80000000 00000000`	`80000000 00000000 00000000 00000000`
Denorm		`80000001 –` `807FFFFF`	`80000000 00000001 –` `800FFFFF FFFFFFFF`	`80000000 00000000 00000000 00000001 –` `8000FFFF FFFFFFFF FFFFFFFF FFFFFFFF`
Normalized		`80800000 –` `FF7FFFFF`	`80100000 00000000 –` `FFEFFFFF FFFFFFFF`	`80010000 00000000 00000000 00000000 –` `FFFEFFFF FFFFFFFF FFFFFFFF FFFFFFFF`
Infinity		`FF800000`	`FFF00000 00000000`	`FFFF0000 00000000 00000000 00000000`
Quiet NaN	none	`FF800001 –` `FFBFFFFF`	`FFF00000 00000001 –` `FFF7FFFF FFFFFFFF`	`FFFF0000 00000000 00000000 00000001 –` `FFFF7FFF FFFFFFFF FFFFFFFF FFFFFFFF`
Signaling NaN		`FFC00000 –` `FFFFFFFF`	`FFF80000 00000000 –` `FFFFFFFF FFFFFFFF`	`FFFF8000 00000000 00000000 00000000 –` `FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF`

NaNs are not ordered; neither the fraction nor the sign bits have any significance.

Fixed-Point Formats

Fixed-point values are held in the formats shown in Figure 6-5.

Single Binary Fixed-point

two's complement integer
32

unsigned integer
32

Double Binary Fixed-point

two's complement integer
64

unsigned integer
64

Quad Binary Fixed-point

high two's complement integer
low two's complement integer
64

Figure 6-5. Fixed-point Formats

Status Register

The Status Register controls the arithmetic rounding mode, enables user-level traps, indicates exceptions that have occurred, indicates the results of comparisons, and contains information to identify the implementation of the coprocessor. The Status Register is located in bits 0 to 31 of Floating-point Register 0, and is accessed by specifying Floating-point Register 0 with single-word or double-word load and store instructions.

Non-load/store instructions do not access the Status Register. Specifying Floating-point Register 0 in a non-load/store operation, encodes a floating-point +0 or a fixed-point 0, whichever is appropriate, when used as a source and is an undefined operation when used as a destination.

Figure 6-6 shows the two formats of the Status Register. The first format is valid most of the time. The second format is only valid immediately after the execution of a FLOATING-POINT IDENTIFY instruction. This format remains valid until a floating-point instruction is executed which is not a double-word store of Floating-point Register 0. The first format is valid thereafter. See the description of the FLOATING-POINT IDENTIFY instruction on page 6-63 for more information.

0	1	2	3	4	5	6–9	10–20	21–22	23–24	25	26	27	28	29	30	31
V	Z	O	U	I	C	un	CQ	RM	un	T	D	V	Z	O	U	I

Flags (bits 0–4) Enables (bits 27–31)

0–9	10–15	16–20	21–31
un	model	revision	un

Figure 6-6. Status Register

Field	**Description**

RM The rounding mode for all floating-point operations. The values corresponding to each rounding mode are listed in Table 6-5.

Table 6-5. Rounding Modes

Rounding mode	Description
0	Round to nearest
1	Round toward zero
2	Round toward $+\infty$
3	Round toward $-\infty$

Enables The exception trap enables. An enable bit is associated with each IEEE exception. When an enable bit equals 1, the corresponding trap is enabled. An instruction completes with a delayed trap when the instruction causes an exception whose corresponding enable bit equals 1. If an enable bit equals 0, the corresponding IEEE exception usually sets the corresponding exception flag to 1 instead of causing a trap. However, see “Unimplemented Exception” on page 6-30, “Overflow Exception” on page 6-32, and “Underflow Exception” on page 6-33 for cases when a trap is taken even when the trap is not enabled. Table 6-6 lists the bits that correspond to each IEEE exception.

Flags The exception flags. A flag bit is associated with each IEEE exception. The coprocessor sets an exception flag to 1 when the corresponding exception occurs but does not cause a trap. An implementation may also choose to set a flag bit to 1 when the corresponding exception occurs and causes a trap. An exception flag is never set to 0 as a side effect of floating-point operations, but it may be set to either 1 or 0 by a load instruction. Table 6-6 lists the bits that correspond to each IEEE exception.

Table 6-6. IEEE Exceptions

Bit Name	Description
V	Invalid operation
Z	Division-by-zero
O	Overflow
U	Underflow
I	Inexact result

C The Compare bit. The C-bit contains the result of the most recent compare instruction. This bit is set to 1 if the result of the most recent compare is true, and 0 if false. No other non-load/store instruction affects this bit, but it may be set to 1 or 0 by load instructions.

CQ The Compare Queue. The CQ field contains the results of the second-most recent compare (in CQ{0}) through the twelfth-most recent compare (in CQ{10}). Every compare instruction shifts the CQ field right by one bit (discarding the rightmost bit) and the C-bit from the previous compare is copied into CQ{0}. No other non-load/store instruction affects this field, but it may be set to any value by load instructions.

T The Delayed Trap bit. The coprocessor sets this bit to 1 when it takes a trap on an unimplemented exception or an IEEE exception. When software sets this bit to 1, the coprocessor is armed to trap, and the next floating-point instruction forces the processor to take an assist exception trap. No non-load/store instructions affect this bit, but load instructions may set this bit to 1 or 0, and double-word stores of the Status Register set this bit to 0 after completion of the store. Also, save and restore software uses this bit to record the state of traps. See "Saving and Restoring State" on page 6-35 for information on state swapping and "Interruptions and Exceptions" on page 6-26 for a detailed discussion of the T-bit's operation.

D The Denormalized As Zero bit. The D-bit provides the arithmetic floating-point instructions a "fast mode" handling of denormalized operands and tiny results. When the D-bit is 1, any of these instructions may optionally produce a correctly signed zero when the result, before or after rounding, lies between $\pm 2^{E_{\min}}$, and may optionally treat any denormalized operand as an equivalently signed zero. When the D-bit is 1, and an arithmetic instruction treats an operand as a zero, or produces a zero result as described above, the values of the Underflow and Inexact flags become undefined. When the D-bit is 0, all denormalized operands and tiny results must be handled as described in the remainder of this chapter.

Implementation of the D-bit is optional; if not implemented, it is a nonexistent bit.

model The implementation-dependent model number. Model number zero (0) is reserved for the software emulation routines.

revision The implementation-dependent revision number.

un Undefined bits.

Instruction Set

The floating-point instruction set consists of load and store instructions, and floating-point operations. All these instructions are part of the PA-RISC standard instruction set. When the instruction specifies a register for double precision and 64-bit fixed-point values, a 5-bit encoding maps directly into the associated floating-point register. When the instruction specifies a register for single-precision or 32-bit fixed-point values, a 6-bit encoding maps into the appropriate ‘L’ or ‘R’ single-word floating-point register.

Instruction Validity

Table 6-7 shows which floating-point instructions are defined, undefined, or will take an Assist Emulation Trap for various values of the uid field and the Coprocessor Configuration Register.

Table 6-7. Floating-Point Instruction Validity

Opcode	CCR{0..1}		
	0	1, 2	3
0B/0C	trap	undefined	uid=0: defined uid=1: undefined
06/09/0E/26	trap	undefined	defined

Load and Store Instructions

The floating-point load and store instructions are implementations of the PA-RISC coprocessor load and store instructions described in “Coprocessor Instructions” on page 5-178. Table 6-8 shows the floating-point load and store instructions.

Table 6-8. Floating-Point Load and Store Instructions

Mnemonic	Description
FLDWX	Load word using index register
FLDDX	Load doubleword using index register
FSTWX	Store word using index register
FSTDX	Store doubleword using index register
FLDWS	Load word using short displacement
FLDDS	Load doubleword using short displacement
FSTWS	Store word using short displacement
FSTDS	Store doubleword using short displacement

The load and store instructions transfer data between the floating-point registers and memory. These instructions transfer aligned words or aligned double-words, and use either a short displacement value or the value of an index register. They also have completers that specify base register modification and cache control hints.

Single-word loads and stores access either bits 0 to 31 (suffix ‘L’) or bits 32 to 63 (suffix ‘R’) of a register. The ability to specify more than 32 locations is accomplished by the use of bit 25 of the instruction. A 0 in this bit specifies an access of bits 0 to 31 (left half of the register); a 1 specifies an access of bits 32 to 63 (the right half of the register). However, single-word loads and stores of floating-point registers 0R (the right half of floating-point register 0), 1, 2, and 3 are undefined operations.

Double-word loads and stores can access any of the floating-point registers and may be used to load or store a pair of single-word values in the left and right halves of a register.

A single-word or double-word load or store of Floating-point Register 0 forces the coprocessor to complete all pending floating-point instructions and signal all floating-point exceptions for those instructions. Additionally, a double-word store of Floating-point Register 0 cancels traps due to all previous instructions and, after completion of the store, sets the Status Register T-bit to 0.

Single-word stores of register 0 do not cancel traps. Also, single-word loads of register 0 that set the Status Register T-bit to 1 are undefined operations.

A double-word load or store of registers 1, 2 or 3 forces the floating-point coprocessor to complete all previous instructions that may update the specified exception register.

Load and store instructions may cause a number of memory reference traps. They are not arithmetic instructions and do not cause IEEE exceptions.

Load and store instructions that access the I/O address space are undefined operations.

Floating-point Operations

There are two categories of floating-point operation instructions. Each instruction in the first category performs a single operation. Instructions in the second category perform multiple operations.

Single-operation Instructions

This section describes the single-operation floating-point instructions. Single-operation floating-point instructions are encoded using two major opcodes - 0C and 0E. Most of the functions in the 0C opcode are also duplicated in the 0E opcode with the following exceptions:

- Instructions using the 0E opcode can address both the left and right halves of the floating-point register set whereas the instructions using the 0C opcode can only address the left halves of the floating-point register set.
- The COPR,0,0 and FTEST instructions are available only in the 0C opcode.
- The format completer for the quad-word data type cannot be specified for the 0E opcode.
- The XMPYU instruction is available only in the 0E opcode.

There are four classes of operations:

- Class 0 contains single source, single destination operations and includes the non-arithmetic operations.
- Class 1 consists of the conversion operations.

- Class 2 operations provide mechanisms to compare two operands.
- Class 3 consists of the arithmetic operations with two sources and one destination.

Figure 6-7 shows the format of these operations.

Floating-point operation class zero: 1 source, 1 destination*

0C	r	0	sub	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

0E	r	0	sub	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

Floating-point operation class one: 1 source, 1 destination

0C	r	0	sub	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

0E	r	0	sub	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

Floating-point operation class two: 2 sources, no destination*

0C	r1	r2	sub	fmt	2	0	n	c
6	5	5	3	2	2	3	1	5

0E	r1	r2	sub	r2	f	2	0	r1	0	0	c
6	5	5	3	1	1	2	1	1	1	1	5

Floating-point operation class three: 2 sources, 1 destination†

0C	r1	r2	sub	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

0E	r1	r2	sub	r2	f	3	x	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

* The FLOATING-POINT IDENTIFY and FLOATING-POINT TEST instructions have no source or destination operands, and no format specifiers, so the register and format fields equal 0.

† The FIXED-POINT MULTIPLY UNSIGNED instruction has no format specifier, so the format field equals 0.

Figure 6-7. Single-operation Instruction Formats

Whenever single-precision operands are specified for the 0E opcode, the *t* at bit position 25 of class zero, one, and three instructions represents a sixth bit of the *t* field, and the *r* or *r1* at bit position 24 represents a sixth bit of the *r* or *r1* field. Similarly, the *r2* at bit position 19 represents a sixth bit of the *r2* field. These bits specify the left side single-word register, bits 0 to 31, when 0, and the right side single-word register, bits 32 to 63, when 1. The *x* at bit position 23 of a class three instruction indicates, when 1, that the sub-opcode is to be interpreted as a fixed-point operation.

Table 6-9 shows the floating-point operations, their mnemonics, classes, and sub-opcodes.

Table 6-9. Floating-Point Operations

Opcode	Sub-op	Class	Mnemonic	Operation
0C	0	0	COPR,0,0	Identify coprocessor
0E	0			undefined
0C/0E	1			undefined
0C/0E	2		FCPY	Copy
0C/0E	3		FABS	Absolute value
0C/0E	4		FSQRT	Square root
0C/0E	5		FRND	Round to integer
0C/0E	6-7			reserved
0C/0E	0	1	FCNVFF	Convert from floating-point to floating-point
0C/0E	1		FCNVXF	Convert from fixed-point to floating-point
0C/0E	2		FCNVFX	Convert from floating-point to fixed-point
0C/0E	3		FCNVFXT	Convert from floating-point to fixed-point with explicit round to zero rounding
0C/0E	0	2	FCMP	Arithmetic compare
0C	1		FTEST	Test condition bit
0E	1			undefined
0C/0E	2-7			undefined
0C/0E	0	3	FADD	Add
0C/0E	1		FSUB	Subtract
0C/0E	2		FMPY	Multiply ($x = 0$)
0C/0E	3		FDIV	Divide
0C	4			reserved
0E	4			undefined
0C/0E	5-6			reserved
0C/0E	7			undefined

While the coprocessor may simultaneously operate on more than one instruction, the coprocessor is restricted by the number of exception registers to executing no more than seven floating-point operations at one time.

Except for the FLOATING-POINT COPY and FLOATING-POINT ABSOLUTE VALUE instructions, all the operations which have at least one floating-point operand are considered arithmetic instructions and will generate an invalid exception when operating on a signaling NaN.

The FLOATING-POINT IDENTIFY and FLOATING-POINT TEST instructions do not cause exceptions.

Table 6-10 shows the only fixed-point operation, its mnemonic, class and sub-opcode.

Table 6-10. Fixed-Point Operations

Opcode	Sub-op	x-bit	Class	Mnemonic	Operation
0E	2	1	3	XMPYU	Fixed-point Multiply Unsigned

Operand Format Completers

For class 0, 2 and 3 operations, except for FIXED-POINT MULTIPLY UNSIGNED, the source and destination widths are the same and the instructions operate only on floating-point numbers. Except for the FLOATING-POINT IDENTIFY, FLOATING-POINT TEST, and FIXED-POINT MULTIPLY UNSIGNED operations, each has an accompanying completer which specifies the data width the operation is using.

The operations in class 1 (the conversion instructions) have two completers which specify the source and destination widths independently. However, the floating-point to floating-point conversions single-to-single and double-to-double in the 0C and 0E opcodes are undefined operations; In addition, in the 0C opcode, quad-to-quad floating-point conversion is an undefined operation (quad precision cannot be specified in the 0E opcode).

Table 6-11 indicates the instruction completers and their corresponding format codes for the 0C opcode.

Table 6-11. Floating-Point Operand Format Completers (0C opcode)

Opcode	Mnemonic	Code	Description
0C	<none>	00	single-word number (32 bits)
0C	SGL	00	single-word number (32 bits)
0C	DBL	01	double-word number (64 bits)
0C	QUAD	11	quad-word number (128 bits)
0C		10	undefined

The *Code* field above indicates the two-bit encoding corresponding to each mnemonic. The absence of a completer specifies a single-word number. An operation with a *Code* value of 10 is an undefined operation.

Table 6-12 indicates the instruction completers and their corresponding format codes for the 0E opcode.

Table 6-12. Floating-Point Operand Format Completers (0E opcode)

Opcode	Mnemonic	Code	Description
0E	<none>	r0	single-word number (32 bits)
0E	SGL	r0	single-word number (32 bits)
0E	DBL	01	double-word number (64 bits)
0E		11	undefined

The *Code* field above indicates the two-bit encoding corresponding to each mnemonic. The absence of a completer specifies a single-word number. For the single-word format, the left bit of the *Code* value is used as the sixth, low-order bit of the *r2* specifier for operations in classes 2 and 3, and is zero in classes 0 and 1. An operation with a *Code* value of 11 is a undefined operation.

Comparison Conditions

The FLOATING-POINT COMPARE instruction has an additional completer which indicates the condition being tested. These conditions are listed in Table 6-13 which is derived from the IEEE standard.

Table 6-13. Floating-Point Compare Conditions

Condition	Relations				Code	Condition	Relations				Code
	>	<	=	unordered			>	<	=	unordered	
false?	F	F	F	F	0	!?<=	T	F	F	F	16
false	F	F	F	F *	1	>	T	F	F	F *	17
?	F	F	F	T	2	?>	T	F	F	T	18
!<=>	F	F	F	T *	3	!<=	T	F	F	T *	19
=	F	F	T	F	4	!?<	T	F	T	F	20
=T	F	F	T	F *	5	>=	T	F	T	F *	21
?=	F	F	T	T	6	?>=	T	F	T	T	22
!<>	F	F	T	T *	7	!<	T	F	T	T *	23
!?>=	F	T	F	F	8	!?=	T	T	F	F	24
<	F	T	F	F *	9	<>	T	T	F	F *	25
?<	F	T	F	T	10	!=	T	T	F	T	26
!>=	F	T	F	T *	11	!=T	T	T	F	T *	27
!?>	F	T	T	F	12	!?	T	T	T	F	28
<=	F	T	T	F *	13	<=>	T	T	T	F *	29
?<=	F	T	T	T	14	true?	T	T	T	T	30
!>	F	T	T	T *	15	true	T	T	T	T *	31

Comparisons are exact and neither overflow or underflow. Between any two operands, one of four mutually exclusive relations is possible: **less than**, **equal**, **greater than**, and **unordered**. The last case arises when at least one operand is a NaN. Every NaN compares unordered with every operand, including itself. Comparisons ignore the sign of zero, so +0 is equal to -0.

In the table above, *Condition* is the condition mnemonic used in the assembly language and *Code* is the machine language encoding. The floating-point coprocessor sets the Status Register C-bit to the result indicated in the appropriate relations column, 1 for a true result, 0 for false. The asterisk (*) indicates that the instruction causes an invalid operation exception if its operands are unordered. However, if at least one operand is a signaling NaN, the compare instruction always causes an invalid operation exception.

Test Conditions

The FLOATING-POINT TEST instruction has an additional completer which indicates the condition being tested. These conditions are listed in Table 6-14.

Table 6-14. Floating-Point Test Conditions

<table>
<tr><th>Completer</th><th>Description</th><th>Condition</th><th>Code</th></tr>
<tr><td><none></td><td>Simple Test</td><td>C == 1</td><td>0</td></tr>
<tr><td>ACC</td><td>Graphics (12-bit) Trivial Accept</td><td>C == 0 && CQ{0..10} == 0</td><td>1</td></tr>
<tr><td>ACC8</td><td>Graphics 8-bit Trivial Accept</td><td>C == 0 && CQ{0..6} == 0</td><td>5</td></tr>
<tr><td>ACC6</td><td>Graphics 6-bit Trivial Accept</td><td>C == 0 && CQ{0..4} == 0</td><td>9</td></tr>
<tr><td>ACC4</td><td>Graphics 4-bit Trivial Accept</td><td>C == 0 && CQ{0..2} == 0</td><td>13</td></tr>
<tr><td>ACC2</td><td>Graphics 2-bit Trivial Accept</td><td>C == 0 && CQ{0} == 0</td><td>17</td></tr>
<tr><td>REJ</td><td>Graphics (12-bit) Trivial Reject</td><td>C == 1 && CQ{5} == 1 ||
CQ{0} == 1 && CQ{6} == 1 ||
CQ{1} == 1 && CQ{7} == 1 ||
CQ{2} == 1 && CQ{8} == 1 ||
CQ{3} == 1 && CQ{9} == 1 ||
CQ{4} == 1 && CQ{10} == 1</td><td>2</td></tr>
<tr><td>REJ8</td><td>Graphics 8-bit Trivial Reject</td><td>C == 1 && CQ{3} == 1 ||
CQ{0} == 1 && CQ{4} == 1 ||
CQ{1} == 1 && CQ{5} == 1 ||
CQ{2} == 1 && CQ{6} == 1</td><td>6</td></tr>
</table>

Multiple-Operation Instructions

The floating-point instruction set includes instructions which perform more than one floating-point operation. These instructions are encoded using the 06 and 26 opcodes. Multiple-operation instructions are five-operand instructions which combine a three-operand multiply with a two-operand operation (ADD or SUB) of the form: dest ← dest <op> source.

The format of the multiple-operation instructions is as follows:

op	rm1	rm2	ta	ra	f	tm
6	5	5	5	5	1	5

Figure 6-8. Multiple-Operation Instruction Format

The *rm1*, *rm2*, and *tm* fields specify the two source operands and the destination operand for the multiply operation. These fields occupy the same positions within the instruction word as the operands of a class 3 single-operation floating-point instruction. The *ra* and *ta* fields specify source and destination operands for the alu operation.

The behavior of the multiple-operation instructions is undefined if *ra* specifies the same register as *tm*, or if *ta* specifies the same register as any of *rm1*, *rm2*, or *tm*. The behavior of these instructions is also undefined if *ra* specifies double-precision register 0 or single-precision register 16L.

Table 6-15 lists the different multiple-operation instructions, their mnemonics and opcodes.

Table 6-15. Multiple-Operation Instructions

Opcode	Mnemonic	Operation
06	FMPYADD	Multiply/Add
26	FMPYSUB	Multiply/Subtract

The *f* field in the floating-point multiple-operation instructions is an operand format completer, similar to the *fmt* field of the single-operation instructions. Only single-word and double-word formats are supported. The interpretation of the format completer is given in Table 6-16.

Table 6-16. Multiple-Operation Instruction Format Completers

Mnemonic	Code	Description	
<none>	1	single-word number	(32 bits)
SGL	1	single-word number	(32 bits)
DBL	0	double-word number	(64 bits)

NOTE

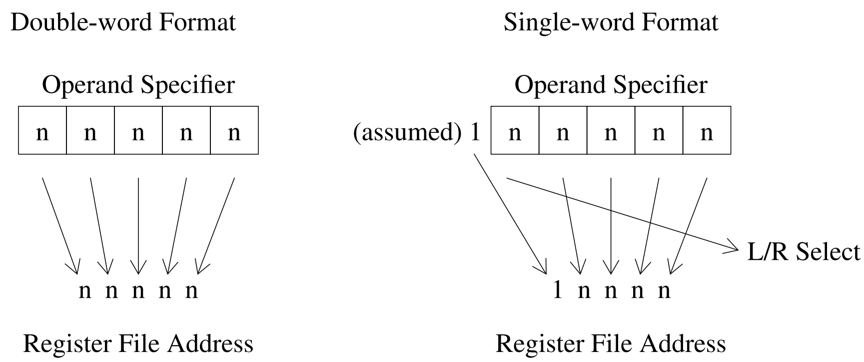
Note that the instruction format completers for the multiple-operation instructions do not follow the same pattern as those for the single-operation instructions.

Because the floating-point multiple-operation instructions have only five-bit operand specifiers, these instructions operate on only 32 locations, even when the single-word data format is specified. When double-word data is specified, the interpretation of these operand specifiers is the same as for the single-operation instructions. For single-word data, however, the operand specifiers are restricted to the top 16 registers (32 locations). The details of the interpretation of the operand specifier field in the instruction are shown in Table 6-17.

Table 6-17. Single-Precision Operand Specifier Use in Multi-Operation Instructions

Register specifier field in instruction	Register selected
0	16L
16	16R
1	17L
17	17R
2	18L
18	18R
3	19L
19	19R
4	20L
20	20R
5	21L
21	21R
6	22L
22	22R
7	23L
23	23R
8	24L
24	24R
9	25L
25	25R
10	26L
26	26R
11	27L
27	27R
12	28L
28	28R
13	29L
29	29R
14	30L
30	30R
15	31L
31	31R

NOTE



The L/R Select bit in the single-word format specifies the suffix ‘L’ single-word register, bits 0 to 31, when 0, and the suffix ‘R’ single-word register, bits 32 to 63, when 1.

Rounding

The specification for the rounding operation from the IEEE standard is:

> Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination’s format while signaling the inexact exception (7.5) [see “Inexact Exception” on page 6-32]. Except for binary-decimal conversion (whose weaker conditions are specified in 5.6 [not included]), every operation specified in Section 5 [see “Floating-point Operations” on page 6-13] shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the modes in this section.
>
> The rounding modes affect all arithmetic operations except comparison. The rounding modes may affect the signs of zero sums (6.3) [see “Sign Bit” on page 6-23], and do affect the thresholds beyond which overflow (7.3) [see “Overflow Exception” on page 6-32] and underflow (7.4) [see “Underflow Exception” on page 6-33] may be signaled.
>
> [§]**4.1 Round to nearest.** An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least-significant bit zero shall be delivered. However, an infinitely precise result with magnitude at least $2^{E_{max}}\left(2-2^{-p}\right)$ shall round to ∞ with no change in sign; here E_{max} and p are determined by the destination format (§3) [see “Floating-point Formats” on page 6-6] unless overridden by a rounding precision mode (4.3) [not possible in PA-RISC].
>
> [§]**4.2 Directed Roundings.** An implementation shall also provide three user-selectable directed rounding modes: round toward +∞, round toward −∞, and round toward 0. When rounding toward +∞, the result shall be the format’s value (possibly +∞) closest to and no less than the infinitely precise result. When rounding toward −∞, the result shall be the format’s

value (possibly $-\infty$) closest to and no greater than the infinitely precise result. When rounding toward 0, the result shall be the format's value closest to and no greater in magnitude than the infinitely precise result.

The *RM* field in the Status Register determines the rounding mode.

While the above IEEE quote describes the process of rounding, an operation does not always return a rounded result. The result of an operation may be affected if the operation causes an exception.

Infinity Arithmetic

From the standard:

[§]**6.1 Infinity Arithmetic.** Infinity arithmetic shall be construed as the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is:

$$-\infty < \text{(every finite number)} < +\infty$$

Arithmetic with an infinite operand is always exact and can only signal invalid and unimplemented exceptions. An infinite result is created from finite operands only by a non-trapping overflow exception or a non-trapping division-by-zero exception.

Operations With NaNs

From the standard:

[§]**6.2 Operations with NaNs.** Two different kinds of NaN, signaling and quiet, shall be supported in all operations. Signaling NaNs afford values for uninitialized variables and arithmetic-like enhancements (such as complex-affine infinities or extremely wide range) that are not the subject of the standard. Quiet NaNs should, by means left to the implementor's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the NaNs be preserved through arithmetic operations and floating-point format conversions.

An operation causes an invalid exception when at least one operand is a signaling NaN and the operation is any arithmetic operation except a convert to an integer format. Also, certain compare operations cause an invalid exception if an operand is a quiet NaN. See "Comparison Conditions" on page 6-17 for more detail.

Converting either a quiet or a signaling NaN to an integer format causes an unimplemented exception.

A NaN is created in two ways. Any operation that causes a non-trapping invalid exception returns a quiet NaN. Otherwise, an operation returns a quiet NaN when at least one of its operands is a quiet NaN, and the operation is any arithmetic operation except a conversion to an integer format.

An operation converts a signaling NaN to a quiet NaN when the operation causes a non-trapping invalid exception and one of its operands is a signaling NaN. If both operands are signaling NaNs, the operation converts the contents of the first operand (the *r1* register).

An operation which converts a signaling NaN to a quiet NaN sets the first bit of the fraction (b_1) to 0. If the remaining bits in the fraction are all zeros ($b_2 \ldots b_{p-1} = 0$), the operation must set the second bit in the fraction to 1. Otherwise, if the remainder of the fraction is not 0, an implementation has the option of setting the second bit in the fraction to 1, or leaving it unchanged. Only the first and second bits in the fraction may change when creating a quiet NaN from a signaling NaN. The remaining fraction bits are copied from the signaling NaN.

When one of its operands is a quiet NaN, but neither operand is a signaling NaN, an operation copies the quiet NaN to the destination. If both operands are quiet NaNs, the *r1* register is copied to the destination.

The creation of a quiet NaN when neither input is a NaN sets the second fraction bit (b_2) to 1 and sets each of the remaining fraction bits to 0.

A conversion operation which does not trap, and which converts a NaN to a smaller floating-point format, preserves the most-significant portion of the fraction while returning a quiet NaN. But if the most-significant portion of the fraction is all zeros, the second bit of the fraction must be set to 1 to prevent the number from becoming an infinity. A conversion which does not trap, and which converts a NaN to a larger floating-point format, augments the fraction with zeros to the right of the smaller fraction while returning a quiet NaN.

Load and store instructions, as well as the FLOATING-POINT COPY and FLOATING-POINT ABSOLUTE VALUE instructions, are not arithmetic and do not signal an invalid operation exception.

Sign Bit

From the standard:

> [§]**6.3 The sign bit.** This standard does not interpret the sign of a NaN. Otherwise the sign of a product or quotient is the exclusive OR of the operands' signs; and the sign of the sum, or of a difference $x - y$ regarded as a sum $x + (-y)$, differs from at most one of the addends' signs. These rules shall apply even when operands or results are zero or infinite.
>
> When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be "+" in all rounding modes except round toward –∞, in which mode the sign shall be "–". However, $x + x = x - (-x)$ retains the same sign as x even when x is zero.
>
> Except that $\sqrt{-0}$ shall be "–0", every valid square root shall have positive sign.

Exception Registers

The exception registers contain information on floating-point operations that have completed execution and have caused a delayed trapping exception. All the registers must be present and storage provided for loads and stores even if an implementation never uses a particular register to record exception state.

The exception registers are accessed with double-word load and store instructions. Single-word loads and stores of registers 0R, 1L, 1R, 2L, 2R, 3L, and 3R are undefined. Specifying an exception register as a source or destination of a non-load/store operation is undefined.

For single-operation instructions, an exception register contains a modified copy of an excepting instruction that traps. The coprocessor replaces the field that normally contains the instruction opcode with a code that indicates the type of exception detected. The remaining bits are duplicates from the original instruction.

The multiple-operation instructions (FLOATING-POINT MULTIPLY/ADD and FLOATING-POINT MULTIPLY/SUBTRACT) cannot directly cause trapping IEEE exceptions, because the exception state cannot be represented in the exception registers. When one of these instructions would cause a trapping IEEE exception, the implementation does one of the following:

- Cause, instead, an unimplemented exception. The multiple-operation instruction is placed in an exception register along with the appropriate unimplemented exception code. Or,
- Treat the multiple-operation instruction as two separate single-operation instructions. In this case, an instruction pattern is fabricated for the portion of the instruction that caused the trapping exception (e.g., the instruction pattern for a FLOATING-POINT ADD if the add operation caused the exception), and this pattern, along with the appropriate exception code, is placed in an exception register. The other operation, if it does not also cause a trapping exception, completes normally. If both operations cause trapping IEEE exceptions, then two instruction patterns are fabricated and placed in two exception registers. These two instructions have the same ordering constraints with respect to other instructions as for other single-operation instructions as described in the next section.

Figure 6-9 shows the format of the exception registers.

exception	ei
6	26

Figure 6-9. Exception Register Format

Field	**Description**
exception	The exception code corresponding to the exception detected as shown in Table 6-18. Exception codes not listed are reserved.

Table 6-18. Exception Codes

Exception code	Opcode	Description
000000	0C/0E	No exception
100000	0C/0E	Invalid operation
010000	0C/0E	Division-by-zero
001000	0C/0E	Overflow
pp0100	0C/0E	Underflow
000010	0C/0E	Inexact
000001	0C/0E	Unimplemented
001010	0C/0E	Inexact & Overflow
pp0110	0C/0E	Inexact & Underflow
001001	0C	Unimplemented
001011	0E	Unimplemented
000011	06	Unimplemented
100011	26	Unimplemented

The two bits labeled ‘pp’ in the exception code contain information regarding the parameters for the underflow exceptions. See “Underflow Exception” on page 6-33 for a detailed description of this field.

ei All bits other than the major opcode, copied from the excepting instruction. This field is undefined if the exception code is set to ‘no exception’.

Exception Register Operation

When all pending instructions are forced to complete, all operations which complete with a trapping exception are placed in the exception registers together with their corresponding exception codes. In order to complete an operation, the coprocessor may place the operation in an exception register and mark it with an unimplemented exception.

The coprocessor places the excepting instruction that first entered the IA queues in any of Exception Registers 1 through 7. Other instructions which complete with a trap are placed in any of the other available Exception Registers (those which are not already occupied by excepting instructions). Excepting instructions may be placed in the Exception Registers 1 through 7 in any order as long as the data dependencies are preserved (the order need not be the order in which they were fetched). If an instruction completes without a trapping exception, no record of that instruction appears in the exception registers. The exception queue need not be packed.

Once software has processed the exception registers, it must clear the exception registers by setting them all to zeros before non-load/store instructions can be executed.

If the T-bit equals 0 and any exception register has an exception field not equal to "no exception", execution of any non-load/store floating-point instruction is an undefined operation.

Interruptions and Exceptions

Floating-point instructions may cause interruptions in the processor, exceptions in the coprocessor, or both. Coprocessor exceptions are divided into immediate trapping exceptions and delayed trapping exceptions. The only immediate trapping exception is the reserved-op exception. The delayed trapping exceptions consist of the unimplemented exception and the IEEE exceptions.

The IEEE exceptions are the following:

- invalid operation
- division-by-zero
- inexact
- overflow
- underflow

While the unimplemented and reserved-op exceptions must always trap, the IEEE exception traps may be disabled.

Each IEEE exception has a corresponding enable bit in the Status Register. When an enable bit is 1, the corresponding trap is enabled, and if the corresponding exception occurs, a delayed trap is taken. However, on the overflow and underflow exceptions, an implementation may choose to ignore the enable bit and always trap on the exception. In such implementations, the corresponding trap is always enabled.

Immediate Trapping

Floating-point instructions may cause three types of immediate trapping interruptions: memory reference interruptions, the assist emulation trap, and the reserved-op exception. Immediate trapping exceptions and interruptions always cause a trap or fault when the front of the IA queues points to the interrupting instruction. An interrupting instruction must not alter its operands.

As described in Chapter 4, “Flow Control and Interruptions”, when the processor detects a memory reference problem, a memory reference fault or trap occurs. Only load and store instructions cause memory reference interruptions. The memory reference interruptions associated with floating-point instructions are the following:

- Data TLB miss fault/Data page fault
- Data memory access rights trap
- Data memory protection ID trap
- Unaligned data reference trap
- Data memory break trap
- TLB dirty bit trap
- Page reference trap
- Data debug trap

As described in Chapter 5, “Instruction Set”, the Coprocessor Configuration Register (CCR) in the processor controls the assist emulation trap. Software may set this register to force an assist emulation trap on every occurrence of a floating-point instruction. See “Coprocessor Instructions” on page 5-178 for more information.

Finally, attempting an instruction with a reserved sub-opcode may cause an immediate assist exception trap. See “Reserved-op Exception” on page 6-30 for details.

Delayed Trapping

Delayed traps report an exception when the excepting instruction is a pending instruction but is not in the IA queues. The following descriptions indicate when the processor and coprocessor may take a delayed trap and must take a delayed trap. Normally, a delayed trap forces the processor to take an assist exception trap. However, if the current instruction is a double-word store of Floating-point Register 0, all the floating-point registers are set normally as if a trap occurred, but the processor does not take the assist exception trap.

The coprocessor may signal a delayed trap when at least one of the following occurs:

- A pending instruction caused an unimplemented exception and the current instruction is a floating-point instruction, or
- A pending instruction caused an IEEE exception, the corresponding exception trap is enabled, and the current instruction is a floating-point instruction.

A delayed trap must occur when at least one of the following conditions exist:

- The T-bit is 1 and the current instruction is any floating-point instruction.
- The exception queue is full.
- A pending instruction causes a trapping exception and the current instruction is a load or store of Floating-point Register 0.
- The current instruction is a load or store of an exception register that will be set by a pending instruction.
- The current instruction is a load or store of the destination register of a pending, trapping instruction or an operation which depends on a pending, trapping instruction.
- The current instruction is a load of the source register of a pending, trapping instruction or an operation which depends on a pending, trapping instruction.
- The current instruction is a FLOATING-POINT TEST instruction and the previous FLOATING-POINT COMPARE either is pending and caused a trapping exception or depends on a pending, trapping instruction.

An instruction depends on a previous instruction whenever it must wait for the previous instruction to complete in order to ensure that the instructions appear sequentially executed to software. Instruction dependency is transitive. For example, if the exception queue is full, and every instruction in the queue depends on the instruction immediately preceding it, then each instruction in the queue depends on all the instructions preceding it.

When a delayed trap occurs, the following happens:

1. The coprocessor completes all pending floating-point instructions.
2. The coprocessor sets the exception registers as described in "Exception Register Operation" on page 6-25.
3. The coprocessor sets the Status Register T-bit to 1.
4. For each pending instruction that completes with a trapping IEEE exception, the corresponding exception flag may either be set to 1, or left unchanged, but cannot be set to 0.
5. If the current instruction is any floating-point instruction except a double-word store of Floating-point Register 0, the processor takes an assist exception trap. Otherwise, if the current instruction is a double-word store of Floating-point Register 0, the store completes, no trap occurs, the T-bit is set to 0 and execution proceeds normally.

Any pending instruction which depends on a pending, trapping instruction must complete with an unimplemented exception.

Table 6-19 specifies the status of the source and destination registers when an instruction causes a delayed trap. When the table indicates the original operand values are preserved, and if the destination register is not one of the source registers, the contents of the destination register are undefined.

Table 6-19. Delayed Trap Results

Exception type	Trapped result
Invalid operation	original operand values preserved
Division-by-zero	original operand values preserved
Overflow	rounded bias-adjusted result in destination
Underflow	rounded bias-adjusted result in destination
Inexact	rounded result in destination
Unimplemented	original operand values preserved

As indicated in the table, trapping overflow exceptions and underflow exceptions return a rounded bias-adjusted result. A bias-adjusted result is obtained by dividing (in the case of overflow) or multiplying (in the case of underflow) the infinitely precise result by 2^a and then rounding. The bias adjustment, a, is 192 for single-word numbers, 1536 for double-word numbers, and 24576 for quad-word numbers.

Non-trapping Exceptions

If an IEEE exception occurs, but the corresponding trap is disabled, then the coprocessor sets the corresponding flag bit in the Status Register to 1. Table 6-20 lists the results returned by an operation which completes with a non-trapping exception.

Table 6-20. Non-trapped Exception Results

Exception type	Non-trapped result
Invalid operation	quiet NaN to destination
Division-by-zero	properly signed ∞ to destination
Overflow	rounded result to destination
Underflow	rounded result to destination
Inexact	rounded result to destination

Multiple Exceptions

If the current instruction causes a reserved-op exception, and at the same time the coprocessor signals a delayed trap caused by a previous exception, the delayed trap occurs. Software then retries the instruction to handle the reserved-op exception.

The only other exceptions which may both occur on the same instruction are one of the following:

- inexact and overflow exceptions
- inexact and underflow exceptions

When one of these two cases occur, the action taken is as follows:

1. If both traps are enabled when the coprocessor takes a delayed trap, the implementation may set either or both corresponding status flags to 1, or leave them unchanged. The coprocessor sets the exception field in the corresponding exception register to the value that indicates both exceptions occurred.
2. If only one trap is enabled when the coprocessor takes a delayed trap, the coprocessor sets the corresponding exception field to the value that indicates the enabled trap. The implementation may either set the flag bit that corresponds to the enabled trap to 1, or leave it unchanged. The coprocessor sets the flag bit that corresponds to the disabled trap to 1.
3. If neither trap is enabled, the coprocessor sets both corresponding status flags to 1.

If the overflow or underflow exception caused a trap on the instruction, a rounded-bias adjusted result is returned. Otherwise, a rounded result is returned.

Trap Handlers

PROGRAMMING NOTE

The IEEE standard strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions. The mechanisms to accomplish this are programming language and operating system dependent.

Since the coprocessor continues to trap if the Status Register T-bit is 1, the trap handler must first set the bit to 0 by executing a double-word store of register 0. The trap handler may then emulate any of the instructions in the exception queue beginning with the instruction in Exception Register 1 and proceeding sequentially to the end.

The trap handler must clear all the exception registers. If the trap handler chooses not to emulate all the instructions, it must reset the T-bit to 1 before returning to the trapped process.

To emulate an instruction, the trap handler computes or specifies a substitute result to be placed in the destination register of the operation. The trap handler may determine what operation was being performed and what exceptions occurred during the operation by examining the corresponding exception register. On overflow, underflow, and inexact exceptions, the trap handler has access to the correctly rounded result by examining the destination register of the operation. On unimplemented, invalid operation, and divide-by-zero exceptions, the trap handler has access to the operand values by examining the source registers of the instruction.

Reserved-op Exception

When a non-load/store instruction has a reserved sub-opcode, an implementation signals either a reserved-op exception or an unimplemented exception.

A reserved-op exception always forces the processor to take an immediate assist exception trap. It does not set the exception registers or the T-bit, and does not change any of the flag bits in the Status Register. The reserved-op exception cannot be disabled.

PROGRAMMING NOTE

Trapping is immediate for reserved-op exceptions. The trap handler must check for a Status Register T-bit equal to 0 to determine that the trap was caused by a reserved-op exception. When a reserved-op exception occurs, software interprets the contents of the IIR, nullifies the instruction pointed to by the front of the IIA queues, and returns control to the trapping process.

Unimplemented Exception

If an implementation chooses not to execute an instruction, the instruction signals an unimplemented exception. An unimplemented exception always causes a delayed trap on a later floating-point instruction. It does not change the Status Register Flag bits and cannot be disabled. When a non-load/store floating-point operation references a reserved sub-opcode, an implementation signals either an unimplemented exception or a reserved-op exception.

An implementation may signal an unimplemented exception on any floating-point instruction except the FLOATING-POINT TEST instruction, the FLOATING-POINT IDENTIFY instruction, a load instruction, or a store instruction.

When a trap forces the coprocessor to complete all pending instructions, implementations may put uncompleted instructions in the exception registers and set the corresponding *exception* field to the appropriate unimplemented exception code.

A conversion to a floating-point format always causes an unimplemented exception when the result overflows, the result lies too far outside the range for the exponent to be bias-adjusted, and the overflow trap is enabled. Table 6-21 shows the result values which produce an unimplemented exception; a is the bias-adjustment value for the destination format, p is the precision, and v is the source value.

Table 6-21. Overflow Results Causing Unimplemented Exception

Rounding Mode	Ranges	
nearest	$-\infty < v \le -2^{(E_{\max} + a)}\left(2 - 2^{-p}\right)$	$2^{(E_{\max} + a)}\left(2 - 2^{-p}\right) \le v < +\infty$
to 0	$-\infty < v \le -2^{(E_{\max} + a + 1)}$	$2^{(E_{\max} + a + 1)} \le v < +\infty$
to $+\infty$	$-\infty < v \le -2^{(E_{\max} + a + 1)}$	$2^{(E_{\max} + a)}\left(2 - 2^{-(p-1)}\right) < v < +\infty$
to $-\infty$	$-\infty < v < -2^{(E_{\max} + a)}\left(2 - 2^{-(p-1)}\right)$	$2^{(E_{\max} + a + 1)} \le v < +\infty$

Similarly, an unimplemented exception is always caused by a conversion to a floating-point format that underflows, lies too far outside the range for the exponent to be bias-adjusted, and the underflow trap is enabled. Table 6-22 shows the floating-point underflow results which cause an unimplemented exception; a is the bias-adjustment value for the destination format, p is the precision, and v is the source value.

Table 6-22. Underflow Results Causing Unimplemented Exception

Rounding Mode	Range
nearest	$-2^{(E_{\min} - a)}\left(1 - 2^{-(p+1)}\right) < v < 2^{(E_{\min} - a)}\left(1 - 2^{-(p+1)}\right)$
to 0	$-2^{(E_{\min} - a)} < v < 2^{(E_{\min} - a)}$
to $+\infty$	$-2^{(E_{\min} - a)} < v \le 2^{(E_{\min} - a)}\left(1 - 2^{-p}\right)$
to $-\infty$	$-2^{(E_{\min} - a)}\left(1 - 2^{-p}\right) \le v < 2^{(E_{\min} - a)}$

Finally, the unimplemented exception is always signaled when the operand of a conversion to an integer format is a NaN or an ∞, or when the result overflows. Table 6-23 shows the results which produce an integer overflow. In the table, $I_{\max}$ is the most positive integer representable by the destination format, $I_{\min}$ is the most negative, and v is the source value.

Table 6-23. Integer Results Causing Unimplemented Exception

Rounding Mode	Ranges	
nearest	$v < I_{\min} - 1/2$	$v \ge I_{\max} + 1/2$
to 0	$v \le I_{\min} - 1$	$v \ge I_{\max} + 1$
to $+\infty$	$v \le I_{\min} - 1$	$v > I_{\max}$
to $-\infty$	$v < I_{\min}$	$v \ge I_{\max} + 1$

Invalid Operation Exception

An instruction signals the invalid operation exception if an operand is invalid for the operation to be performed. When the exception occurs without a trap, the coprocessor delivers a quiet NaN to the destination register. If the exception causes a trap, the coprocessor leaves the operands unchanged.

The invalid operations are:

1. Any arithmetic operation on a signaling NaN except for conversions to integer formats.
2. Magnitude subtraction of infinities like $(+\infty) + (-\infty)$ or $(+\infty) - (+\infty)$;
3. The multiplication of 0 and ∞;
4. The division operations $0/0$ and ∞/∞ ;
5. Square root if the operand is less than zero;
6. Comparison using conditions involving a "T" or conditions involving "<", ">", "true", or "false" without a "?", when the operands are unordered. See “Comparison Conditions” on page 6-17.

Division-by-zero Exception

From the standard:

> [§]**7.2 Division by zero.** If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception is signaled. The result, when no trap occurs, is a correctly signed ∞ (6.3) [see “Sign Bit” on page 6-23].

When a trap occurs, the operands must be left unchanged.

Inexact Exception

From the standard:

> [§]**7.5 Inexact.** If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination or, if an inexact trap occurs, to the trap handler [the destination register in this architecture].

A conversion to a fixed-point format also signals the inexact exception when the result is not exact.

Overflow Exception

To determine overflow on an operation, the coprocessor uses the result that would have occurred had the result been computed and rounded as if the destination’s exponent range were unbounded. On all operations except converts, the coprocessor signals an overflow exception when the magnitude of this result exceeds the destination format’s largest finite number. The same is true of conversion operations, except that when this result is beyond the range of bias-adjusted numbers and the overflow trap is enabled, the instruction causes an unimplemented exception.

An instruction cannot cause an overflow exception when at least one operand is a NaN or infinity.

Table 6-24 summarizes the result values that cause an overflow exception. In the table, E_{max} is the maximum exponent value for the destination format, p is the precision of the format, and v is the value of the exact result before rounding.

Table 6-24. Results Causing Overflow Exception

Rounding Mode	Ranges	
nearest	$2^{E_{max}}\left(2-2^{-p}\right) \le v < +\infty^{*}$	$-\infty^{*} < v \le -2^{E_{max}}\left(2-2^{-p}\right)$
to 0	$2^{(E_{max}+1)} \le v < +\infty^{*}$	$-\infty^{*} < v \le -2^{(E_{max}+1)}$
to +∞	$2^{E_{max}}\left(2-2^{-(p-1)}\right) < v < +\infty^{*}$	$-\infty^{*} < v \le -2^{(E_{max}+1)}$
to −∞	$2^{(E_{max}+1)} \le v < +\infty^{*}$	$-\infty^{*} < v < -2^{E_{max}}\left(2-2^{-(p-1)}\right)$

* When the overflow trap is enabled and the operation is a conversion to a floating-point format, this bound is limited to bias-adjusted numbers. See “Unimplemented Exception” on page 6-30.

When no trap occurs, the result of an overflow exception is one of the following:

1. Round to nearest carries all overflows to ∞ with no change in sign.
2. Round toward 0 carries all overflows to the format’s largest finite number with no change in sign.
3. Round toward −∞ carries positive overflows to the format’s largest finite number, and carries negative overflows to −∞.
4. Round toward +∞ carries negative overflows to the format’s most negative finite number, and carries positive overflows to +∞.

When an overflow exception causes a trap, the excepting operation returns a bias-adjusted number to the destination register.

The overflow exception is not signaled for integer results. The coprocessor signals integer overflows with an unimplemented exception.

Underflow Exception

From the standard:

> [§]**7.4 Underflow.** Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between $\pm 2^{E_{min}}$ which, because it is tiny, may cause some other exception later such as overflow upon division. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

Tininess is detected on a nonzero result which lies strictly between $\pm 2^{E_{min}}$, when the result is rounded as if the exponent range were unbounded. Note that rounding for detection of tininess and rounding to determine a result are distinct. In certain cases, the coprocessor signals an underflow exception even though it returns a normalized result to the destination register.

Table 6-25 shows the range of exact results which will cause detection of tininess. In the table, E_{min} is the minimum exponent value for the destination format, p is the precision of the format, and v is the value of the exact result before rounding.

Table 6-25. Results Causing Tininess

Rounding Mode	Range
nearest	$-2^{E_{min}}\left(1-2^{-(p+1)}\right) < v < 2^{E_{min}}\left(1-2^{-(p+1)}\right)$
to 0	$-2^{E_{min}} < v < 2^{E_{min}}$
to +∞	$-2^{E_{min}} < v \le 2^{E_{min}}\left(1-2^{-p}\right)$
to −∞	$-2^{E_{min}}\left(1-2^{-p}\right) \le v < 2^{E_{min}}$

Loss of accuracy occurs when the coprocessor detects an inexact result, where the result returned after rounding differs from what the result would have been if the destination had infinite precision and unbounded range.

An instruction causes an underflow exception when the underflow trap is enabled and tininess occurs. An instruction also causes an underflow exception when the underflow trap is disabled and both tininess and loss of accuracy occur.

An operation which causes a non-trapping underflow exception may return a zero, denormalized number, or $\pm 2^{E_{min}}$.

Trapped underflows on all operations except conversions deliver a bias-adjusted result to the destination register. Trapped underflow on conversions to a floating-point format delivers a bias-adjusted result when the result can be represented by a bias-adjusted number. If not, an unimplemented exception is signaled instead of an underflow exception.

Conversion to an integer format cannot underflow. The result when the magnitude of the source operand is less than 1 is either 0, +1, or −1 depending on the rounding mode and the sign of the source operand.

When an instruction causes a trapping underflow exception and the trap enable bit equals 0, the leftmost two bits in the corresponding exception register's *exception* field are set (see Figure 6-10). The first parameter bit, the round away (RA) bit, is set to 1 whenever the result is rounded away from zero. The second is the inexact (I) bit which is set to 1 if the rounded bias-adjusted result is not the infinitely precise result. The trap handler uses this information to denormalize the result and prevent errors caused by rounding twice.

RA	I	0	1	0/1	0
1	1	1	1	1	1

Figure 6-10. Exception Field Underflow Parameters

Saving and Restoring State

To save state, software first performs a double-word store of register 0, then double-word stores of registers 1, 2, and 3, and a sufficient number of double-word stores to save registers needed at a later time. Thirty-two double-word coprocessor stores are sufficient to save the entire state of the floating-point coprocessor.

A double-word store of register 0 cancels all pending traps, forces the completion of all previous instructions, suppresses any ensuing trap, completes the store, and sets the Status Register T-bit to 0. When the store cancels a trap, the value written to memory has the bit corresponding to the Status Register T-bit set to 1; otherwise, this bit is set to 0. This special treatment of a double-word store lets the save routine be nested, does not require the assistance of a trap handler, and need not have the IIA queues enabled.

To restore state, software performs double-word loads of all required registers, followed by a double-word load of Floating-point Register 0. Thirty-two double-word loads are sufficient to restore the entire state of the coprocessor. A double-word load of Floating-point Register 0 which sets the Status Register T-bit to 1 re-arms a trap. The next floating-point instruction will cause a trap (apart from a double-word store of Floating-point Register 0).

The following sequences save and restore the entire state of the coprocessor.

```
; enter with SaveAreaPtr pointing at the first double-word of the save area
SAVEFPU
            FSTDS,MA    FPR0,8(SaveAreaPtr)     ;quiescent,cancel trap
            FSTDS,MA    FPR1,8(SaveAreaPtr)     ;save exception register
            FSTDS,MA    FPR2,8(SaveAreaPtr)     ;save exception register
            FSTDS,MA    FPR3,8(SaveAreaPtr)     ;save exception register
            FSTDS,MA    FPR4,8(SaveAreaPtr)     ;save data register
            FSTDS,MA    FPR5,8(SaveAreaPtr)     ;save data register
                •
                •
                •
            FSTDS,MA    FPR30,8(SaveAreaPtr)    ;save data register
            FSTDS       FPR31,0(SaveAreaPtr)    ;save last data register
```

```
; enter with SaveAreaPtr pointing at the last double-word of the save area.
RSTFPU
            FLDDS        0(SaveAreaPtr),FPR31     ;restore data register
            FLDDS,MB     -8(SaveAreaPtr),FPR30    ;restore data register
               •
               •
               •
            FLDDS,MB     -8(SaveAreaPtr),FPR4     ;restore data register
            FLDDS,MB     -8(SaveAreaPtr),FPR3     ;restore exception register
            FLDDS,MB     -8(SaveAreaPtr),FPR2     ;restore exception register
            FLDDS,MB     -8(SaveAreaPtr),FPR1     ;restore exception register
            FLDDS,MB     -8(SaveAreaPtr),FPR0     ;restore exception register
                                                  ;potentially re-arm trap
```

The only required ordering in these sequences is that Floating-point Register 0 must be saved first and restored last.

Instruction Set Description

The following pages contain descriptions of each instruction.

The *Description* section of each non-load/store instruction contains a list of the Floating-Point Exceptions which the instruction may cause. Each instruction description has an *Exceptions* section which lists the processor interruptions that may occur while the instruction is pointed to by the front of the IA queues.

In the following pages, the notation, FPR, refers to floating-point coprocessor registers 0 through 31. FPSR refers to the Floating-point Status Register. Refer to “Instruction Notations” on page 5-7 for the explanation of the operation section. The mem_load and the mem_store descriptions are located in “Memory Reference Instructions” on page 5-15.

Format: FLDWX,cmplt,cc x(s,b),t

09	b	x	s	u	0	cc	0	0	t	m	t
6	5	5	2	1	1	2	1	2	1	1	5

Purpose: To load a word into a floating-point coprocessor register.

Description: The aligned word at the effective address is loaded into floating-point register *t*. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 2. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Specifying floating-point registers 0R, 1L, 1R, 2L, 2R, 3L, or 3R is an undefined operation. Specifying Floating-point Register 0L forces the coprocessor to complete all previous floating-point instructions. However, loading Floating-point Register 0L with a value that sets the Status Register T-bit to 1 is an undefined operation.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],2);       /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                          /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                          /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],2);
                break;
    default:    offset ← GR[b] + GR[x];                  /*u=0, m=0*/
                break;
}
FPR[t] ← mem_load(space,offset,0,31,cc);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap
Assist emulation trap

FLOATING-POINT LOAD DOUBLEWORD INDEXED — FLDDX

Format: FLDDX,cmplt,cc x(s,b),t

0B	b	x	s	u	0	cc	0	0	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a doubleword into a floating-point coprocessor register.

Description: The aligned doubleword at the effective address is loaded into floating-point register *t*. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 3. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Specifying Floating-point Register 0 forces the coprocessor to complete all previous floating-point instructions.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:      offset ← GR[b] + lshift(GR[x],3);         /*u=1, m=0*/
                 break;
    case M:      offset ← GR[b];                           /*u=0, m=1*/
                 GR[b] ← GR[b] + GR[x];
                 break;
    case SM:     offset ← GR[b];                           /*u=1, m=1*/
                 GR[b] ← GR[b] + lshift(GR[x],3);
                 break;
    default:     offset ← GR[b] + GR[x];                   /*u=0, m=0*/
                 break;
}
FPR[t] ← mem_load(space,offset,0,63,cc);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap
Assist emulation trap

FLOATING-POINT STORE WORD INDEXED — FSTWX

Format: FSTWX,cmplt,cc r,x(s,b)

09	b	x	s	u	0	cc	1	0	r	m	r
6	5	5	2	1	1	2	1	2	1	1	5

Purpose: To store a word from a floating-point coprocessor register.

Description: Floating-point register *r* is stored in the aligned word at the effective address. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 2. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Specifying floating-point registers 0R, 1L, 1R, 2L, 2R, 3L, or 3R is an undefined operation. Specifying Floating-point Register 0L forces the coprocessor to complete all previous floating-point instructions.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],2);          /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                            /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                            /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],2);
                break;
    default:    offset ← GR[b] + GR[x];                    /*u=0, m=0*/
                break;
}
mem_store(space,offset,0,31,cc,FPR[r]);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap
Assist emulation trap

FLOATING-POINT STORE DOUBLEWORD INDEXED FSTDX

Format: FSTDX,cmplt,cc r,x(s,b)

0B	b	x	s	u	0	cc	1	0	m	r
6	5	5	2	1	1	2	1	3	1	5

Purpose: To store a doubleword from a floating-point coprocessor register.

Description: Floating-point register *r* is stored in the aligned doubleword at the effective address. The base register, *b*, and the index register, *x*, are combined to form an address offset. The completer, *cmplt*, determines if the offset is the base register, the base register plus the index register, or the base register plus the index register shifted by 3. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-11 on page 5-22 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Specifying Floating-point Register 0 forces the coprocessor to complete all previous floating-point instructions and sets the Status Register T-bit to 0 following completion of the store.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case S:     offset ← GR[b] + lshift(GR[x],3);        /*u=1, m=0*/
                break;
    case M:     offset ← GR[b];                          /*u=0, m=1*/
                GR[b] ← GR[b] + GR[x];
                break;
    case SM:    offset ← GR[b];                          /*u=1, m=1*/
                GR[b] ← GR[b] + lshift(GR[x],3);
                break;
    default:    offset ← GR[b] + GR[x];                  /*u=0, m=0*/
                break;
}
mem_store(space,offset,0,63,cc,FPR[r]);
if (r == 0)
    FPSR[T] ← 0;
```

Exceptions:

Assist exception trap	Data memory break trap
Data TLB miss fault/data page fault	TLB dirty bit trap
Data memory access rights trap	Page reference trap
Data memory protection ID trap	Data debug trap
Unaligned data reference trap	Assist emulation trap

FLOATING-POINT LOAD WORD SHORT **FLDWS**

Format: FLDWS,cmplt,cc d(s,b),t

09	b	im5	s	a	1	cc	0	0	t	m	t
6	5	5	2	1	1	2	1	2	1	1	5

Purpose: To load a word into a floating-point coprocessor register.

Description: The aligned word at the effective address is loaded into floating-point register *t*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Specifying floating-point registers 0R, 1L, 1R, 2L, 2R, 3L, or 3R is an undefined operation. Specifying Floating-point Register 0L forces the coprocessor to complete all previous floating-point instructions. However, loading Floating-point Register 0L with a value that sets the Status Register T-bit to 1 is an undefined operation.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:   offset ← GR[b] + low_sign_ext(im5,5);          /*a=1, m=1*/
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    case MA:   offset ← GR[b];                                /*a=0, m=1*/
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    default:   offset ← GR[b] + low_sign_ext(im5,5);               /*m=0*/
               break;
}
FPR[t] ← mem_load(space,offset,0,31,cc);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Page reference trap
Data debug trap
Assist emulation trap

FLOATING-POINT LOAD DOUBLEWORD SHORT FLDDS

Format: FLDDS,cmplt,cc d(s,b),t

0B	b	im5	s	a	1	cc	0	0	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a doubleword into a floating-point coprocessor register.

Description: The aligned doubleword at the effective address is loaded into floating-point register *t*. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-8 on page 5-17).

Specifying Floating-point Register 0 forces the coprocessor to complete all previous floating-point instructions.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                              /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);        /*m=0*/
                break;
}
FPR[t] ← mem_load(space,offset,0,63,cc);
```

Exceptions:

Assist exception trap	Unaligned data reference trap
Data TLB miss fault/data page fault	Page reference trap
Data memory access rights trap	Data debug trap
Data memory protection ID trap	Assist emulation trap

FLOATING-POINT STORE WORD SHORT **FSTWS**

Format: FSTWS,cmplt,cc r,d(s,b)

09	b	im5	s	a	1	cc	1	0	r	m	r
6	5	5	2	1	1	2	1	2	1	1	5

Purpose: To store a word from a floating-point coprocessor register.

Description: Floating-point register *r* is stored in the aligned word at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Specifying floating-point registers 0R, 1L, 1R, 2L, 2R, 3L, or 3R is an undefined operation. Specifying Floating-point Register 0L forces the coprocessor to complete all previous floating-point instructions.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:   offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    case MA:   offset ← GR[b];                              /*a=0, m=1*/
               GR[b] ← GR[b] + low_sign_ext(im5,5);
               break;
    default:   offset ← GR[b] + low_sign_ext(im5,5);        /*m=0*/
               break;
}
mem_store(space,offset,0,31,cc,FPR[r]);
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap
Assist emulation trap

FLOATING-POINT STORE DOUBLEWORD SHORT FSTDS

Format: FSTDS,cmplt,cc r,d(s,b)

0B	b	im5	s	a	1	cc	1	0	m	r
6	5	5	2	1	1	2	1	3	1	5

Purpose: To store a doubleword from a floating-point coprocessor register.

Description: Floating-point register *r* is stored in the aligned doubleword at the effective address. The completer, *cmplt*, determines if the offset is the base register, *b*, or the base register plus the short displacement, *d*. The displacement is encoded in the *im5* field. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-12 on page 5-24 for the assembly language completer mnemonics.)

The completer, *cc*, specifies the cache control hint (see Table 5-9 on page 5-18).

Specifying Floating-point Register 0 forces the coprocessor to complete all previous floating-point instructions and sets the Status Register T-bit to 0 following completion of the store.

Operation:

```
space ← space_select(s,GR[b]);
switch (cmplt) {
    case MB:    offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    case MA:    offset ← GR[b];                              /*a=0, m=1*/
                GR[b] ← GR[b] + low_sign_ext(im5,5);
                break;
    default:    offset ← GR[b] + low_sign_ext(im5,5);        /*m=0*/
                break;
}
mem_store(space,offset,0,63,cc,FPR[r]);
if (r == 0)
    FPSR[T] ← 0;
```

Exceptions:

Assist exception trap
Data TLB miss fault/data page fault
Data memory access rights trap
Data memory protection ID trap
Unaligned data reference trap
Data memory break trap
TLB dirty bit trap
Page reference trap
Data debug trap
Assist emulation trap

FLOATING-POINT CONVERT FROM FLOATING-POINT TO FLOATING-POINT — FCNVFF

Format: FCNVFF,sf,df   r,t

0E	r	0	0	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

0C	r	0	0	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

Purpose: To change a floating-point value of one format to a floating-point value of a different format.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is interpreted in the specified source format, *sf*, and arithmetically converted to the specified destination format, *df*. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Rounding occurs according to the currently specified rounding mode.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: FPR[t] $\leftarrow$ convert_float_to_float(FPR[r],sf,df,FPSR[RM]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT CONVERT FROM FIXED-POINT TO FLOATING-POINT — FCNVXF

Format: FCNVXF,sf,df r,t

0E	r	0	1	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

0C	r	0	1	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

Purpose: To change the format of a fixed-point value to a floating-point value.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is interpreted as a signed fixed-point number in the specified source format, *sf*, and arithmetically converted to the specified destination format, *df*, as a floating-point number. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Rounding occurs according to the currently specified rounding mode.

Floating-point exceptions:

- Unimplemented
- Inexact

Operation: FPR[t] $\leftarrow$ convert_fixed_to_float(FPR[r],sf,df,FPSR[RM]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT

FCNVFX

Format: FCNVFX,sf,df r,t

0E	r	0	2	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

0C	r	0	2	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

Purpose: To change the format of a floating-point value to a fixed-point value.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is interpreted as a floating-point number in the specified source format, *sf*, and arithmetically converted to a signed fixed-point number in the specified destination format, *df*. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Rounding occurs according to the currently specified rounding mode.

Floating-point exceptions:

- Unimplemented
- Inexact

Operation: FPR[t] ← convert_float_to_fixed(FPR[r],sf,df,FPSR[RM]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT AND TRUNCATE — FCNVFXT

Format: FCNVFXT,sf,df   r,t

0E	r	0	3	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

0C	r	0	3	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

Purpose: To change the format of a floating-point value to a fixed-point value.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is interpreted as a floating-point number in the specified source format, *sf*, and arithmetically converted to a signed fixed-point number in the specified destination format, *df*. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

The current rounding mode is ignored and the result is rounded toward zero.

Floating-point exceptions:

- Unimplemented
- Inexact

Operation: FPR[t] $\leftarrow$ convert_float_to_fixed(FPR[r],sf,df,ROUND_TOWARD_ZERO);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT COPY FCPY

Format: FCPY,fmt r,t

0E	r	0	2	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

0C	r	0	2	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To copy a floating-point value to another floating-point register.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is copied into the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*. This operation is non-arithmetic and does not cause an invalid operation exception when a NaN is copied.

Floating-point exceptions:

- Unimplemented

Operation: FPR[t] $\leftarrow$ FPR[r];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT ABSOLUTE VALUE FABS

Format: FABS,fmt r,t

0E	r	0	3	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

0C	r	0	3	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point absolute value.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is copied to the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t* with the sign bit set to 0. This instruction is non-arithmetic and does not cause an invalid operation exception when the sign of a NaN is set to 0.

Floating-point exceptions:

- Unimplemented

Operation: FPR[t]{all_bits_except_sign} ← FPR[r]{all_bits_except_sign};
FPR[t]{sign_bit} ← 0;

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT SQUARE ROOT — FSQRT

Format: FSQRT,fmt r,t

0E	r	0	4	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

0C	r	0	4	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point square root.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is interpreted in the specified format and the positive arithmetic square root is taken. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. If the source register contains -0 , the result will be -0 . The result is placed in the appropriate side of the floating-point register, the entire the floating-point register, or the register pair specified by *t*.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Inexact

Operation: FPR[t] $\leftarrow$ square_root(FPR[r]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT ROUND TO INTEGER FRND

Format: FRND,fmt r,t

0E	r	0	5	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

0C	r	0	5	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To round a floating-point value to an integral value.

Description: The appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *r* is interpreted in the specified format and arithmetically rounded to an integral value. This result remains a floating-point number. Results are rounded according to the current rounding mode with the proviso that when rounding to nearest, if the difference between the unrounded operand and the rounded result is exactly one half, the rounded result is even. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*. An inexact exception is signaled when the result and source are not the same.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Inexact

Operation: FPR[t] $\leftarrow$ floating_point_round(FPR[r]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT ADD — FADD

Format: FADD,fmt   r1,r2,t

0E	r1	r2	0	r2	f	3	0	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

0C	r1	r2	0	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point addition.

Description: The appropriate sides of floating-point registers, entire floating-point registers, or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically added. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: FPR[t] $\leftarrow$ FPR[r1] + FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT SUBTRACT FSUB

Format: FSUB,fmt r1,r2,t

0E	r1	r2	1	r2	f	3	0	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

0C	r1	r2	1	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point subtraction.

Description: The appropriate sides of floating-point registers, entire floating-point registers, or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically subtracted. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: FPR[t] $\leftarrow$ FPR[r1] – FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT MULTIPLY — FMPY

Format: FMPY,fmt r1,r2,t

0E	r1	r2	2	r2	f	3	0	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

0C	r1	r2	2	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point multiply.

Description: The appropriate sides of floating-point registers, entire floating-point registers, or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: FPR[t] ← FPR[r1] * FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT DIVIDE FDIV

Format: FDIV,fmt r1,r2,t

0E	r1	r2	3	r2	f	3	0	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

0C	r1	r2	3	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point division.

Description: The appropriate sides of floating-point registers, entire floating-point registers, or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically divided. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register, the entire floating-point register, or the register pair specified by *t*.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Division-by-zero
- Overflow
- Underflow
- Inexact

Operation: FPR[t] $\leftarrow$ FPR[r1] / FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT MULTIPLY/ADD **FMPYADD**

Format: FMPYADD,fmt rm1,rm2,tm,ra,ta

06	rm1	rm2	ta	ra	f	tm
6	5	5	5	5	1	5

Purpose: To perform a floating-point multiply and a floating-point add.

Description: The appropriate sides of floating-point registers or entire floating-point registers specified by *rm1* and *rm2* are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register or the entire floating-point register specified by *tm*.

The appropriate sides of floating-point registers or entire floating-point registers specified by *ta* and *ra* are interpreted in the specified format and arithmetically added. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register or the entire floating-point register specified by *ta*.

The behavior of this instruction is undefined if *ra* specifies the same register as *tm*, or if *ta* specifies the same register as any of *rm1*, *rm2*, or *tm*. The behavior of this instruction is also undefined if *ra* specifies double-precision register 0 or single-precision register 16L.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: FPR[tm] ← FPR[rm1] * FPR[rm2];
FPR[ta] ← FPR[ta] + FPR[ra];

Exceptions: Assist emulation trap
Assist exception trap

Notes: When operating on single-precision operands, each register field specifies one of registers 16L through 31L, or one of 16R through 31R. See Table 6-17 on page 6-20 for the register specifier encodings.

This instruction can be decomposed into FMPY and FADD and then the full set of floating-point exceptions can be reported.

FLOATING-POINT MULTIPLY/SUBTRACT — FMPYSUB

Format: FMPYSUB,fmt rm1,rm2,tm,ra,ta

26	rm1	rm2	ta	ra	f	tm
6	5	5	5	5	1	5

Purpose: To perform a floating-point multiply and a floating-point subtract.

Description: The appropriate sides of floating-point registers or entire floating-point registers specified by *rm1* and *rm2* are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register or the entire floating-point register specified by *tm*.

The appropriate sides of floating-point registers or entire floating-point registers specified by *ta* and *ra* are interpreted in the specified format and arithmetically subtracted. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the appropriate side of the floating-point register or the entire floating-point register specified by *ta*.

The behavior of this instruction is undefined if *ra* specifies the same register as *tm*, or if *ta* specifies the same register as any of *rm1*, *rm2*, or *tm*. The behavior of this instruction is also undefined if *ra* specifies double-precision register 0 or single-precision register 16L.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: FPR[tm] ← FPR[rm1] * FPR[rm2];
FPR[ta] ← FPR[ta] - FPR[ra];

Exceptions: Assist emulation trap
Assist exception trap

Notes: When operating on single-precision operands, each register field specifies one of registers 16L through 31L, or one of 16R through 31R. See Table 6-17 on page 6-20 for the register specifier encodings.

This instruction can be decomposed into FMPY and FSUB and then the full set of floating-point exceptions can be reported.

FIXED-POINT MULTIPLY UNSIGNED XMPYU

Format: XMPYU r1,r2,t

0E	r1	r2	2	r2	0	3	1	r1	0	0	t
6	5	5	3	1	1	2	1	1	1	1	5

Purpose: To perform unsigned fixed-point multiplication.

Description: The floating-point registers specified by *r1* and *r2* are interpreted as unsigned 32-bit integers and arithmetically multiplied. The unsigned 64-bit result is placed in the floating-point register specified by *t*.

Floating-point exceptions:

- Unimplemented

Operation: FPR[t] ← FPR[r1] * FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

Format: FCMP,fmt,cond r1,r2

0E	r1	r2	0	r2	f	2	0	r1	0	0	c
6	5	5	3	1	1	2	1	1	1	1	5

0C	r1	r2	0	fmt	2	0	0	c
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point comparison.

Description: The appropriate sides of floating-point registers, entire floating-point registers, or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically compared. A result is determined based on the comparison and the condition, *cond*. The condition is encoded in the *c* field of the instruction.

The CQ field in the floating-point Status Register is shifted right by one bit (discarding the rightmost bit) and the C-bit is copied into CQ{0}. Then, if the comparison result is true, the C-bit in the floating-point Status Register is set to 1, otherwise the C-bit is set to 0.

If at least one of the values is a signaling NaN, or if at least one of the values is a NaN and the low-order bit of the condition is 1, an invalid operation exception is signaled.

For unimplemented and trapped invalid operation exceptions, the state of the C-bit is unchanged, and the CQ field is not shifted.

For untrapped invalid operation exceptions, the state of the C-bit is the AND of the unordered relation (which is true) and bit 3 of the *c* field.

Comparisons are exact and neither overflow nor underflow. Four mutually exclusive relations are possible results: *less than*, *equal*, *greater than*, and *unordered*. The last case arises when at least one operand is a NaN. Every NaN compares *unordered* with everything, including itself. Comparisons ignore the sign of zero, so $+0 = -0$.

Floating-point exceptions:

- Unimplemented
- Invalid operation

Operation:

```
if (NaN(FPR[r1]) || NaN(FPR[r2]))
    if (c{4})
        invalid_operation_exception;
    else {
        greater_than ← false;
        less_than ← false;
        equal_to ← false;
        unordered ← true;
    }
```

```
else {
    greater_than ← FPR[r1] > FPR[r2];
    less_than ← FPR[r1] < FPR[r2];
    equal_to ← FPR[r1] = FPR[r2];
    unordered ← false;
}
FPSR[CQ] ← rshift(FPSR[CQ],1);
FPSR[CQ{0}] ← FPSR[C];
FPSR[C] ← (((c{0} == 1) && greater_than) ||
           ((c{1} == 1) && less_than) ||
           ((c{2} == 1) && equal_to) ||
           ((c{3} == 1) && unordered));
```

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT TEST FTEST

Format: FTEST,cond

0C	0	0	1	0	2	0	1	c
6	5	5	3	2	2	3	1	5

Purpose: To test the results of one or more earlier comparisons.

Description: The specified condition in the floating-point Status Register is tested. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied, then the following instruction is nullified.

Floating-point exceptions:

- None

Conditions: The condition is any of the conditions shown in Table 6-14 on page 6-18. When a condition completer is not specified, the "Simple Test" (C == 1) condition is used. The boolean variable "cond_satisfied" in the operation section is set when the specified condition is satisfied.

Operation:

```
if (cond_satisfied)
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

Notes: This instruction must be implemented, may not be queued and may not cause any assist exception traps. However, any assist exception traps caused by previous instructions may be taken while this instruction is in the IA queue.

FLOATING-POINT IDENTIFY COPR,0,0

Format: COPR,0,0

0C	0
6	26

Purpose: To validate fields in the Status Register which identify the floating-point coprocessor.

Description: The *model* and *revision* fields in the Status Register become defined. The contents of the other fields in the Status Register are undefined after the execution of this instruction. The *model* and *revision* fields remain defined until a floating-point instruction is executed which is not a double-word store of register 0.

Floating-point exceptions:

- None

Operation: FPSR[model] ← *implementation-dependent model number*;
FPSR[revision] ← *implementation-dependent revision number*;

Exceptions: Assist emulation trap

Notes: This instruction must be implemented. Software may use the following sequence to obtain the *model* and *revision* fields in the Status Register:

```
        .CODE
        LDIL      L%fpreg0,r2         ; load address of
        LDO       R%fpreg0(r2),r2     ; fp reg0 save area
        FSTDS     fr0,0(r2)           ; save fp reg0, cancel exception traps
        COPR,0,0                      ; identify coprocessor
        LDIL      L%version,r2        ; load address of
        LDO       R%version(r2),r2    ; model/rev save area
        FSTDS     fr0,0(r2)           ; store coprocessor id, cancel
                                      ; exception traps
        .DATA
fpreg0  .DOUBLE   0
version .DOUBLE   0
```

For the COPR,0,0 instruction to work correctly, the floating-point instructions immediately preceding and following it must be double-word stores of Floating-point Register 0. If not, the instruction is an undefined operation.

The sequence described will work in user mode. For example, if a context switch occurs just prior to COPR,0,0 but after the first FSTDS 0,0(2) instruction, the floating-point state save and state restore sequence will restore the state of the Status Register ("T" bit off, cancel trap) just prior to the execution of COPR,0,0.

7 Performance Monitor Coprocessor

Introduction

This chapter describes the architecture for the performance monitor coprocessor.

The performance monitor coprocessor is an optional, implementation-dependent coprocessor which provides a minimal common software interface to implementation-dependent performance monitor hardware.

The performance monitor coprocessor responds to coprocessor instructions with a uid equal to 2.

The Instruction Set

The performance monitor instruction set consists of two instructions, PERFORMANCE MONITOR ENABLE (PMENB) and PERFORMANCE MONITOR DISABLE (PMDIS), which provide a common software interface to enable and disable the implementation-dependent performance monitor features.

Figure 7-1 shows the format of these operations and Table 7-1 shows the operations, their mnemonics, and sub-opcodes:

0C	rv	sub	2	n	rv
6	12	5	3	1	5

Figure 7-1. Performance Monitor Operation Format

Table 7-1. Performance Monitor Operations

Opcode	Sub-op	Mnemonic	Operation
0C	1	PMDIS	Disable performance monitor
0C	3	PMENB	Enable performance monitor
0C	0,2,4..1F		undefined

Interruptions

Performance Monitor Interruption

Interruption vector number 29 in interruption group 2 is defined as the performance monitor coprocessor interrupt for implementation-dependent use by the performance monitor coprocessor. The interrupt is unmasked when the PSW F-bit is 1, and is masked when the PSW F-bit is 0. See “Interruptions” on page 4-13 for additional details.

Monitor Units

The monitor units are hardware units used to collect the necessary information during performance monitoring. The number of the monitor units and their hardware types are implementation dependent.

If a monitor unit provides counters, the most significant bit of the counter is required to be an overflow indicator. The bit must be set when the counter overflows and must remain set until explicitly reset by software. When the overflow indicator is set the remaining bits of the counter are undefined.

NOTE

If counters are used to implement the measurement units, it is recommended that the counters be at least 32 bits wide.

Instruction Set Description

The following pages describe the performance monitor coprocessor instructions.

When a performance monitor coprocessor instruction is executed and CCR{2} is 0, the coprocessor instruction causes an assist emulation trap. It is an undefined operation to set CCR{2} to 1 if the performance monitor coprocessor is nonexistent.

PERFORMANCE MONITOR ENABLE — PMENB

Format: PMENB

0C	rv	3	2	0	rv
6	12	5	3	1	5

Purpose: To enable the implementation-dependent performance monitor coprocessor.

Description: Enable the measurement units, starting with the next instruction.

Operation: measurement_enabled $\leftarrow$ 1;

Exceptions: Assist emulation trap

PERFORMANCE MONITOR DISABLE PMDIS

Format: PMDIS,n

0C	rv	1	2	0	rv
6	12	5	3	1	5

Purpose: To disable the implementation-dependent performance monitor coprocessor.

Description: Disable all measurement units, after the current instruction.

Operation: measurement_enabled $\leftarrow$ 0;

Exceptions: Assist emulation trap

8 Debug Special Function Unit

Introduction

This chapter describes the architecture for the debug special function unit.

The debug special function unit is an optional, architected SFU which provides hardware assistance for software debugging using breakpoints. The debug SFU is currently defined only for Level 0 processors.

The debug special function unit responds to SFU instructions with an SFU id equal to 1.

Debug Registers

The debug SFU supports separate register sets for data breakpoints and instruction breakpoints.

For both data breakpoints and instruction breakpoints, there are two types of registers – address offset registers and address mask registers. The address offset and address mask registers are implemented in pairs – each offset register has a corresponding mask register. There are between 0 and 31 of each type of register. The number of registers is implementation dependent, but the implemented registers must be contiguous starting with register 0. The number of instruction register pairs and the number of data register pairs that exist in each processor are returned by the IDENTIFY DEBUG SFU instruction.

NOTE

The maximum number of registers that can be implemented in each register set is 31, not 32, because the IDENTIFY DEBUG SFU instruction only contains a five-bit field to return the number of implemented registers, and 0 indicates that no registers of that type exist.

The data and instruction breakpoint address offset registers (DBAORs and IBAORs) have the same format and are shown in Figure 8-1.

	0 … 31
DBAOR/IBAOR 0	address offset
DBAOR/IBAOR 1	address offset
DBAOR/IBAOR 2	address offset
	• • •
DBAOR/IBAOR 29	address offset
DBAOR/IBAOR 30	address offset

Figure 8-1. Data and Instruction Breakpoint Address Offset Registers

The breakpoint address offset registers (DBAORs and IBAORs) contain the absolute addresses of the breakpoints.

The data breakpoint address mask registers (DBAMRs) contain access-type enable bits and address mask bits for the breakpoints, as shown in Figure 8-2.

	0	1	2 … 7	8 … 31
DBAMR 0	r	w	rv	mask
DBAMR 1	r	w	rv	mask
DBAMR 2	r	w	rv	mask
			• • •	
DBAMR 29	r	w	rv	mask
DBAMR 30	r	w	rv	mask

Figure 8-2. Data Breakpoint Address Mask Registers

The *r* and *w* bits in each DBAMR determine the type of access this data breakpoint is enabled for. If the *r* bit is 1, any non-nullified load or semaphore instruction to an address matching the corresponding DBAOR will cause a data debug trap (if enabled in the PSW). If the *w* bit is 1, any non-nullified store or semaphore instruction or cache purge operation to an address matching the corresponding DBAOR will cause a data debug trap (if enabled in the PSW). If the *r* and *w* bits are both 0, that data breakpoint is disabled. The data address breakpoint facility does not apply to the flush, probe, insert TLB, purge TLB, LPA, or LCI instructions, nor to cache purge instructions which perform a flush operation.

The *mask* field in each DBAMR determines which of the address bits in the corresponding DBAOR will be compared to determine if there is a match. Each address bit whose corresponding *mask* bit is 0 must match for the breakpoint to be signalled. Each address bit whose corresponding *mask* bit is 1 is ignored in the comparison. The *mask* field may contain between 0 and 24 bits. Each implementation may choose the number of *mask* bits to implement – the number that exist in each processor is returned by the SFU identify instruction. The *mask* bits that are implemented must be contiguous and begin at bit position 31. The *mask* bits that are not implemented, as well as the bits marked *rv* in the mask registers are reserved bits. Address bits for which there are no corresponding *mask* bits are always compared. If the *mask* field of a mask register has any 1 bits, the corresponding address register must have a 0 in those bit positions – not doing so is an undefined operation. For example, a *mask* value of 0 means that all 32 bits of the address are compared while *mask* value of 0xFFF causes all addresses within the specified 4KB page to match.

The instruction breakpoint address mask registers are shown in Figure 8-3.

	0	1 7	8 31
IBAMR 0	e	rv	mask
IBAMR 1	e	rv	mask
IBAMR 2	e	rv	mask
		• • •	
IBAMR 29	e	rv	mask
IBAMR 30	e	rv	mask

Figure 8-3. Instruction Breakpoint Address Mask Registers

The *e* bit in each IBAMR determines whether this instruction breakpoint is enabled. If the *e* bit is 1, any attempt to execute a non-nullified instruction at an address matching the corresponding IBAOR will cause an instruction debug trap (if enabled in the PSW). Note that all 32 bits (including the privilege level bits) are included in the instruction address comparison, unless masked as described below. If the *e* bit is 0, that instruction breakpoint is disabled.

The *mask* field in each IBAMR determines which of the address bits in the corresponding IBAOR will be compared to determine if there is a match. The *mask* field operates exactly as described earlier for the *mask* field in the DBAMRs.

The Instruction Set

The debug SFU instructions are implemented within the generic SFU instruction framework described in “Special Function Unit (SFU) Instructions” on page 5-177. Eight debug SFU instructions allow unprivileged access to the debug SFU registers. A ninth new instruction provides the required IDENTIFY SFU operation which includes fields identifying the version of the debug SFU, the number of data breakpoint address offset registers (and thus also the number of data mask registers), the number of instruction breakpoint address offset registers (and thus also the number of instruction mask registers), and the width of the *mask* field in each mask register, that the processor implements. All nine instructions must be implemented if the debug SFU is implemented.

Figure 8-4 shows the formats of the debug SFU instructions. The description, operation, mnemonics, and encodings of the debug SFU instructions are summarized in Table 8-1.

Debug Operation One: 1 debug register source, 1 general register destination

04	r	rv	sub	1	1	0	t
6	5	7	3	2	3	1	5

Debug Operation Two: 1 general register source, 1 debug register destination

04	r	rv	sub	2	1	0	t
6	5	7	3	2	3	1	5

Figure 8-4. Debug SFU Instruction Formats

Table 8-1. Debug SFU Instructions

Opcode	Format	Sub-op	Mnemonic	Description	Operation
04	1	0	DEBUGID	Identify debug SFU	GR[t] ← id number
		4	MFDBAO	Move from data breakpoint address offset register	GR[t] ← DBAOR[r]
		5	MFDBAM	Move from data breakpoint address mask register	GR[t] ← DBAMR[r]
		6	MFIBAO	Move from instruction breakpoint address offset register	GR[t] ← IBAOR[r]
		7	MFIBAM	Move from instruction breakpoint address mask register	GR[t] ← IBAMR[r]
04	2	4	MTDBAO	Move to data breakpoint address offset register	DBAOR[t] ← GR[r]
		5	MTDBAM	Move to data breakpoint address mask register	DBAMR[t] ← GR[r]
		6	MTIBAO	Move to instruction breakpoint address offset register	IBAOR[t] ← GR[r]
		7	MTIBAM	Move to instruction breakpoint address mask register	IBAMR[t] ← GR[r]

When a debug SFU instruction is executed and SCR{1} is 0, the instruction causes an assist emulation trap. It is an undefined operation to set SCR{1} to 1 if the debug SFU is nonexistent.

Interruptions

Two interruptions are associated with the debug SFU. The instruction debug trap is triggered whenever the instruction address matches the parameters set up in the debug SFU, the PSW G-bit is 1, and the PSW Z-bit is 0. The data debug trap is triggered whenever the data address of a load, store, or semaphore instruction, or a cache purge operation, matches the parameters set up in the debug SFU, the PSW G-bit is 1, and the PSW Y-bit is 0. See "Interruptions" on page 4-13 for additional details.

Changes to the debug SFU state are not always required to affect whether the immediately following instructions will trap or not. If the state change affects whether a subsequent instruction would take a data debug trap, then the change is required to affect the immediately following instruction. However, if the change affects whether a subsequent instruction would take an instruction debug trap, the change is not required to affect any instructions until the 8th instruction following the MTIBAO or MTIBAM which changed the state.

Changes to the PSW G-bit are also not always required to affect whether the immediately following instructions will trap or not. If the change is made via the system mask instructions, and it affects whether a subsequent instruction would take an instruction debug trap, the change is not required to affect any instructions until the 8th instruction following the system mask instruction which changed the state. Changes to the G-bit due to interruptions or return from interruption instructions, and changes which affect data debug traps are required to affect the immediately following instruction.

All changes to the PSW Y-bit and PSW Z-bit (due to interruptions or return from interruption instructions) are required to affect the immediately following instructions.

Instruction Set Description

The following pages describe the debug SFU instructions.

IDENTIFY DEBUG SFU DEBUGID

Format: DEBUGID t

04	0	0	0	1	1	0	t
6	5	7	3	2	3	1	5

Purpose: To identify the particular implementation of the debug SFU and its capabilities.

Description: The SFU id number is copied into GR *t*. The format of the id number is shown below.

model	revision	rv	ver	ireg	dreg	mask
6	5	3	3	5	5	5

The *model* field returns the implementation-dependent model number. The *revision* field returns the implementation-dependent revision number. The *ver* field returns the version of the debug SFU architecture to which this implementation conforms. The only value of *ver* currently defined is 0. The *ireg* field returns the number of instruction breakpoint address offset registers that have been implemented. The *dreg* field returns the number of data breakpoint address offset registers that have been implemented. Either *ireg* or *dreg* can be 0 indicating that none of that type of register has been implemented. The *mask* field returns the width of the implemented address masks. A *mask* value of 0 indicates that address masking is not supported.

If the debug SFU is not present, an assist emulation trap occurs. The assist emulation trap handler is required to return 0 as the identification number.

Operation: GR[t] ← id number;

Exceptions: Assist emulation trap

Notes: The DEBUGID pseudo operation generates aSPOP1,1,0 t instruction.

MOVE FROM DATA BREAKPOINT ADDRESS MASK REGISTER — MFDBAM

Format: MFDBAM r,t

04	r	rv	5	1	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value to a general register from a data breakpoint address mask register.

Description: DBAMR *r* is copied into GR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing a DBAMR beyond the number implemented is an undefined operation.

Operation: GR[t] ← DBAMR[r];

Exceptions: Assist emulation trap

MOVE FROM DATA BREAKPOINT ADDRESS OFFSET REGISTER — MFDBAO

Format: MFDBAO r,t

04	r	rv	4	1	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value to a general register from a data breakpoint address offset register.

Description: DBAOR *r* is copied into GR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing a DBAOR beyond the number implemented is an undefined operation.

Operation: GR[t] $\leftarrow$ DBAOR[r];

Exceptions: Assist emulation trap

MOVE FROM INSTRUCTION BREAKPOINT ADDRESS MASK REGISTER — MFIBAM

Format: MFIBAM r,t

04	r	rv	7	1	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value to a general register from an instruction breakpoint address mask register.

Description: IBAMR *r* is copied into GR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing an IBAMR beyond the number implemented is an undefined operation.

Operation: GR[t] $\leftarrow$ IBAMR[r];

Exceptions: Assist emulation trap

MOVE FROM INSTRUCTION BREAKPOINT ADDRESS OFFSET REGISTER — MFIBAO

Format: MFIBAO r,t

04	r	rv	6	1	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value to a general register from an instruction breakpoint address offset register.

Description: IBAOR *r* is copied into GR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing an IBAOR beyond the number implemented is an undefined operation.

Operation: GR[t] $\leftarrow$ IBAOR[r];

Exceptions: Assist emulation trap

MOVE TO DATA BREAKPOINT ADDRESS MASK REGISTER — MTDBAM

Format: MTDBAM r,t

04	r	rv	5	2	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value from a general register to a data breakpoint address mask register.

Description: GR *r* is copied into DBAMR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing a DBAMR beyond the number implemented is an undefined operation.

Operation: DBAMR[t] $\leftarrow$ GR[r];

Exceptions: Assist emulation trap

MOVE TO DATA BREAKPOINT ADDRESS OFFSET REGISTER — MTDBAO

Format: MTDBAO r,t

04	r	rv	4	2	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value from a general register to a data breakpoint address offset register.

Description: GR *r* is copied into DBAOR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing a DBAOR beyond the number implemented is an undefined operation.

Operation: DBAOR[t] ← GR[r];

Exceptions: Assist emulation trap

MOVE TO INSTRUCTION BREAKPOINT ADDRESS MASK REGISTER — MTIBAM

Format: MTIBAM r,t

04	r	rv	7	2	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value from a general register to an instruction breakpoint address mask register.

Description: GR *r* is copied into IBAMR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing an IBAMR beyond the number implemented is an undefined operation.

Operation: IBAMR[t] $\leftarrow$ GR[r];

Exceptions: Assist emulation trap

MOVE TO INSTRUCTION BREAKPOINT ADDRESS OFFSET REGISTER — MTIBAO

Format: MTIBAO r,t

04	r	rv	6	2	1	0	t
6	5	7	3	2	3	1	5

Purpose: To move a value from a general register to an instruction breakpoint address offset register.

Description: GR *r* is copied into IBAOR *t*.

If the debug SFU is not present, an assist emulation trap occurs.

Accessing an IBAOR beyond the number implemented is an undefined operation.

Operation: IBAOR[t] ← GR[r];

Exceptions: Assist emulation trap

A Glossary

Absolute Address

See Physical Address.

Access Rights

A function of virtual address translation that controls access to each page through privilege levels for read, write, execute, and gateway. The TLB contains, within each entry, information used to determine who may have access to that page. This information is divided into two groups: (1) page access (access ID) which is used to determine if a process or user may access a page; and (2) the access rights field that is combined with the user's privilege level to determine if the type of access the user is requesting will be allowed.

Address

PA-RISC is a byte-addressable system which uses both virtual and absolute addresses. A virtual address can be split into two parts: the high-order bits which are the space identifier and the 32 low-order bits that give the offset within the space. Absolute addresses do not have space identifiers; only a 32-bit offset. Doublewords, words, and halfwords are always located at addresses which are aligned to their size (in bytes). Quadwords are aligned on doubleword boundaries.

Address Translation

For a virtual memory system, the process whereby the virtual (logical) address of data or instructions is translated to its absolute address in physical memory.

Aliasing

The condition when the same physical memory location is accessed by different virtual addresses or by both an absolute and a virtual address.

Alter

The action of setting the E-bit of a TLB entry to 0 and modifying some portion of the physical page number field. Altered entries in the TLB are still visible to software through the insert TLB protection instructions.

Architecture

Refers to the time independent functional appearance of a computer system. An implementation of an architecture is an ensemble of hardware, firmware, and software that provides all the functions as defined in the architecture.

Arithmetic and Logical Unit (ALU)

The part of a PA-RISC processor that performs arithmetic and logic operations on its inputs, producing output and status information.

Assist Processor

A processor which may be added to the basic PA-RISC system to enhance performance or functionality for algorithms which experience substantial gains from the use of specialized hardware. Assist processors are differentiated by the level at which they interface with the memory hierarchy. (See special function units and coprocessors).

B-bit (Taken Branch in Previous Cycle)

A bit in the PSW that is 1 if the previous instruction was a taken branch.

Base Register

A register that holds the numeric value that is used as a base value in the calculation of addresses. Displacements or index values are added to this base value.

Base-Relative Branch

When a general register is used as the base offset to obtain the target address, the branch is called base relative.

Biased Exponent

The exponent field for a floating-point number. It consists of the exponent plus the bias.

Binary Floating-point Number

A number format consisting of the three components: sign, exponent, and significand.

Block TLB

A block TLB provides address translations which map address ranges larger than a page.

Byte

A group of eight contiguous bits which is the smallest addressable unit on a PA-RISC system.

C-bit (Code Address Translation Enable)

A bit in the PSW that specifies whether virtual address translation of the instruction address is to be performed.

Cache

A high-speed buffer unit between main memory and the CPU. The cache is continually updated to contain recently accessed contents of main memory to reduce access time. When a program makes a memory request, the CPU first checks to see if the data is in the cache so that it can be retrieved without accessing memory. There may be one cache for both instructions and data or separate caches for each.

Cache Coherence

The property of multiple caches whereby they provide identical shared memory images. Processors in a multiprocessor system are said to be cache coherent if they provide the image of single cache.

Cache Control Hint

A 2-bit field in some memory reference instructions which provides a hint to the processor on how to resolve cache coherence. The processor may disregard the hint without compromising system integrity, but performance may be enhanced by following the hint.

Cache Miss

A cache miss occurs when the cache does not contain a copy of the cache line being requested by the address. The cache is updated with data and re-accessed.

Carry/Borrow Bits

An 8-bit field in the PSW that indicates if a carry or borrow occurred from the corresponding nibble (4 bits) as a result of the previous arithmetic operation.

Central Processing Unit (CPU)

The part of a PA-RISC processor that fetches and executes instructions.

Check

The interruption condition when the processor detects an internal or external malfunction. Checks may be either synchronous or asynchronous with respect to the instruction stream.

Coherence Check

An action taken by hardware to insure coherence.

Combined TLB

Some systems have a combined TLB which provides address translation for both instruction and data references.

Compatibility

The ability for software developed for one machine type to execute on another machine type. PA-RISC provides compatible execution of application programs written for earlier-generation Hewlett-Packard computer systems.

Completer

A machine instruction field used to specify instruction options. Typical options include address modification, address indexing, precision of operands, and conditions to be tested to determine whether to nullify the following instruction.

Condition

The state of a value or a relationship between values used in determining whether an instruction is to branch, nullify, or trap.

Control Register (CR)

A register which contains system state information used for memory access protection, interruption control, and processor state control. A PA-RISC processor contains 25 control registers (7 more are reserved).

Coprocessor

A type of assist processor which interfaces to the memory hierarchy at the level of the cache. Coprocessors are special purpose units that work with the main processor to speed up specialized operations such as floating-point arithmetic and graphics processing. Coprocessors generally have their own internal state and hardware evaluation mechanism.

Coprocessor Configuration Register (CCR)

The CCR (in CR 10) is an 8-bit register which records the presence and usability of coprocessors. Each bit position (0-7) corresponds to the coprocessor with the same unit number. Setting a bit in the CCR to 1 enables the use of the corresponding coprocessor, if present and operational. If a CCR bit is 0, the corresponding coprocessor, if present, is logically decoupled and an attempt to reference the coprocessor causes an assist emulation trap.

Current Instruction

The instruction whose address is in the front element of the instruction address queues (IASQ and IAOQ).

D-bit (Data Address Translation Enable)

A bit in the PSW that specifies whether virtual address translation of data addresses is to be performed.

Data Cache (D-cache)

A high-speed storage device which contains data items that have been recently accessed from main memory. The D-cache can be accessed independently of the instruction cache (I-cache) and no synchronization is performed.

Data TLB (DTLB)

A separate TLB which does address translation only for data.

Denormalized Numbers

Any non-zero floating-point number with the exponent field all zeros. Denormalized numbers are distinguished from normal numbers in that the value of the "hidden" bit to the left of the implied binary point is zero.

Dirty

A block of memory (commonly a cache line or a page) which has been written to is referred to as dirty.

Displacement

The amount that is added to a base register to form an offset in the virtual address computation.

Dynamic Displacement

If the displacement value is computed during the course of program execution and is obtained from a general register, it is called dynamic.

E-bit (Little Endian Memory Access Enable)

A bit in the PSW which determines whether memory references assume big endian or little endian byte ordering.

Effective Address

The address of the operand for the current instruction, derived by applying specific address building rules.

Equivalently Aliased

A condition when two virtual addresses map to the same physical address, and where the two addresses are identical in the following bits: Offset bits 12 through 31. If the use of space bits in generating the cache index is enabled, the addresses must also be identical in these bits: Space Identifier bits 4 through 7, 12 through 15, and 20 through 31.

Equivalently Mapped

A condition when a virtual address is equal to its absolute address.

Exponent

The part of a binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number.

External Branch Instructions

The target of these instructions may lie in a different address space than that of the instruction. The external branch instructions are: BE and BLE.

External Interrupt Enable Mask (EIEM)

The EIEM (CR 15) is a 32-bit register containing one bit for each external interrupt class. When set to 0, bits in the EIEM mask interruptions pending for the external interrupts corresponding to those bit positions.

External Interrupt Request Register (EIR)

The EIR register (CR 23) is a 32-bit register containing one bit for each external interrupt. When set to 1, a bit designates that an interruption is pending for the corresponding external interrupt.

F-bit (Performance Monitor Interruption Unmask)

A bit in the PSW used to unmask the performance monitor interruption.

Fault

The interruption condition when the current instruction requests a legitimate action which cannot be carried out due to a system problem such as the absence of a main memory page. After the system problem is cleared, the faulting instruction will execute normally. Faults are synchronous with respect to the instruction stream.

Floating-point Register (FPR)

A storage unit which constitutes the basic resource of the floating-point coprocessor. Floating-point registers are at the highest level of memory hierarchy and are used to load data from and store data to memory and hold operands and results of the floating-point coprocessor. The floating-point coprocessor contains 32 double-precision (64-bit) floating-point registers which may also be accessed as 64 single-precision (32-bit), or 16 quad-precision (128-bit) registers.

Following Instruction

The instruction whose address is in the back element of the instruction address queues (IASQ and IAOQ). This instruction will be executed after the current instruction. This instruction is not necessarily the next instruction in the linear code space.

Fraction

The portion of the significand explicitly contained in a binary floating-point number. The rest of the significand is the "hidden" bit to the left of the implied binary point. The "hidden" bit normally has the value one.

G-bit (Debug Trap Enable)

A bit in the PSW used to enable data and instruction debug traps.

General Register (GR)

A storage unit which constitutes the basic resource of the CPU. General registers are at the highest level of memory hierarchy and are used to load data from and store data to memory and hold operands and results from the ALU. A PA-RISC processor contains 32 general registers.

H-bit (Higher Privilege Transfer Trap Enable)

A bit in the PSW that enables an interruption whenever the following instruction will execute at a higher privilege level.

High-Priority Machine Check (HPMC)

An interruption which occurs when a hardware error has been detected which requires immediate attention.

I-bit (External, Power Failure, and LPMC Interruption Unmask)

A bit in the PSW used to unmask external interrupts, power failure interrupts, and low-priority machine check interruptions.

IAOQ (Instruction Address Offset Queue)

A two-element queue of 32-bit registers that is used to hold the Instruction Address offset (IA offset). The first element is IAOQ_Front and holds the IA offset of the current instruction. The other element is IAOQ_Back and holds the IA offset of the following instruction.

IA-Relative Branches

When a displacement is added to the current Instruction Address offset (IA offset) to obtain the target address, the branch is called IA relative.

IASQ (Instruction Address Space Queue)

A two-element queue of 16-, 24-, or 32-bit registers that is used to hold the Instruction Address space (IA space). The first element is IASQ_Front and holds the IA space of the current instruction. The other element is IASQ_Back and holds the IA space of the following instruction.

IIAOQ (Interruption Instruction Address Offset Queue)

A two-element queue of 32-bit registers that is used to save the Instruction Address offset for use in processing interruptions.

IIASQ (Interruption Instruction Address Space Queue)

A two-element queue of 16-, 24-, or 32-bit registers that is used to save the Instruction Address space for use in processing interruptions.

Infinity

The binary floating-point numbers that have all ones in the exponent and all zeros in the fraction. The values of these two numbers are distinguished only by the sign. Thus, they are $+\infty$ and $-\infty$.

Instruction Cache (I-cache)

A high-speed storage device that contains instructions that have been recently accessed from main memory. The I-cache can be accessed independently of the data cache (D-cache) and no synchronization is performed.

Instruction TLB (ITLB)

A separate TLB which does address translation only for instructions.

Interrupt

The interruption condition when an external entity (such as an I/O device or the power supply) requires attention. Interrupts are asynchronous with respect to the instruction stream.

Interruption

An event that changes the instruction stream to handle exceptional conditions including traps, checks, faults, and interrupts.

Interruption Instruction Register (IIR)

The IIR (CR 19) is used by the hardware to store the instruction that caused the interruption or the instruction that was in progress at the time the interruption occurred.

Interruption Offset Register (IOR)

The IOR (CR 21) receives a copy of the offset portion of a virtual address at the time of an interruption whenever the PSW Q-bit is 1. The value copied is dependent upon the type of interruption.

Interruption Parameter Registers (IPRs)

The Interruption Instruction Register or IIR (CR 19), Interruption Space Register or ISR (CR 20), and Interruption Offset Register or IOR (CR 21) are collectively termed the Interruption Parameter Registers or IPRs. They are used to pass the interrupted instruction and a virtual address to an interruption handler. These registers are set (or frozen) at the time of an interruption when the PSW Q-bit is 1. The IPRs can be read reliably only when the PSW Q-bit is 0. The values saved in these registers are dependent upon the type of interruption.

Interruption Processor Status Word (IPSW)

The IPSW (CR 22) receives the value of the PSW when an interruption occurs. The layout of IPSW is identical to that of PSW and it always reflects the machine state at the point of interruption.

Interruption Space Register (ISR)

The ISR (CR 20) receives a copy of the space portion of a virtual address at the time of an interruption whenever the PSW Q-bit is 1. The value copied is dependent upon the type of interruption.

Interruption Vector Address (IVA)

The IVA (CR 14) contains the absolute address of an array of service procedures assigned to interruptions.

Interspace Branches

When the target of the branch lies in a different address space as that of the branch instruction, it is referred to as an interspace branch.

Intraspace Branches

When the target of the branch lies in the same address space as that of the branch instruction, it is referred to as an intraspace branch.

Interval Timer

Two internal registers which are both accessed through Control Register 16. The Interval Timer is a free-running counter that signals an interruption when equal to a comparison value.

Invalidate

The action of setting the E-bit of a TLB entry to a 0, leaving the virtual page number and physical page number fields unchanged. Invalid entries in the TLB are still visible to software through insert TLB protection instructions.

L-bit (Lower Privilege Transfer Trap Enable)

A bit in the PSW that enables an interruption whenever the following instruction will execute at a lower privilege level.

Levels of Processor Architecture

Four levels of the processor architecture have been defined: 0, 1, 1.5, and 2. Level 0 systems support absolute memory addressing only; virtual memory is not supported, and so space identifiers are not used. Level 1, 1.5, and 2 systems have virtual addressing and differ only in the number of significant bits in their space identifiers. They have 2^{16}, 2^{24}, and 2^{32} virtual spaces, respectively.

Local Branch Instructions

The target of these instructions always lie in the same address space as that of the instruction. The local branch instructions are: BL, GATE, BLR, BV, MOVB, MOVIB, COMBT, COMBF, COMIBT, COMIBF, ADDBT, ADDBF, ADDIBT, ADDIBF, BB, and BVB.

Long Pointer

A virtual pointer which is made up of a space identifier and a 32-bit byte offset within the virtual space.

Low-Priority Machine Check (LPMC)

An interruption which occurs when a recoverable hardware error has been detected.

M-bit (High-Priority Machine Check Mask)

A bit in the PSW that disables the recognition of an HPMC.

Many-Reader/One-Writer Non-Equivalent Aliasing

A condition where multiple virtual addresses are non-equivalent aliases. Generally, before enabling a write-capable translation, any non-equivalent read-only aliases must be disabled, and the affected address range flushed from the cache. Similarly, before re-enabling the read translation(s), the write-capable translation must be disabled, and the affected address range flushed from the cache.

Memory

A device capable of storing information in binary form. The term "memory" typically refers to main memory.

Memory Address Space

The memory address space consists of absolute addresses in the range 0x00000000 through 0xEFFFFFFF.

Memory-mapped I/O

Control of input and output through load and store instructions to particular virtual or physical addresses.

Move-in

The action of bringing data or instructions into a cache.

Multiprocessor

A computer with multiple processors.

NaN

The binary floating-point numbers that have all ones in the exponent and a non-zero fraction. NaN is the term used for a binary floating-point number that has no value (i.e., "Not a Number"). The two types of NaNs, quiet and signaling, are distinguished by the value of the most significant bit in the fraction field. A zero indicates a quiet NaN and a one indicates a signaling NaN.

Non-Equivalently Aliased

A condition when two virtual addresses map to the same physical address, but do not meet the requirements for equivalently aliased addresses. (See “Equivalently Aliased” on page A-5.)

Nullify

To nullify an instruction is equivalent to skipping over that instruction. A nullified instruction has no effect on the machine state (except that the IA queues advance and the PSW B, N, X, Y, and Z bits are set to 0). The current instruction is nullified when the PSW N-bit is 1.

P-bit (Protection Identifier Validation Enable)

A bit in the PSW that is used as a protection identifier validation enable bit. If the P-bit is 1, the Protection Identifiers in control registers 8, 9, 12, and 13 are used to enforce protection.

Page

Virtual memory is partitioned into pages which can be resident in matching size blocks (called page frames) in memory. The page size is 4096 bytes (4 Kbytes).

Page Group

Eight contiguous pages, with the first of these pages beginning on a 32-Kbyte boundary.

Physical Address

The address that is the result of the virtual address translation or any address that is not translated. A physical address is the concatenation of the physical page number and the offset. Physical addresses are also referred to as absolute addresses.

Physical Page Directory (PDIR)

A table which is used to perform virtual address translations. The PDIR contains virtual address translations that either the TLB miss software or hardware will load into the TLB.

Privilege Level

The PA-RISC access control mechanisms are based on 4 privilege levels numbered from 0 to 3, with 0 being the most privileged. The current privilege level is maintained in the front element of the Instruction Address Offset Queue (IAOQ_Front).

Processor Status Word (PSW)

A 32-bit register which contains information about the processor state.

Q-bit (Interruption State Collection Enable)

A bit in the PSW that, when set to 1, enables collection of the machine state at the instant of interruption (IIASQ, IIAOQ, IIR, ISR, and IOR).

R-bit (Recovery Counter Enable)

A bit in the PSW that enables recovery counter trapping and decrementing of the Recovery Counter.

Read-Only Non-Equivalent Aliasing

A condition where multiple virtual addresses map to the same physical address, and where each virtual address has a read-only translation.

Read-Only Translation

A virtual address translation for which either the D-bit is equal to 0, or the page type in the access rights field is 0, 2, 4, 5, 6, or 7, in both the TLB and the page table. (See "Equivalently Aliased" on page A-5.)

Recovery Counter

The Recovery Counter (CR 0) counts down by 1 during execution of each non-nullified instruction for which the PSW R-bit is 1.

Remove

The action of taking a TLB entry out of the TLB. Insertion of translations into the TLB, for example, causes other entries to be removed.

S-bit (Secure Interval Timer)

A bit in the PSW that, when set to 1, allows the Interval Timer to be read only by code executing at the most privileged level.

SFU Configuration Register (SCR)

The SCR (in CR 10) is an 8-bit register which records the presence and usability of SFUs (Special Function Units), Each bit position (0-7) corresponds to the SFU with the same unit number. Setting a bit in the SCR to 1 enables the use of the corresponding SFU if present and operational. If a SCR bit is 0, the corresponding SFU if present, is logically decoupled and an attempt to reference the SFU causes an assist emulation trap.

Shadow Register (SHR)

A register into which the contents of a general register are copied upon interruptions. A PA-RISC processor contains 7 shadow registers which receive the contents of GRs 1, 8, 9, 16, 17, 24, and 25. The contents of the shadow registers are copied back to these GRs by the RETURN FROM INTERRUPTION AND RESTORE instruction.

Shift Amount Register (SAR)

The SAR (CR 11) is used by the variable shift, extract, deposit, and branch on bit instructions. It specifies the number of bits or the ending bit position of a quantity that is to be shifted, extracted or deposited.

Short Pointer

A 32-bit pointer used in virtual addressing. The two high-order bits point to one of four virtual address spaces. Following the determination of the virtual address space, all 32 bits are used to specify the byte offset within that space.

Sign

A one bit field in which one indicates a negative value and zero indicates a positive value.

Significand

The component of a binary floating-point number that consists of the implicit (or "hidden") leading bit to the left of the implied binary point together with the fraction field to its right.

Space Identifier (Space ID)

A 16-, 24-, or 32-bit value which occupies the upper portion of a virtual address and specifies the virtual space portion of the virtual address.

Space Register (SR)

A register used to specify the space identifier for virtual addressing. A PA-RISC processor contains 8 space registers.

Special Function Unit (SFU)

A type of assist processor which interfaces to the memory hierarchy at the general register level. It acts as an alternate ALU for the main processor and may have its own internal state.

Static Displacement

If the displacement is a fixed value that is known at compile time, it is called static.

Strong Ordering

The property that accesses to storage, such as loads and stores, appear to software to be done in program order. In multiprocessing systems, strong ordering means that accesses by a given processor appear to that processor as well as to all other processors in the system, to be done in program order.

System Mask

The G, F, R, Q, P, D, and I bits of the PSW are known as the system mask. Each of these bits, with the exception of the Q-bit, may be set to 1, set to 0, written, and read by the system control instructions that manipulate the system mask.

T-bit (Taken Branch Trap Enable)

A bit in the PSW that enables the taken branch trap.

Taken Branch

Conditional branches are considered to be "taken" if the specified condition is met. Unconditional branches are always "taken".

TLB Entry

A virtual to physical address translation, either valid or invalid, which is present in the TLB. Entries are visible to software through either references (such as loads, stores, and semaphores) or insert TLB protection instructions (IITLBP and IDTLBP).

TLB Miss Handling

The action taken, either by hardware or software, on a TLB miss. This involves inserting the missing translation into the proper TLB.

TLB Miss

The condition when there is no entry in the TLB matching the current virtual page number. In this case, the TLB is updated either by software or by hardware.

TLB Slot

A hardware resource in the TLB which holds a TLB entry.

Translation Lookaside Buffer (TLB)

A hardware unit which serves as a cache for virtual-to-absolute memory address mapping. When a memory reference is made to a given virtual address, the virtual page number is passed to the TLB and the TLB is searched for an entry matching the virtual page number. If the entry exists, the 20-bit absolute page number (contained in the entry) is concatenated with the 12-bit page offset from the original virtual address to form a 32-bit absolute address.

Trap

The interruption condition when either (1) the function requested by the current instruction cannot or should not be carried out, or (2) system intervention is requested by the user before or after the instruction is executed.

Virtual Addressing

A capability that eliminates the need to assign programs to fixed locations in main memory. Addresses supplied by a program are treated as logical addresses which are translated to absolute addresses when physical memory is addressed.

Write-Capable Translation

A translation which does not meet the requirement of a read-only translation. (See “Read-Only Translation” on page A-11.)

Write Disable (WD) Bit

The low-order bit of each of the four protection identifiers (PIDs) which, when 1, disables the use of that PID for validating write accesses.

X-bit (Data Memory Break Disable)

A bit in the PSW that disables the data memory break trap if equal to 1. A data memory break trap happens if a write is attempted to a page whose TLB B-bit is 1.

Y-bit (Data Debug Trap Disable)

A bit in the PSW that disables the data debug trap if equal to 1. A data debug trap happens if a memory reference is performed to an address which matches an enabled data breakpoint.

Z-bit (Instruction Debug Trap Disable)

A bit in the PSW that disables the instruction debug trap if equal to 1. An instruction debug trap happens if an attempt is made to execute an instruction at an address which matches an enabled instruction breakpoint.

B Instruction Index

C Instruction Formats

The PA-RISC instruction formats are shown below. The most general form of each format is given. Individual instructions in each class may have reserved or zero fields in place of one or more of the fields shown.

1. Loads and Stores, Load and Store Word Modify, Load Offset

op	b	t/r	s	im14
6	5	5	2	14

2. Indexed Loads

op	b	x	s	u	0	cc	ext4	m	t
6	5	5	2	1	1	2	4	1	5

3. Short Displacement Loads

op	b	im5	s	a	1	cc	ext4	m	t
6	5	5	2	1	1	2	4	1	5

4. Short Displacement Stores, Store Bytes Short

op	b	r	s	a	1	cc	ext4	m	im5
6	5	5	2	1	1	2	4	1	5

5. Long Immediates

op	t/r	im21
6	5	21

6. Arithmetic/Logical

op	r2	r1	c	f	ext6	0	t
6	5	5	3	1	6	1	5

7. Arithmetic Immediate

op	r	t	c	f	e	im11
6	5	5	3	1	1	11

8. Extract

op	r	t	c	ext3	p	clen
6	5	5	3	3	5	5

9. Deposit

op	t	r/im5	c	ext3	cp	clen
6	5	5	3	3	5	5

10.Shift

op	r2	r1	c	ext3	cp	t
6	5	5	3	3	5	5

11.Conditional Branch

op	r2/p	r1/im5	c	w1	n	w
6	5	5	3	11	1	1

12.Branch External, Branch and Link External

op	b	w1	s	w2	n	w
6	5	5	3	11	1	1

13.Branch and Link, Gateway

op	t	w1	ext3	w2	n	w
6	5	5	3	11	1	1

14.Branch and Link Register, Branch Vectored

op	t/b	x	ext3	0	n	0
6	5	5	3	11	1	1

15.Data Memory Management, Probe

op	b	r/x/im5	s	ext8	m	t
6	5	5	2	8	1	5

16.Instruction Memory Management

op	b	r/x/im5	s	ext7	m	0
6	5	5	3	7	1	5

17.Break

op	im13	ext8	im5
6	13	8	5

18.Diagnose

op	im26
6	26

19.Move to/from Space Register

op	rv	r	s	ext8	t
6	5	5	3	8	5

20.Load Space ID

op	b	rv	s	0	ext8	t
6	5	5	2	1	8	5

21.Move to Control Register

op	t	r	rv	ext8	0
6	5	5	3	8	5

22.Move from Control Register

op	r	0	rv	ext8	t
6	5	5	3	8	5

23.System Control

op	b	r/im5	0	ext8	t
6	5	5	3	8	5

24.Special Operation Zero

op	sop1	0	sfu	n	sop2
6	15	2	3	1	5

25.Special Operation One

op	sop	1	sfu	n	t
6	15	2	3	1	5

26.Special Operation Two

op	r	sop1	2	sfu	n	sop2
6	5	10	2	3	1	5

27.Special Operation Three

op	r2	r1	sop1	3	sfu	n	sop2
6	5	5	5	2	3	1	5

28.Coprocessor Operation

op	sop1	uid	n	sop2
6	17	3	1	5

29.Coprocessor Indexed Loads

op	b	x	s	u	0	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

30.Coprocessor Indexed Stores

op	b	x	s	u	0	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

31.Coprocessor Short Displacement Loads

op	b	im5	s	a	1	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

32.Coprocessor Short Displacement Stores

op	b	im5	s	a	1	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

33.Floating-point Operation Zero, Major Opcode 0C

op	r	0	sop	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

34.Floating-point Operation One, Major Opcode 0C

op	r	0	sop	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

35.Floating-point Operation Two, Major Opcode 0C

op	r1	r2	sop	fmt	2	0	n	c
6	5	5	3	2	2	3	1	5

36.Floating-point Operation Three, Major Opcode 0C

op	r1	r2	sop	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

37.Floating-point Operation Zero, Major Opcode 0E

op	r	0	sop	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

38.Floating-point Operation One, Major Opcode 0E

op	r	0	sop	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

39.Floating-point Operation Two, Major Opcode 0E

op	r1	r2	sop	r2	f	2	0	r1	0	0	c
6	5	5	3	1	1	2	1	1	1	1	5

40.Floating-point Operation Three, Major Opcode 0E

op	r1	r2	sop	r2	f	3	x	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

41.Floating-point Multiple-operation

op	rm1	rm2	ta	ra	f	tm
6	5	5	5	5	1	5

The field names used in the previous instruction format layouts are described in the following table. Some of the field names may be followed by one or two digits. Those digits indicate the length of the field. An example of a field name may be *im5* which indicates the field is a 5-bit immediate value. But names, such as *r1*, which refers to the first source register field, are the actual field names.

Field	**Description**
a	modify before/after bit
b	base register
c	condition specifier
cc	cache control hint
clen	31 - extract/deposit length
cp	31 - bit position
df	floating-point destination format
e or ext	operation code extension
f	condition negation bit
f or fmt	floating-point data format
im	immediate value
m	modify bit
n	nullify bit
op	operation code
p	extract/deposit/shift bit position
r, r1, or r2	source register
ra, rm1, or rm2	floating-point multiple-operation source register
rv	reserved instruction field
s	2 or 3 bit space register
sf	floating-point source format
sfu	special function unit number
sop, sop1, or sop2	special function unit or coprocessor operation
t, ta, or tm	target register
u	shift index bit
uid	coprocessor unit identifier
w, w1, or w2	word offset/word offset part
x	index register

D Operation Codes

Major Opcode Assignments

The major opcode assignments are listed in Table D-1. Instructions are shown in uppercase. Instruction classes are capitalized. Extensions of the major opcodes can be found in the tables indicated, where applicable. In the following discussions of opcode extensions the major opcode class names are shown in parentheses.

Table D-1. Major Opcode Assignments

bits 2:5	bits 0:1			
	0	1	2	3
0	System_op (Table D-2)	LDB	COMBT	BVB
1	Mem_Mgmt (Tables D-3 and D-4)	LDH	COMIBT	BB
2	Arith/Log (Table D-5)	LDW	COMBF	MOVB
3	Index_Mem (Table D-6)	LDWM	COMIBF	MOVIB
4	SPOPn (Table D-11)	—	COMICLR	Extract (Table D-8)
5	DIAG	—	Subi (Table D-7)	Deposit (Table D-8)
6	FMPYADD	—	FMPYSUB	—
7	—	—	—	—
8	LDIL	STB	ADDBT	BE
9	Copr_w (Table D-10)	STH	ADDIBT	BLE
A	ADDIL	STW	ADDBF	Branch (Table D-9)
B	Copr_dw (Table D-10)	STWM	ADDIBF	—
C	COPR	—	Addit (Table D-7)	—
D	LDO	—	Addi (Table D-7)	—
E	Float (Tables D-16 through D-20)	—	—	—
F	Product Specific	—	—	—

Opcode Extension Assignments

Many instructions require both a major opcode and an opcode extension to be uniquely identified. The extension can be one to nine bits, depending on the major opcode.

System Control Instructions (System_op)

Figure D-1 shows the format of the system control instructions (major opcode 00) and Table D-2 lists the opcode extensions. Bits 19:21 encode the source of the operation and bits 24:26 encode the destination.

0	5			19	26
00	b/r	ext5	0/s	ext8	t
6	5	5	3	8	5

Figure D-1. Format for System Control Instructions

Table D-2. System Control Instructions

Instruction	Opcode	Extension				
	hex	binary			hex	
	bits 0:5	bits 19:21	bits 22:23	bits 24:26	bits 19:26	bits 11:15
BREAK	00	000	00	000	00	im5
SYNC	00	001	00	000	20	0
SYNCDMA	00	001	00	000	20	10
RFI	00	011	00	000	60	rv
RFIR	00	011	00	101	65	rv
SSM	00	011	01	011	6B	i
RSM	00	011	10	011	73	i
MTSM	00	110	00	011	C3	r
LDSID	00	100	00	101	85	rv
MTSP	00	110	00	001	C1	r
MFSP	00	001	00	101	25	0
MTCTL	00	110	00	010	C2	r
MFCTL	00	010	00	101	45	0

Bits	Value	Description
19:21 / 24:26	000	no source / no destination
	001	system resource
	010	control register
	011	PSW system mask
	100	space register
	101	general register destination
	110	general register source
22:23	01	encodes SSM
	10	encodes RSM

Memory Management Instructions (Mem_Mgmt)

Figure D-2 shows the format of the memory management instructions (major opcode 01). The opcode extensions (bits 19:26) for instruction memory management instructions are listed in Table D-3. The opcode extensions (bits 18:26) for data memory management instructions are listed in Table D-4 on page D-6. This group includes instructions that access the translation lookaside buffers and the caches.

Instruction Memory Management

0	5			19	25	27 31
01	b	r/x/im5	s	ext7	m	ext5
6	5	5	3	7	1	5

Data Memory Management, Probe

0	5			18	25	27 31
01	b	r/x/im5	s	ext8	m	ext5
6	5	5	2	8	1	5

Figure D-2. Formats for Memory Management Instructions

Table D-3. Instruction Memory Management Instructions

Instruction	Opcode	Extension						Modify
	hex	binary				hex		binary
	bits 0:5	bit 19	bits 20:21	bits 22:24	bit 25	bits 19:25	bits 27:31	bit 26
IITLBA	01	0	00	000	1	01	0	0
IITLBP	01	0	00	000	0	00	0	0
PITLB	01	0	00	100	0	08	rv	m
PITLBE	01	0	00	100	1	09	rv	m
FIC	01	0	00	101	0	0A	rv	m
FICE	01	0	00	101	1	0B	rv	m

Bits	**Value**	**Description**
19	0	instruction memory management
22:24	000	insert instruction
	100	purge TLB instruction
	101	flush instruction
22	1	modify (bit 26) enable
24	1	nonprivileged instruction
26	*m*	modification is allowed for this instruction

Table D-4. Data Memory Management Instructions

Instruction	Opcode	Extension							Modify
	hex	binary					hex		binary
	bits 0:5	bit 18	bit 19	bits 20:21	bits 22:24	bit 25	bits 18:25	bits 27:31	bit 26
IDTLBA	01	0	1	00	000	1	41	0	0
IDTLBP	01	0	1	00	000	0	40	0	0
PDTLB	01	0	1	00	100	0	48	rv	m
PDTLBE	01	0	1	00	100	1	49	rv	m
FDC	01	0	1	00	101	0	4A	rv	m
FDCE	01	0	1	00	101	1	4B	rv	m
PDC	01	0	1	00	111	0	4E	0	m
PROBER	01	0	1	00	011	0	46	t	0
PROBERI	01	1	1	00	011	0	C6	t	0
PROBEW	01	0	1	00	011	1	47	t	0
PROBEWI	01	1	1	00	011	1	C7	t	0
LPA	01	0	1	00	110	1	4D	t	m
LCI	01	0	1	00	110	0	4C	t	0

Bits	Value	Description
18	0	non-immediate value
	1	immediate value
19	1	data memory management
22:24	000	insert instruction
	011	probe instruction
	100	purge TLB instruction
	101	flush instruction
	110	load instruction
	111	purge cache instruction
22	1	modify (bit 26) enable
23	1	store result
24	1	nonprivileged instruction
26	*m*	modification is allowed for this instruction

Arithmetic/Logical Instructions (Arith/Log)

Figure D-3 shows the format of the arithmetic/logical instructions. The opcode extensions for the arithmetic/logical instructions (major opcode 02) are listed in Table D-5.

0	5				20	25	
02	r2	r1	c	f	ext6	0	t
6	5	5	3	1	6	1	5

Figure D-3. Format for Arithmetic/Logical Instructions

Table D-5. Arithmetic/Logical Instructions

Instruction	Opcode	Extension		
	hex	binary		hex
	bits 0:5	bits 20:21	bits 22:25	bits 20:25
ADD	02	01	1000	18
ADDO	02	11	1000	38
ADDC	02	01	1100	1C
ADDCO	02	11	1100	3C
SH1ADD	02	01	1001	19
SH1ADDO	02	11	1001	39
SH2ADD	02	01	1010	1A
SH2ADDO	02	11	1010	3A
SH3ADD	02	01	1011	1B
SH3ADDO	02	11	1011	3B
SUB	02	01	0000	10
SUBO	02	11	0000	30
SUBT	02	01	0011	13
SUBTO	02	11	0011	33
SUBB	02	01	0100	14
SUBBO	02	11	0100	34
DS	02	01	0001	11
ANDCM	02	00	0000	00
AND	02	00	1000	08
OR	02	00	1001	09
XOR	02	00	1010	0A
UXOR	02	00	1110	0E
COMCLR	02	10	0010	22
UADDCM	02	10	0110	26
UADDCMT	02	10	0111	27
ADDL	02	10	1000	28

Table D-5. Arithmetic/Logical Instructions (Continued)

Instruction	Opcode	Extension		
	hex	binary		hex
	bits 0:5	bits 20:21	bits 22:25	bits 20:25
SH1ADDL	02	10	1001	29
SH2ADDL	02	10	1010	2A
SH3ADDL	02	10	1011	2B
DCOR	02	10	1110	2E
IDCOR	02	10	1111	2F

Bits	Value	Description
20:21	00	unit/logical; do not set carry/borrow bits.
	01	arithmetic; set carry/borrow bits; do not trap.
	10	unit/logical; do not set carry/borrow bits.
	11	arithmetic; set carry/borrow bits; trap on overflow.

Indexed and Short Displacement Load/Store Instructions (Index_Mem)

Figure D-4 shows the formats of the indexed and short displacement load and store instructions. The opcode extensions (bits 22:25) for indexed and short displacement memory reference instructions (major opcode 03) are listed in Table D-6. The short displacement forms are distinguished from the indexed instructions by bit 19 (0=indexed, 1=short).

Indexed Loads

0	5				19		22	25	
03	b	x	s	u	0	cc	ext4	m	t
6	5	5	2	1	1	2	4	1	5

Short Displacement Loads

0	5				19		22	25	
03	b	im5	s	a	1	cc	ext4	m	t
6	5	5	2	1	1	2	4	1	5

Short Displacement Stores, Store Bytes Short

0	5				19		22	25	
03	b	r	s	a	1	cc	ext4	m	im5
6	5	5	2	1	1	2	4	1	5

Figure D-4. Formats for Indexed and Short Displacement Load/Store Instructions

Table D-6. Indexed and Short Displacement Load/Store Instructions

Instruction	Opcode	Extension			
	hex	binary			hex
	bits 0:5	bit 19	bits 22:23	bits 24:25	bits 22:25
LDBX	03	0	00	00	0
LDHX	03	0	00	01	1
LDWX	03	0	00	10	2
LDWAX	03	0	01	10	6
LDCWX	03	0	01	11	7
LDBS	03	1	00	00	0
LDHS	03	1	00	01	1
LDWS	03	1	00	10	2
LDWAS	03	1	01	10	6
LDCWS	03	1	01	11	7
STBS	03	1	10	00	8
STHS	03	1	10	01	9
STWS	03	1	10	10	A
STBYS	03	1	11	00	C
STWAS	03	1	11	10	E

Arithmetic Immediate Instructions (Addit, Addi, Subi)

Figure D-5 shows the format of the arithmetic immediate instructions. The opcode extensions (bit 20) for the arithmetic immediate instructions (major opcodes 25, 2C, and 2D) are listed in Table D-7. The extension field, *e*, determines whether or not the instruction traps on overflow.

0	5				20	
op	r	t	c	f	e	im11
6	5	5	3	1	1	11

Figure D-5. Format for Arithmetic Immediate Instructions

Table D-7. Arithmetic Immediate Instructions

Instruction	Opcode	Extension
	hex	binary
	bits 0:5	bit 20
ADDI	2D	0
ADDIT	2C	0
SUBI	25	0
ADDIO	2D	1
ADDITO	2C	1
SUBIO	25	1

Extract and Deposit Instructions (Extract and Deposit)

Figure D-6 shows the formats of the extract and deposit instructions. The opcode extensions (bits 19:21) for the extract and deposit instructions (major opcodes 34 and 35) are listed in Table D-8.

Extract

0 5				19 21		
34	r2	t/r1	c	ext3	ext5	clen/t
6	5	5	3	3	5	5

Deposit

0 5				19 21		
35	t	r/im5	c	ext3	ext5	clen
6	5	5	3	3	5	5

Figure D-6. Formats for Extract and Deposit Instructions

Table D-8. Extract and Deposit Instructions

Instruction	Opcode	Extension		
	hex	binary	hex	binary
	bits 0:5	bits 19:21	bits 19:21	bits 22:26
VSHD	34	000	0	00000
SHD	34	010	2	cp
VEXTRU	34	100	4	00000
VEXTRS	34	101	5	00000
EXTRU	34	110	6	p
EXTRS	34	111	7	p
ZVDEP	35	000	0	00000
VDEP	35	001	1	00000
ZDEP	35	010	2	cp
DEP	35	011	3	cp
ZVDEPI	35	100	4	00000
VDEPI	35	101	5	00000
ZDEPI	35	110	6	cp
DEPI	35	111	7	cp

Unconditional Branch Instructions (Branch)

Figure D-7 shows the formats of the unconditional branch instructions. The opcode extensions (bits 16:18) for the unconditional branch instructions (major opcode 3A) are listed in Table D-9.

Branch and Link, Gateway

0 5			16 18			
3A	t	w1	ext3	w2	n	w
6	5	5	3	11	1	1

Branch and Link Register, Branch Vectored

0 5			16 18			
3A	t/b	x	ext3	0	n	0
6	5	5	3	11	1	1

Figure D-7. Formats for Unconditional Branch Instructions

Table D-9. Unconditional Branch Instructions

Instruction	Opcode	Extension	
	hex	binary	hex
	bits 0:5	bits 16:18	bits 16:18
BL	3A	000	0
BLR	3A	010	2
BV	3A	110	6
GATE	3A	001	1

Coprocessor Loads and Stores (Copr_w and Copr_dw)

Figure D-8 shows the formats of the coprocessor load and store instructions. The opcode extensions for the coprocessor memory reference instructions (major opcodes 09 and 0B) are listed in Table D-10. Opcode 09 indicates the instruction operates on word data (Copr_w). Opcode 0B indicates the instruction operates on doubleword data (Copr_dw). The short displacement forms are distinguished from the indexed instructions by bit 19 (0 = indexed; 1 = short) and loads from stores by bit 22 (0 = load; 1 = store).

Coprocessor Indexed Loads

0	5				19		22			
op	b	x	s	u	0	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

Coprocessor Indexed Stores

0	5				19		22			
op	b	x	s	u	0	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

Coprocessor Short Loads

0	5				19		22			
op	b	im5	s	a	1	cc	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

Coprocessor Short Stores

0	5				19		22			
op	b	im5	s	a	1	cc	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

Figure D-8. Formats for Coprocessor Load/Store Instructions

Table D-10. Coprocessor Load and Store Instructions

Instruction	Opcode	Extension	
	hex	binary	
	bits 0:5	bit 19	bit 22
CLDWX	09	0	0
CLDDX	0B	0	0
CSTWX	09	0	1
CSTDX	0B	0	1
CLDWS	09	1	0
CLDDS	0B	1	0
CSTWS	09	1	1
CSTDS	0B	1	1

Special Function Unit Instructions

Figure D-9 shows the formats of the special function unit instructions. The opcode extensions for the special function unit instructions (major opcode 04) are listed in Table D-11.

Special Operation Zero

0 5		21:22			
04	sop1	0	sfu	n	sop2
6	15	2	3	1	5

Special Operation One

0 5		21:22			
04	sop	1	sfu	n	t
6	15	2	3	1	5

Special Operation Two

0 5			21:22			
04	r	sop1	2	sfu	n	sop2
6	5	10	2	3	1	5

Special Operation Three

0 5				21:22			
04	r1	r2	sop1	3	sfu	n	sop2
6	5	5	5	2	3	1	5

Figure D-9. Formats for Special Function Unit (SFU) Instructions

Table D-11. Special Function Unit (SFU) Instructions

Instruction	Opcode	Extension	
	hex	binary	hex
	bits 0:5	bits 21:22	bits 21:22
SPOP0	04	00	0
SPOP1	04	01	1
SPOP2	04	10	2
SPOP3	04	11	3

Floating-Point Coprocessor Operation Instructions

Figures D-10 and D-11 show the formats of the floating-point coprocessor operation instructions. The opcode extensions for the floating-point coprocessor operation instructions (major opcode 0C, uid 0 and major opcode 0E) are listed in Tables D-12 through D-20.

Major Opcode 0C

Floating-Point Operation Zero

0 5 16 18 21:22 26

0C	r	0	sub	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

Floating-Point Operation One

0 5 15:16 21:22 26

0C	r	0	sub	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

Floating-Point Operation Two

0 5 16 18 21:22 26

0C	r1	r2	sub	fmt	2	0	n	c
6	5	5	3	2	2	3	1	5

Floating-Point Operation Three

0 5 16 18 21:22 26

0C	r1	r2	sub	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

Figure D-10. Formats for Floating-Point Operations - Major Opcode 0C

Table D-12. Floating-Point Class Zero - Major Opcode 0C Instructions

Instruction	Opcode	Uid	Class	Sub-Op	Cond/Targ	Nullify
	hex	hex	hex	hex	hex	binary
	bits 0:5	bits 23:25	bits 21:22	bits 16:18	bits 27:31	bit 26
COPR,0,0	0C	0	0	0	0	0
FCPY	0C	0	0	2	t	0
FABS	0C	0	0	3	t	0
FSQRT	0C	0	0	4	t	0
FRND	0C	0	0	5	t	0
Reserved	0C	0	0	6,7	–	0

Table D-13. Floating-Point Class One - Major Opcode 0C Instructions

Instruction	Opcode	Uid	Class	Sub-Op	Cond/Targ	Nullify
	hex	hex	hex	hex	hex	binary
	bits 0:5	bits 23:25	bits 21:22	bits 15:16	bits 27:31	bit 26
FCNVFF	0C	0	1	0	t	0
FCNVXF	0C	0	1	1	t	0
FCNVFX	0C	0	1	2	t	0
FCNVFXT	0C	0	1	3	t	0

Table D-14. Floating-Point Class Two - Major Opcode 0C Instructions

Instruction	Opcode	Uid	Class	Sub-Op	Cond/Targ	Nullify
	hex	hex	hex	hex	hex	binary
	bits 0:5	bits 23:25	bits 21:22	bits 16:18	bits 27:31	bit 26
FCMP	0C	0	2	0	c	0
FTEST	0C	0	2	1	c	1
Undefined	0C	0	2	2-7	–	0

Table D-15. Floating-Point Class Three - Major Opcode 0C Instructions

Instruction	Opcode	Uid	Class	Sub-Op	Cond/Targ	Nullify
	hex	hex	hex	hex	hex	binary
	bits 0:5	bits 23:25	bits 21:22	bits 16:18	bits 27:31	bit 26
FADD	0C	0	3	0	t	0
FSUB	0C	0	3	1	t	0
FMPY	0C	0	3	2	t	0
FDIV	0C	0	3	3	t	0
Reserved	0C	0	3	4-6	–	0
Undefined	0C	0	3	7	–	0

Major Opcode 0E (Float)

Floating-Point Operation Zero

0 5 16 18 21:22 26

0E	r	0	sub	fmt	0	0	r	t	0	t
6	5	5	3	2	2	1	1	1	1	5

Floating-Point Operation One

0 5 15:16 21:22 26

0E	r	0	sub	df	sf	1	0	r	t	0	t
6	5	4	2	2	2	2	1	1	1	1	5

Floating-Point Operation Two

0 5 16 18 21:22 26

0E	r1	r2	sub	r2	f	2	0	r1	0	0	c
6	5	5	3	1	1	2	1	1	1	1	5

Floating-Point Operation Three

0 5 16 18 21:22 26

0E	r1	r2	sub	r2	f	3	0	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

Fixed-Point Operation Three

0 5 16 18 21:22 26

0E	r1	r2	sub	r2	m	3	1	r1	t	0	t
6	5	5	3	1	1	2	1	1	1	1	5

Figure D-11. Formats for Floating-Point Operations - Major Opcode 0E

Table D-16. Floating-Point Class Zero - Major Opcode 0E Instructions

Instruction	Opcode	Class	Sub-Op	Fixed	Nullify	Format	Cond/Targ
	hex	hex	hex	binary	binary	hex	hex
	bits 0:5	bits 21:22	bits 16:18	bit 23	bit 26	bits 19:20	bits 25,27:31
Undefined	0E	0	0,1	0	0	–	–
FCPY	0E	0	2	0	0	fmt	t
FABS	0E	0	3	0	0	fmt	t
FSQRT	0E	0	4	0	0	fmt	t
FRND	0E	0	5	0	0	fmt	t
Reserved	0E	0	6,7	0	0	–	–

Table D-17. Floating-Point Class One - Major Opcode 0E Instructions

Instruction	Opcode	Class	Sub-Op	Fixed	Nullify	Format	Cond/Targ
	hex	hex	hex	binary	binary	hex	hex
	bits 0:5	bits 21:22	bits 15:16	bit 23	bit 26	bits 17:20	bits 25,27:31
FCNVFF	0E	1	0	0	0	df,sf	t
FCNVXF	0E	1	1	0	0	df,sf	t
FCNVFX	0E	1	2	0	0	df,sf	t
FCNVFXT	0E	1	3	0	0	df,sf	t

Table D-18. Floating-Point Class Two - Major Opcode 0E Instructions

Instruction	Opcode	Class	Sub-Op	Fixed	Nullify	Format	Cond/Targ
	hex	hex	hex	binary	binary	binary	hex
	bits 0:5	bits 21:22	bits 16:18	bit 23	bit 26	bit 20	bits 25,27:31
FCMP	0E	2	0	0	0	f	0,c
Undefined	0E	2	1-7	–	0	–	–

Table D-19. Floating-Point Class Three - Major Opcode 0E Instructions

Instruction	Opcode	Class	Sub-Op	Fixed	Nullify	Format	Cond/Targ
	hex	hex	hex	binary	binary	binary	hex
	bits 0:5	bits 21:22	bits 16:18	bit 23	bit 26	bit 20	bits 25,27:31
FADD	0E	3	0	0	0	f	t
FSUB	0E	3	1	0	0	f	t
FMPY	0E	3	2	0	0	f	t
FDIV	0E	3	3	0	0	f	t
Undefined	0E	3	4	0	0	–	–
Reserved	0E	3	5-6	0	0	–	–
Undefined	0E	3	7	0	0	–	–

Table D-20. Fixed-Point Class Three - Major Opcode 0E Instructions

Instruction	Opcode	Class	Sub-Op	Fixed	Nullify	Format	Cond/Targ
	hex	hex	hex	binary	binary	binary	hex
	bits 0:5	bits 21:22	bits 16:18	bit 23	bit 26	bit 20	bits 25,27:31
XMPYU	0E	3	2	1	0	0	t

Performance Monitor Coprocessor Instructions

Figure D-12 shows the format of the performance monitor coprocessor operation instructions. The opcode extensions for the performance monitor coprocessor instructions (major opcode 0C, uid 2) are listed in Table D-21.

0 5	18	22	26		
0C	rv	sub	2	n	rv
6	12	5	3	1	5

Figure D-12. Format for Performance Monitor Coprocessor Instructions

Table D-21. Performance Monitor Coprocessor Instructions

Instruction	Opcode hex bits 0:5	Uid hex bits 23:25	Sub-Op hex bits 18:22	Nullify binary bit 26
PMDIS	0C	2	1	0
PMENB	0C	2	3	0
Undefined	0C	2	0,2,4..1F	–

Debug SFU Instructions

Figure D-13 shows the format of the debug special function unit instructions. The opcode extensions for the debug SFU instructions (major opcode 04, uid 1) are listed in Table D-22.

Debug Operation One

0 5 18 20 23 25

04	r	rv	sub	1	1	0	t
6	5	7	3	2	3	1	5

Debug Operation Two

0 5 18 20 23 25

04	r	rv	sub	2	1	0	t
6	5	7	3	2	3	1	5

Figure D-13. Debug SFU Instruction Formats

Table D-22. Debug SFU Instructions

Instruction	Opcode	Uid	Class	Sub-Op
	hex	hex	hex	hex
	bits 0:5	bits 23:25	bits 21:22	bits 18:20
MTDBAO	04	1	1	4
MTDBAM	04	1	1	5
MTIBAO	04	1	1	6
MTIBAM	04	1	1	7
DEBUGID	04	1	2	0
MFDBAO	04	1	2	4
MFDBAM	04	1	2	5
MFIBAO	04	1	2	6
MFIBAM	04	1	2	7

E Level 0 Summary

The following list summarizes the differences between Level 0 and non-Level 0 processors.

- Only absolute memory addressing is supported; virtual memory is not supported, and so space identifiers are not used and Space Registers are nonexistent registers.
- PID Registers (CRs 8, 9, 12, and 13), IIASQ (CR 17), and ISR (CR 20) are nonexistent registers.
- The X, C, P, and D bits of the PSW and the IPSW are nonexistent bits.
- There are only two distinct privilege levels - 0 and non-zero; privilege levels 1, 2, and 3 are equivalent.
- There are no TLBs.
- The s-fields of all instructions are ignored.
- The debug SFU, PSW G-bit, PSW Y-bit, PSW Z-bit, data debug trap, and instruction debug trap are defined only for Level 0 systems.
- The following TLB related instructions are executed as null instructions at all privilege levels:

PURGE DATA TLB (PDTLB)
PURGE INSTRUCTION TLB (PITLB)
PURGE DATA TLB ENTRY (PDTLBE)
PURGE INSTRUCTION TLB ENTRY (PITLBE)
INSERT DATA TLB ADDRESS (IDTLBA)
INSERT INSTRUCTION TLB ADDRESS (IITLBA)
INSERT DATA TLB PROTECTION (IDTLBP)
INSERT INSTRUCTION TLB PROTECTION (IITLBP)

- The following interruptions do not occur.

Interruption Name	Number
Instruction TLB miss fault/instruction page fault	6
Instruction memory protection trap	7
Data TLB miss fault/Data page fault	15
Non-access instruction TLB miss fault	16
Non-access data TLB miss fault/non-access data page fault	17
Data memory access rights trap	26
Data memory protection ID trap	27
Unaligned data reference trap	28
Data memory protection trap/Unaligned data reference trap	18

Interruption Name	Number
Data memory break trap	19
TLB dirty bit trap	20
Page reference trap	21

- The results of executing the following instructions are different; see the corresponding instruction page for details.

Instruction	Difference
LOAD PHYSICAL ADDRESS (LPA)	Undefined instruction
LOAD COHERENCE INDEX (LCI)	Undefined instruction
LOAD WORD ABSOLUTE INDEXED (LDWAX)	Same as LDWX if priv == 0
LOAD WORD ABSOLUTE SHORT (LDWAS)	Same as LDWS if priv == 0
STORE WORD ABSOLUTE SHORT (STWAS)	Same as STWS if priv == 0
GATEWAY (GATE)	Always promotes priv to 0
BRANCH VECTORED (BV)	Demotes priv to any nonzero value
BRANCH EXTERNAL (BE)	Demotes priv to any nonzero value, IASQ is nonexistent
BRANCH AND LINK EXTERNAL (BLE)	Demotes priv to any nonzero value, IASQ and SR0 are nonexistent
RETURN FROM INTERRUPTION (RFI)	IASQ is nonexistent
RETURN FROM INTERRUPTION AND RESTORE (RFIR)	IASQ is nonexistent
LOAD SPACE IDENTIFIER (LDSID)	0 is written into specified GR
MOVE TO SPACE REGISTER (MTSP)	Executed as null instruction
MOVE TO CONTROL REGISTER (MTCTL)	Executed as null instruction if target is CR 8, 9, 12, 13, 17 or 20
MOVE FROM SPACE REGISTER (MFSP)	0 is written into specified GR
MOVE FROM CONTROL REGISTER (MFCTL)	0 is written into specified GR if source is CR 8, 9, 12, 13, 17 or 20
PROBE READ ACCESS (PROBER)	Always sets target GR to 1
PROBE READ ACCESS IMMEDIATE (PROBERI)	Always sets target GR to 1
PROBE WRITE ACCESS (PROBEW)	Always sets target GR to 1
PROBE WRITE ACCESS IMMEDIATE (PROBEWI)	Always sets target GR to 1

I Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z