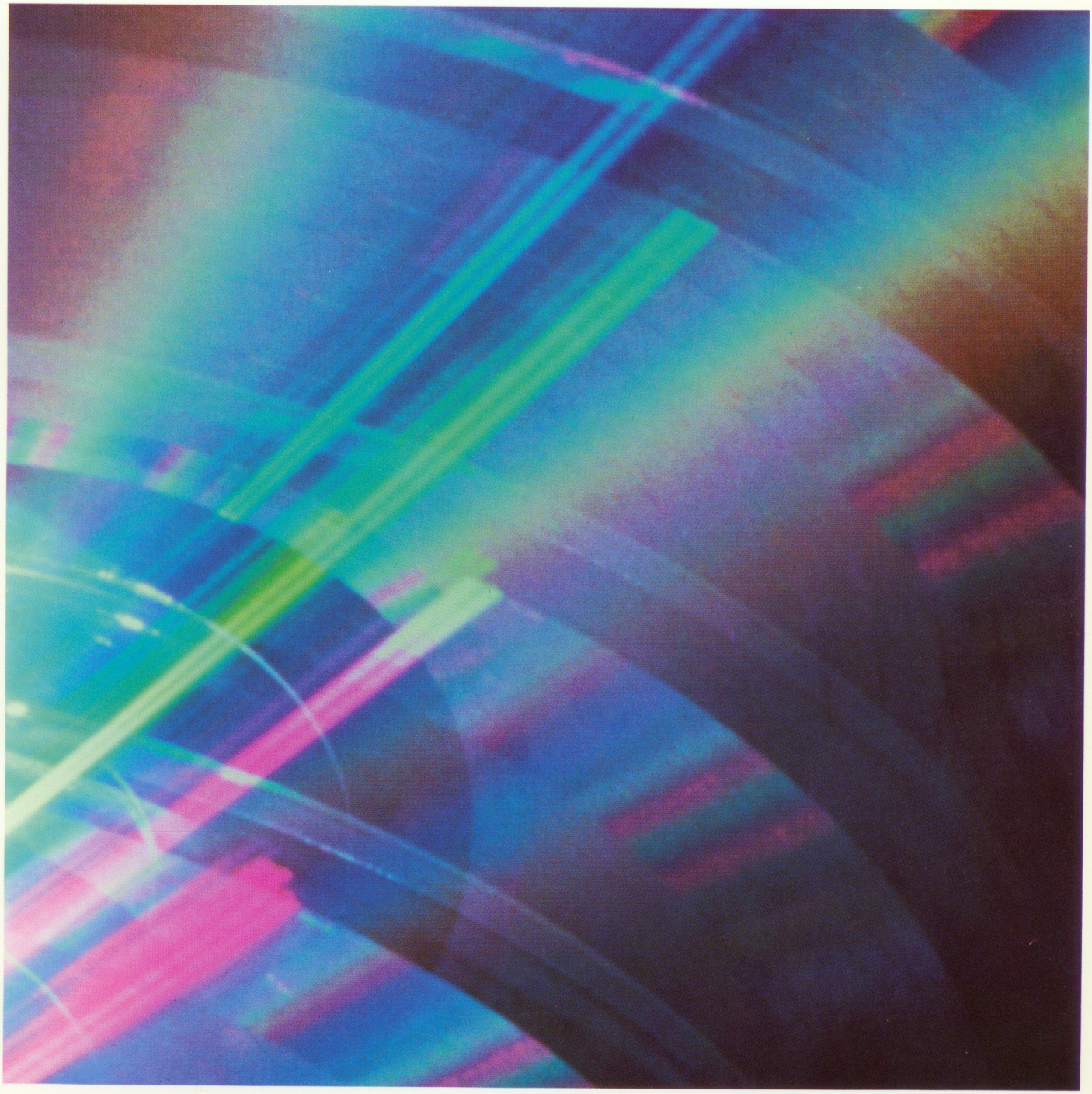
HEWLETT PACKARD

HP 3000/930 and HP 9000/840 Computers

Precision Architecture and Instruction Reference Manual



HP Precision Architecture Computers

Precision Architecture and Instruction

Reference Manual

HEWLETT PACKARD
19483 PRUNERIDGE AVENUE, CUPERTINO, CA 95014

Part No. 09740--90014

Printed in U.S.A. June 1987

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed beside the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

First Edition	Nov 1986
Second Edition	Jun 1987

List of Effective Pages

The List of Effective Pages gives the date of the current edition, and lists the dates of all changed pages. Unchanged pages are listed as "ORIGINAL." Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior date.

Second Edition ..... June 1987

Effective Page	**Date**
ALL	June 1987

Contents

1 Overview 1-1
Introduction 1-1
System Features 1-2
System Organization 1-3

2 System Organization 2-1
Introduction 2-1
Memory and I/O Addressing 2-2
Levels of HP Precision Architecture 2-4
Data Types 2-4
Processing Resources 2-6

3 Addressing and Access Control 3-1
Introduction 3-1
Pointers and Address Specification 3-2
Address Resolution and the TLB 3-4
Access Control 3-6
Software Virtual Address Translation 3-11
Caches, Multiprocessing and I/O 3-14

4 Control Flow and Interruptions 4-1
Introduction 4-1
Instruction Execution 4-1
Instruction Pipelining 4-2
Nullification 4-3
Branching 4-3
Interruptions 4-9

5 Instruction Set 5-1
Introduction 5-1
Undefined Instructions 5-1
Null Instructions 5-2
Conditions and Control Flow 5-2
Instruction Notations 5-8
Instruction Descriptions 5-14
Memory Reference Instructions 5-15
Immediate Instructions 5-52
Branch Instructions 5-56
Computation Instructions 5-80
System Control Instructions 5-137
Assist Instructions 5-172

6 Floating-point Coprocessor 6-1
Introduction 6-1
Coprocessor Registers 6-3
Data Types 6-6
Infinity Arithmetic 6-9
Operations With NaNs 6-9

Sign Bit 6-10
Exceptions 6-10
Saving and Restoring State 6-17
Instruction Format 6-18
Floating-Point Instruction Set 6-19

A Glossary A-1

B Instruction Index B-1

C Instruction Formats C-1

D Operation Codes D-1
Major Opcode Assignments D-1
Opcode Extension Assignments D-2

I Index I-1

Figures

Figure 1-1. System Organization 1-3
Figure 1-2. Processor Organization 1-4
Figure 2-1. Absolute Pointer 2-2
Figure 2-2. Memory and I/O Addresses 2-2
Figure 2-3. Physical Memory Addressing and Storage Units 2-3
Figure 2-4. General Registers 2-6
Figure 2-5. Space Registers 2-7
Figure 2-6. Processor Status Word 2-8
Figure 2-7. Instruction Address Space and Offset Queues 2-10
Figure 2-8. Control Registers 2-11
Figure 2-9. Interruption Instruction Address Space and Offset Queues 2-13
Figure 3-1. Structure of Spaces, Pages and Offsets 3-2
Figure 3-2. Space Identifier Selection 3-3
Figure 3-3. TLB Fields 3-4
Figure 3-4. Protection ID 3-7
Figure 3-5. Access Rights Field 3-7
Figure 3-6. Access Control Checks 3-10
Figure 3-7. Hash Table Entry 3-11
Figure 3-8. Physical Page Directory (PDIR) Entry 3-12
Figure 3-9. Virtual to Physical Address Translation 3-13
Figure 4-1. Delayed Branching 4-4
Figure 4-2. Branch in the Delay of a Branch 4-8
Figure 4-3. Interruption Processing 4-13
Figure 5-1. IA Space and IA Offset Queues 5-8
Figure 5-2. Instruction Description Example 5-14
Figure 5-3. Space Identifier Selection 5-17
Figure 5-4. Loads and Stores 5-17
Figure 5-5. Load and Store Word Modify 5-19
Figure 5-6. Effective Address Computation For Indexed Loads 5-21
Figure 5-7. Short Displacement Loads and Stores 5-23

Figure 5-8. Store Bytes Short 5-25
Figure 5-9. Immediate Instructions 5-52
Figure 5-10. Classification of Branch Instructions 5-58
Figure 5-11. Space Identifier Selection 5-138
Figure 5-12. Effective Address Computation For System Operations 5-138
Figure 6-1. Status Register 6-4
Figure 6-2. Exception Register Format 6-6
Figure 6-3. Floating-point Data Register Format 6-6
Figure 6-4. Binary Floating-point Formats 6-8
Figure 6-5. Binary Fixed-point Formats 6-8
Figure 6-6. Exception Register Parm Field 6-16
Figure 6-7. Floating-point Instruction Format 6-19
Figure D-1. Format for System Control Instructions D-2
Figure D-2. Format for Memory Management Instructions D-3
Figure D-3. Format for Arithmetic/Logical Instructions D-5
Figure D-4. Formats for Indexed and Short Displacement Load/Store Instructions . . . . - D-7
Figure D-5. Format for Arithmetic Immediate Instructions D-8
Figure D-6. Formats for Extract and Deposit Instructions D-9
Figure D-7. Formats for Unconditional Branch Instructions D-10
Figure D-8. Formats for Coprocessor Load/Store Instructions D-11

Tables

Table 3-1. Access Rights Interpretation 3-9
Table 5-1. Arithmetic/Logical Operation Conditions 5-3
Table 5-2. Overflow Results 5-3
Table 5-3. Compare/Subtract Instruction Conditions 5-4
Table 5-4. Add Instruction Conditions 5-5
Table 5-5. Logical Instruction Conditions 5-6
Table 5-6. Unit Instruction Conditions 5-6
Table 5-7. Shift/Extract/Deposit Instruction Conditions 5-7
Table 5-8. Indexed Load and Store Completers 5-20
Table 5-9. Short Displacement Load and Store Completers 5-22
Table 5-10. Store Bytes Short Completers 5-24
Table 5-11. System Control Completers 5-137
Table 6-1. Coprocessor Registers 6-3
Table 6-2. Floating-point Format Parameters 6-7
Table 6-3. Non-trapped Exception Results 6-11
Table 6-4. Trapped Exception Results 6-11
Table 6-5. Floating-point Operand Format Completers 6-19
Table 6-6. Floating-point Operations 6-20
Table 6-7. Floating-point Compare Conditions 6-21
Table D-1. Major Opcode Assignments D-1
Table D-2. System Control Instructions D-2
Table D-3. Memory Management Instructions D-4
Table D-4. Arithmetic/Logical Instructions D-6
Table D-5. Indexed and Short Displacement Load/Store Instructions D-7
Table D-6. Arithmetic Immediate Instructions D-8
Table D-7. Extract, Deposit, and Shift Instructions D-9
Table D-8. Unconditional Branch Instructions D-10
Table D-9. Coprocessor Load and Store Instructions D-11

1

Overview

Introduction

The HP Precision Architecture, an extension of the RISC architectural principles, is the framework for Hewlett-Packard's computer systems. The simple RISC-based 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/engineering, and real-time control.

Computer architectures developed over the last twenty years 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. On architectures with large, complex instruction sets, the simple, often-executed instructions incur a performance penalty caused by the overhead of additional instruction decoding, the use of microcode, and longer cycle time resulting from increased functionality. Together, these penalties negate any advantages of implementing complex instructions. These findings led to the concept of the Reduced Instruction Set Computer (RISC).

All HP Precision Architecture processors use the same processor architecture and the same instruction set. 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 opcodes and register fields always occur 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. Forty-eight-bit or 64-bit virtual addressing 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 gigabytes). A powerful protection mechanism enables secure and structured operating systems.

The HP Precision Architecture is designed to support high-performance or fault-tolerant multiprocessing systems and 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 implemented with the HP Precision Architecture include:

- Direct hardware implementation of instruction set — The instruction set is 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 instruction types — There are less than seventy different instruction operations in the instruction set. Combining basic operations with trapping options and addressing modes, the instruction set expands to only 140 instructions.
- Small number of addressing modes — The instruction set uses short and 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 piped.

HP Precision Architecture includes extensions of RISC concepts and provides a flexible, expandable architecture which maximizes performance from a given semiconductor technology. These extensions also help achieve given levels of performance at significantly lower cost than other systems.

The major extensions are summarized below:

- Very high performance caches systems
- Multiprocessing for fault-tolerance or increased performance
- Assist processors for IEEE floating-point operations
- Extremely large and efficient virtual memory system with 48-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

System Organization

The HP Precision Architecture 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 uniprocessor system with a high-speed central bus and two connections to lower-speed busses. The processor references main memory on the central bus and I/O adapters on the remote busses. The processor is itself a module on the bus and may be the target of transactions such as external interrupts and system resets.

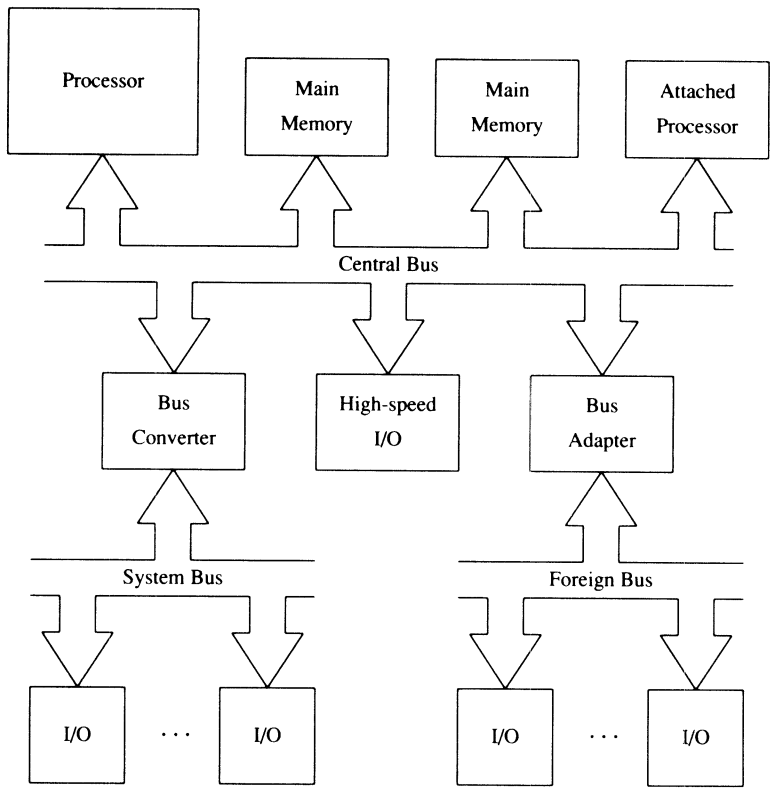


Figure 1-1. System Organization.

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 will incorporate this hardware. On processors that support virtual memory addressing, a hardware structure is included to provide virtual to physical address translations.

Any processor may include Special Function Units (SFUs) and coprocessors to execute algorithms with dedicated hardware to provide substantial performance gain. Collectively, SFUs and coprocessors are called "assist processors." Floating-point functions are provided through 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 cache, TLB, one coprocessor and one SFU.

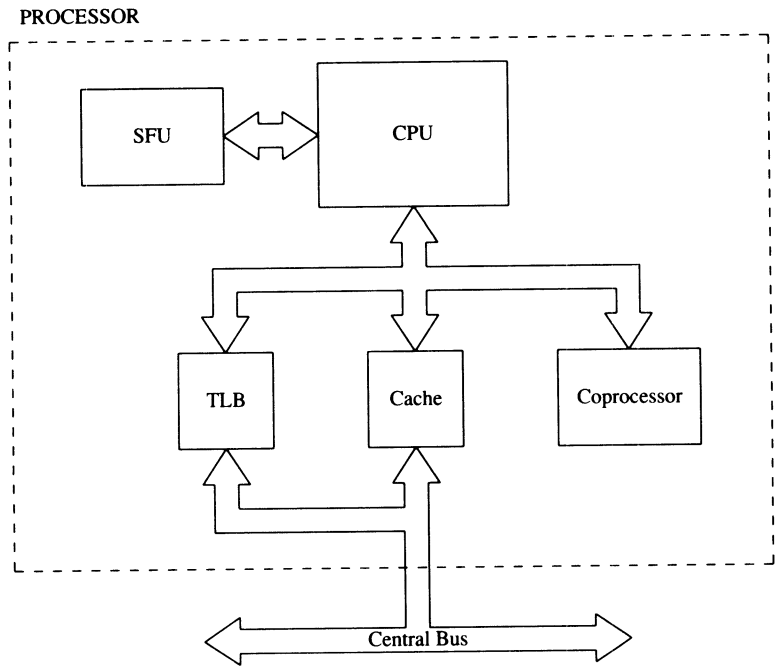


Figure 1-2. Processor Organization.

Register-intensive computation is central to the architecture. Calculations are performed only between high-speed CPU registers or 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 HP Precision Architecture 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 nearly achieves 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 are (highest to lowest) the general registers, caches (if implemented), main memory and, finally, direct access storage devices such as disks.

A cache system is an integral part of the processor in all but the smallest implementations. Caches hold frequently accessed data and instructions to minimize access time to main memory. Caches may be organized as a single, unified cache that holds both instructions and data or as separate instruction (I-cache) and data (D-cache) caches.

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 contains information used to determine valid access to that memory page and the type of access permitted. Once the TLB has 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 code (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 HP Precision Architecture systems. The virtual memory system supports 48-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 gigabytes (2^{32}) long. Each space is specified with a space identifier and divided into fixed-length 2 kilobyte pages.

Data Types

The HP Precision Architecture supports the following data types:

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

Instruction Set

The HP Precision Architecture defines 140 fixed-length instructions. There are two primary addressing modes for data accesses: displacement and indexed. Data references can be specified by either virtual or physical addressing.

Memory Reference Instructions are used to 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.

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 as main memory is allowed.

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 effective 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 immediately determines whether 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 operation of addition, subtraction, multiplication, division, square root, remainder, conversions, and round to integer.

System control instructions provide the functions needed to implement an operating system. These functions include causing the central processor to return from an interruption, and executing an instruction break. They also control the PSW, special registers, caches, translation lookaside buffers, and access rights probes.

Input/Output Organization

The HP Precision Architecture's 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 commands 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. A given I/O device separates its control registers into two addressable pages. One is for protected access by the operating system and trusted software. The other page is for non-privileged users. Since the usual page-level protection mechanism is applied during virtual-to-physical address translation, user programs can be granted direct control over particular I/O devices 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.

A DMA I/O adapter contains sufficient state to control the transfer of data to or from a contiguous range of physical 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 are added to the basic HP Precision Architecture 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.

Multiprocessing

Various types of multiprocessing are supported in HP Precision Architecture systems. Multiprocessors can be configured to provide incremental performance 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 consistency for I/O, 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.

Number Notation

The standard notation for addresses and data is hexadecimal (base 16). The hexadecimal digits are 0 .. F. Memory addresses and fields within instructions are written in hexadecimal. Where numbers could be confused with decimal notation, hexadecimal numbers are preceded with X'. For example, X'2C is equivalent to decimal 44. In the Instruction description section, the C language syntax uses 0x to specify a hexadecimal number. For the previous example, 0x2C is the equivalent number.

2

System Organization

Introduction

The HP Precision Architecture instruction set is only one aspect of the processor architecture; the following components are also specified:

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

The user-accessible registers (i.e., the processing resources) are the storage elements within a processor that are manipulated by the instructions. The registers are at the highest level of storage hierarchy. They participate in instruction control flow, computations, interruption processing, protection mechanisms, and virtual memory management. The processor registers and registers sets available to the user are described in this chapter and listed below:

- General Registers (GR 0 — GR 31)
- Space Registers (SR 0 — SR 31)
- The Control Registers (CR 0 — CR 31)
- Processor Status Word (PSW)

The assist register sets available to the user are described in Chapter 6 and listed below:

- The Coprocessor Registers (CPR 0 — CPR 31)
- The Floating-point Registers (FPR 0 — FPR 15)

The memory system is the next level of data storage beyond the processing resources. It consists of main memory and high-speed buffers that hold recently referenced instructions and/or data. These buffers, called *instruction* and/or *data caches*, reduce the access time to main memory.

The I/O system is memory-mapped. I/O attachments 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 those physical pages and I/O modules represented by words in a page, communication between a processor and an I/O attachment can be performed with load and store instructions. The privilege level and access rights of such a page provide versatile protection.

Memory and I/O Addressing

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

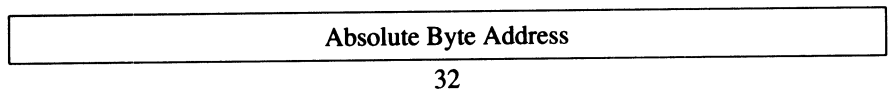


Figure 2-1. Absolute Pointer.

Figure 2-2 illustrates the relationship of the I/O register address space to the main memory address space. Addresses 0 through X'EFFFFFFF reference memory. Addresses X'F0000000 through X'FFFFFFFC reference I/O registers. This structure gives nearly 4 gigabytes of main memory address space and 256 megabytes of address space for I/O devices.

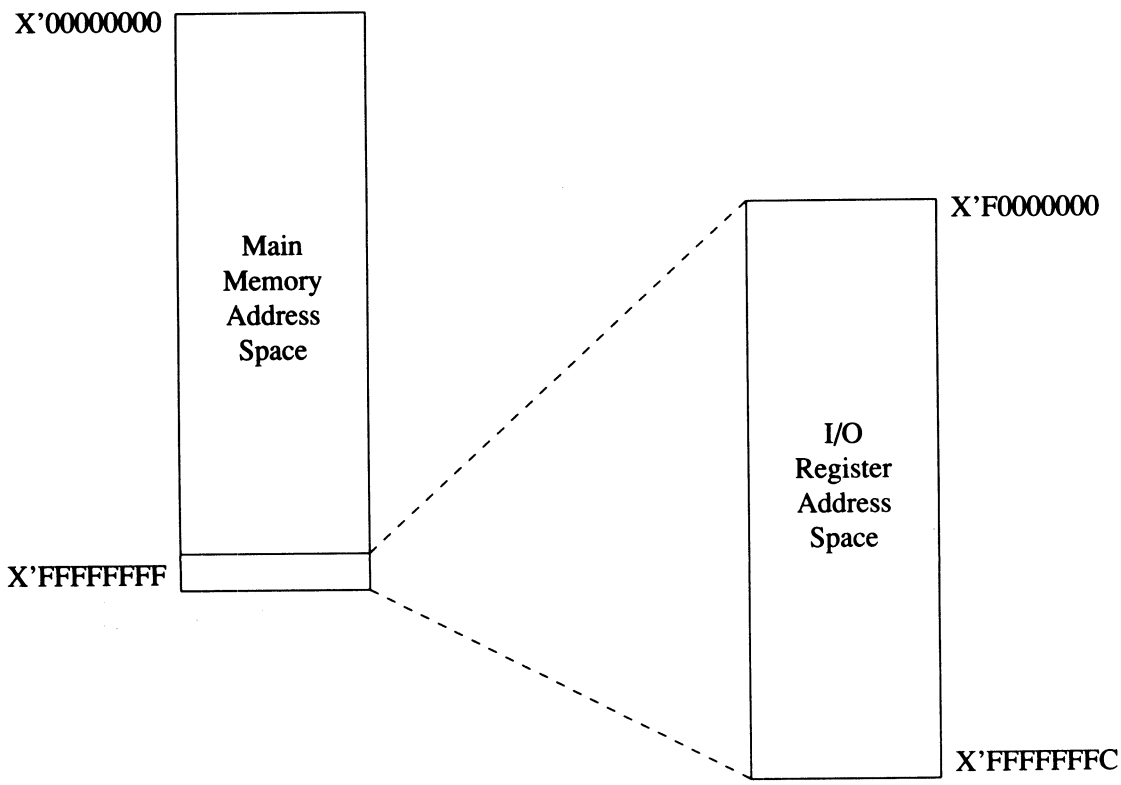


Figure 2-2. Memory and I/O Addresses.

Memory is always referenced with byte addresses, starting with address 0 and extending through the largest defined address (X'EFFFFFFF). 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 and the relative bit positions above each block. Bytes in memory and bits within larger units are numbered from 0 starting with the leftmost byte or bit, respectively. Note that bit or byte 0 is the leftmost.

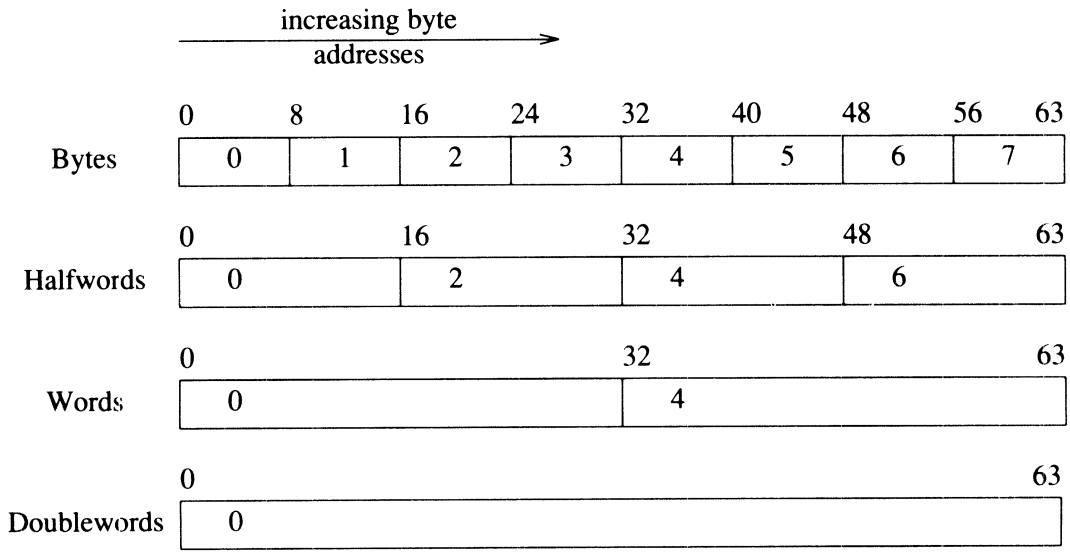


Figure 2-3. Physical Memory Addressing and Storage Units.

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

I/O address space is referenced in words. I/O registers are accessed using the normal load and store word 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 the Translation Lookaside Buffers (TLBs), which are described in Chapter 3. 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. Each eligible process must have a requisite *privilege level* (PL) to complete the access.

Levels of HP Precision Architecture

Three levels of the processor architecture have been defined: *zero*, *one* and *two*. Level zero systems support absolute memory addressing only. Virtual memory is not supported and so space identifiers are not used. Level one and two systems have virtual addressing and differ only in the number of significant bits in space identifiers. Level one systems have 2^{16} virtual spaces. Level two systems have 2^{32} virtual spaces. To provide for growth to larger systems, each higher level processor is a superset of the capabilities of the lower level processors.

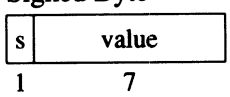
Data Types

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

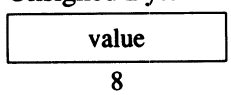
Bits — Memory is not addressed to the resolution of bits; however, efficient support is provided to manipulate and test individual bits in general registers.

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

Signed Byte



Unsigned Byte



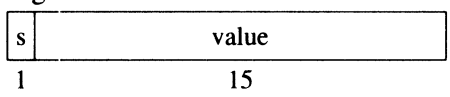
Bytes are packed four to a word and may represent a signed value in the range -128 through +127, an unsigned value in the range 0 through 255, an arbitrary collection of eight bits, or an ASCII character.

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.

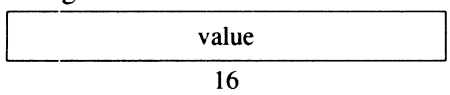
Integers

Integers may be 16 or 32 bits long, signed or unsigned:

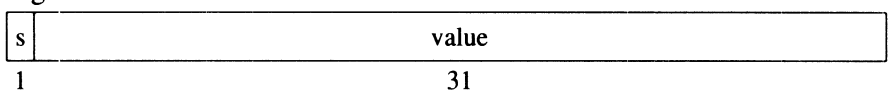
Signed Halfword



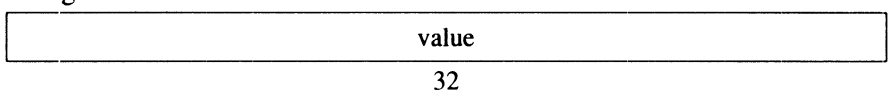
Unsigned Halfword



Signed Word



Unsigned Word



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

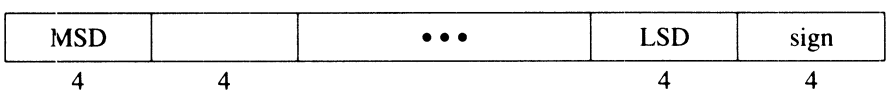
Floating-Point Numbers

The binary floating-point number representation conforms to the ANSI/IEEE 754-1985 standards. The single (32-bit), double (64-bit), and quad (128-bit) binary formats are supported by software and, in some models, by special hardware.

Single-precision floating-point numbers must be aligned on word boundaries when stored in memory. Double-precision and quad-precision numbers must be aligned on doubleword boundaries. See Chapter 6 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 long and having a value in the range of X'0 to X'9, followed by a 4-bit sign as shown in the following figure:



The standard sign for a positive number is X'C, but any value except X'D will be interpreted as positive. X'B 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 four 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. The effect of writing ones is an undefined operation.
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, however, it does not have any effect on system functionality.
3. Reserved registers — A register that is numbered but currently unused. Both READ and WRITE operations are undefined.
4. Nonexistent registers — A register that does not architecturally exist in level zero systems. A READ operation returns zeros. A WRITE operation has no effect (executes a null instruction).

General Registers

Thirty-two 32-bit general registers provide the central resource for all computation (Figure 2-4). They are numbered GR 0 through GR 31, and are available to all programs at all privilege levels.

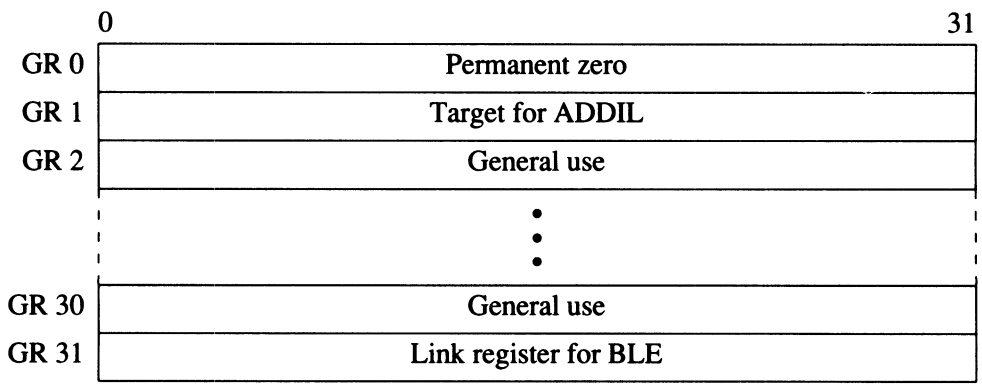


Figure 2-4. General Registers.

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

Space Addressing Registers

For virtual addressing, space identifiers are specified in eight space addressing registers, labeled SR0 through SR7. 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. SR0 is an implied target for the return address of the interspace linkage instruction. Data operands can explicitly specify SR 1 through SR 3. SR 4 through SR 7 can be specified by general registers.

SR 1 through SR 7 have no special architecturally defined 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 are used as scratch registers for the manipulation of 64-bit virtual pointers. SR 4 tracks 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 2-5 illustrates this convention.

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

On level zero systems the space addressing registers are nonexistent. On level one systems they are 16 bits long; the upper 16 bits are nonexistent. On level two systems they are 32 bits long.

SR 0	Link code space ID
SR 1	64-bit pointer
SR 2	64-bit pointer
SR 3	64-bit pointer
SR 4	Tracks IA space
SR 5	Process private data
SR 6	Shared data
SR 7	OS public code, literals and data

Figure 2-5. Space Registers.

Processor Status Word (PSW)

Processor state is encoded in a 32-bit register called the Processor Status Word (PSW). The "on" state for each PSW bit is a 1. When an interruption occurs, the old value of the PSW is saved in the Interruption Processor Status Word (IPSW) and generally all defined PSW bits are set to 0. The format of the PSW is shown in Figure 2-6. The unnamed bits (bits 0-6 and 24-26) are reserved bits. It is software's responsibility to ensure these bits are clear when written.

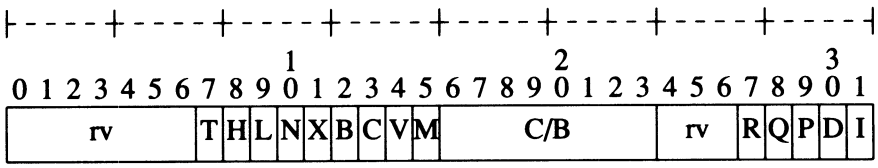


Figure 2-6. Processor Status Word.

The PSW is set to the contents of the IPSW by the interruption return instruction. The interruption handler may restore the original PSW, modify selected bits, or may change the PSW to entirely new values.

The R, Q, P, D, and I bits of the PSW are known as the system mask. They may be set, reset, written, and read by the system control instructions that manipulate the system mask. The Q-bit is specially defined. It can be cleared by system control instructions that manipulate the system mask, but it cannot be set when currently clear. Attempts to change the state of the Q-bit to a 1 by these instructions are undefined operations. The only instruction that can set the Q-bit to a 1 is the interrupt return instruction.

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 field.
T	Taken branch trap enable. When set, any successful branch is terminated with a taken branch trap.
H	Higher-privilege transfer trap enable. When set, a higher privilege transfer trap occurs whenever the following instruction is of a higher privilege.
L	Lower-privilege transfer trap enable. When set, 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 set. This bit is set by an instruction that nullifies the following instruction.
X	Data Memory break disable bit. The PSW X-bit is cleared after the execution of each instruction. The interruption return instruction may set it. When set, 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 by any successful branch instruction and cleared otherwise. This is used to ensure that the privilege increasing instruction is not compromised.
C	Code address translation enable. When set, 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 set, High Priority Machine Checks (HPMCs) are not allowed. Normally 0, this bit is set after a HPMC and cleared 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 if a carry occurred out of its corresponding digit, and cleared otherwise. After a subtract which sets them, each bit is cleared if a borrow occurred into its corresponding digit, and set otherwise.

R Recovery counter enable. When set, recovery counter traps are allowed 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 set, interruption status is collected. Used in processing the interrupt 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 ID validation enable. When set, data and instruction references check for valid Protection Identifiers.

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

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

Instruction Address Queues

The Instruction Address Queues hold the instruction address of the currently executing instruction and the address of the instruction that is 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 entries deep. The Instruction Address Offset Queue (IAOQ) entries are each 32 bits long. The high 30 bits contain the word offset of the instruction and low two bits maintain the privilege level of the corresponding instruction.

The Instruction Address Space Queue (IASQ) is 32 bits wide in a level two system, 16 bits wide in a level system and nonexistent in a level zero system. The upper 16 bits of the IASQ entries in a level one system are also undefined. The space ID of the current instruction, when executing without instruction translation enabled is not specified and may contain any value.

The front entries of the two queues form the virtual address of the current instruction and the back entries contain the address of the following instruction. Figure 2-7 shows this structure. Two addresses must be maintained to support the delayed branching capability. Generally the following instruction is sequentially after the current instruction but after taken branches, they are usually different addresses.

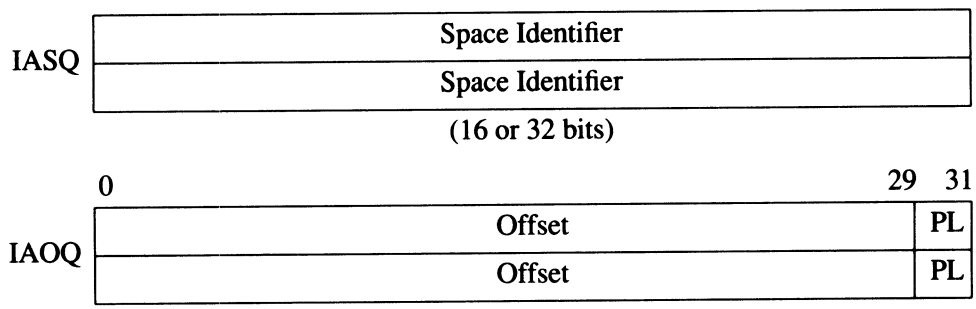


Figure 2-7. Instruction Address Space and Offset Queues.

Control Registers

There are twenty-five control registers, labeled CR 0, CR 8, and CR 9 through CR 31, which contain system state information. CR 11, the Shift Amount Register, is readable and writable by code running at any privilege level. CR 16, the interval timer, is readable by code running at any privilege level. All other control registers are accessible only by code executing at the most privileged level. Their functions are indicated in Figure 2-8 and described in the following sections. Retrieving control registers into general registers copies the register right aligned into the general register.

Control registers 1 through 7 are reserved registers, and the unused bit positions of the protection IDs are reserved bits. The unused bits of the shift amount register are nonexistent bits.

Recovery Counter

The Recovery Counter (CR 0) can be used to provide software recovery of hardware faults in fault tolerant systems. CR 0 counts down by 1 during the execution of each non-nullified instruction for which the R-bit in the PSW is a 1. The recovery counter is restored if the instruction terminates with a group 1, 2, or 3 interruption (see Chapter 4). When the leftmost bit of the Recovery Counter is a 1, a recovery counter trap is raised. The trap, and the decrement operation can be disabled by clearing the R-bit in the PSW. The trap conditions are re-evaluated at the beginning of each instruction. The value of the Recovery Counter may be read reliably only when the R-bit in the PSW is 0. The Recovery Counter may be written to reliably only when the PSW R-bit is 0. Otherwise it is an undefined operation. If the PSW R-bit is reset (to zero) by RSM or MTSM instructions, the recovery counter may not be read or written reliably prior to the execution of the eighth instruction after the RSM or MTSM. An interruption or an RFI which resets the R-bit does not have this restriction.

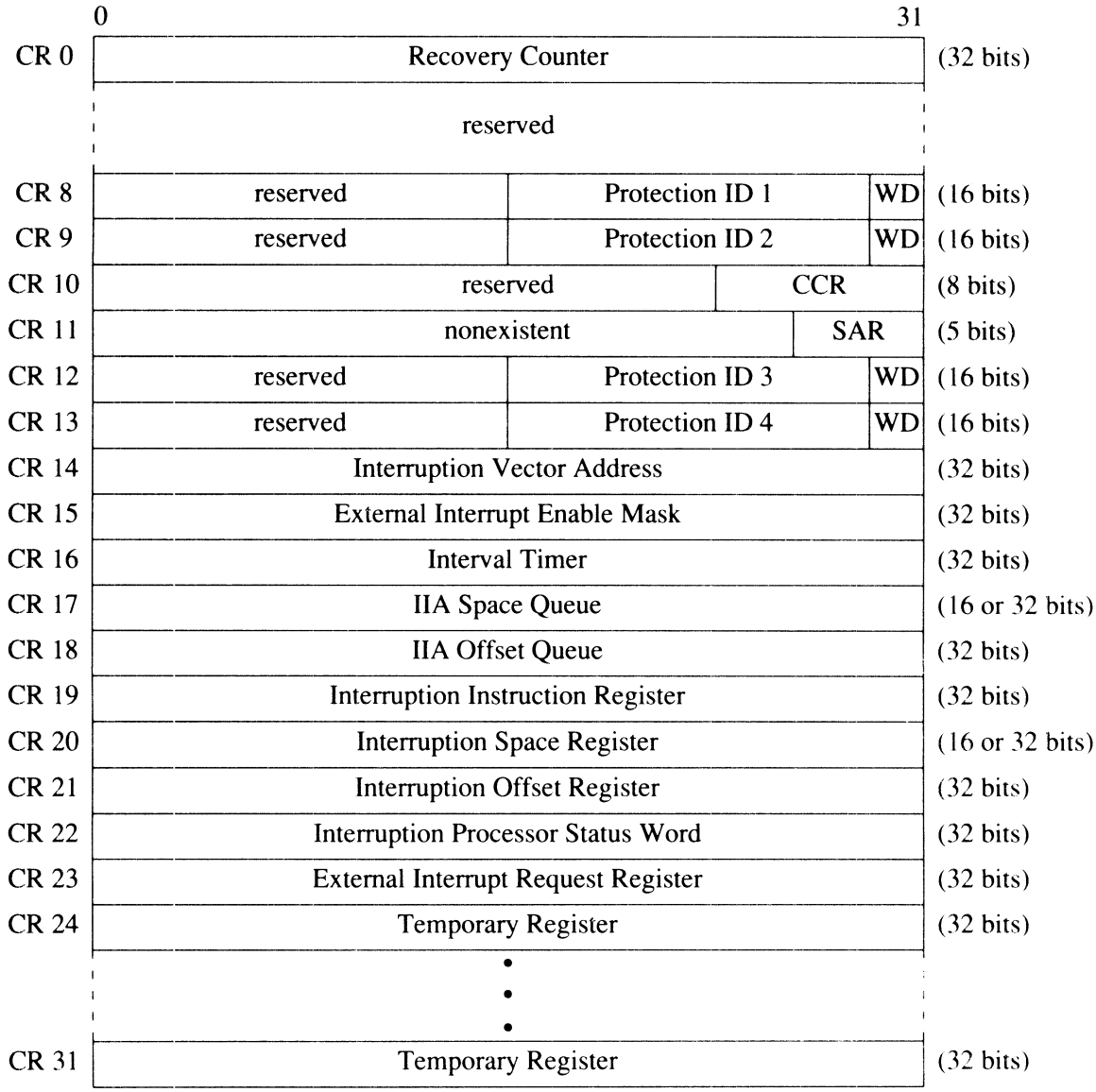


Figure 2-8. Control Registers.

Protection Identifiers (CRs 8, 9, 12, and 13)

The Protection Identifiers designate up to four groups of pages which are accessible to the currently executing process. When translation is enabled, the four protection IDs are compared with a page access identifier to validate the access. The rightmost bit of each of the four Protection IDs is the write disable (WD) bit. When the WD-bit is set to 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 protection ID, including the WD-bit, is ignored.

In level two and level one systems, the protection IDs are defined to be 16-bit registers, with the upper 16 bits as reserved bits. In level zero systems, the protection IDs are nonexistent registers.

Coprocessor Configuration Register (CR 10)

CCR, the Coprocessor Configuration Register, is an 8-bit register which records the presence and usability of coprocessors. The bit positions are numbered 0 through 7, and each bit corresponds to a coprocessor with the same unit number. Bit 7 is the rightmost bit of the CCR. It receives bit 31 from a GR when a GR is written to the CCR. Setting a bit in CR 10 indicates that the corresponding coprocessor is present and operational. If a bit is 0, an attempt to execute an instruction which references the corresponding coprocessor causes an assist emulation trap. The remaining 24 bits are reserved bits. Also, a reference to an unimplemented coprocessor with the appropriate CCR bit set to 1 is undefined.

Shift Amount Register (CR 11)

SAR, the Shift Amount Register, 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 (CR 14)

The Interruption Vector Address contains the absolute address of an array of service procedures assigned to interruption classes. This address must be a multiple of 1024. Use of an unaligned address is an undefined operation. It is indexed by the interruption numbers given in Chapter 4.

External Interrupt Enable Mask (CR 15)

EIEM, the External Interrupt Enable Mask Register, is a 32-bit register containing a bit for each external interrupt. When set to a 0, this bit masks interruptions pending for the external interrupt designated by that bit position.

Interval Timer (CR 16)

The Interval Timer consists of two internal registers which are both accessed through CR 16. One of the internal registers is continually counting up by 1 at a rate which is model-dependent and between twice the "peak instruction rate" and half the "peak instruction rate". Reading CR 16 returns the value of this internal register. The other internal register contains a comparison value and is set by writing to CR 16. 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. CR 16 can only be written by code running at the most privileged level, but can be read by code running at any privilege level.

Interrupt Instruction Address Space and Offset Queues (IIASQ and IIAOQ)

Two 32-bit registers and two 16 or 32-bit registers are used to save the Instruction Address and privilege level information for use in processing interruptions. The registers are arranged as two two-entry 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 queue is continually updated whenever the PSW Q-bit is one and frozen by an interruption (Q-bit becomes 0). After such an interruption, the IIAQs contain copies of the IA queues. Reading the IIA Offset Queue register or IIAOQ (CR 18) while the PSW Q-bit is 0 retrieves the offset and privilege level portions of the front entry 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 entry of the IIAOQ. Reading the IIA Space Queue register or IIASQ (CR 17) while the PSW Q-bit is 0 retrieves the space portion of the front entry of the IIASQ. Writing into IIASQ while the PSW Q-bit is 0 advances the IIASQ and then writes into the back entry 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 Queues are shown in Figure 2-9.

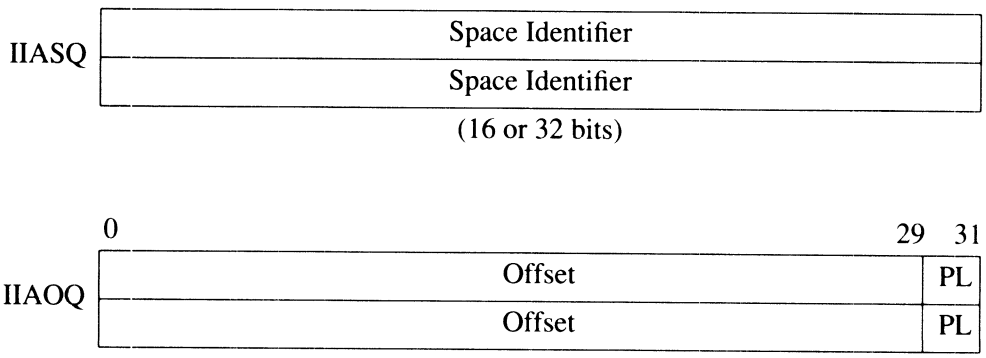


Figure 2-9. Interruption Instruction Address Space and Offset Queues.

The IIASQ is nonexistent on level zero systems, 16 bits wide on level one systems, and 32 bits wide on level two systems. On level one systems, the upper 16 bits are nonexistent. The state contained in the IIA queues are undefined when an interruption return instruction leaves the Q-bit zero, or when system control instructions are used to turn off the Q-bit. If an interruption is taken with the Q-bit 0, the IIA queues are unchanged.

Interruption Parameter Registers (IIR, ISR and IOR)

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, 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. These values are set (or frozen) at the time of the interruption whenever the Q-bit is 1. The ISR may be nonexistent, 16, or 32 bits long. This depends on the existence or length of space IDs in the particular model. On level one systems, the upper 16 bits of the ISR are nonexistent. The value loaded into IOR is the full 32-bit offset of the virtual address without truncation or clearing of rightmost bits. Writing into the Interrupt 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 are undefined when an interruption return instruction leaves the Q-bit zero, or when system control instructions are used to turn off the Q-bit. If an interruption is taken with the Q-bit 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 must always reflect the state of the machine at the point of interruption, regardless of whether the Q-bit was 1 or not. As in the PSW, the unnamed bits are reserved 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 an interruption return instruction leaves the Q-bit zero, or when system control instructions are used to turn off the Q-bit.

External Interrupt Request Register (EIR)

The External Interrupt Request Register or EIR (CR 23) is a 32-bit register containing a bit for each external interrupt. When set to a 1, this bit designates that an interruption is pending for the external interrupt designated by that bit position. 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 logically ANDs the complement of the value moved to it with the previous contents of CR 23. Thus the processor can only clear EIR bits.

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

Temporary Registers

The eight 32-bit Temporary Registers (CRs 24 through 31) are usable only by code executing at the most privileged level. They provide space to save the contents of GR for interruption handlers in the operating system kernel.

PROGRAMMING NOTE

CRs 24, 25, and 26 are reserved for hardware TLB handling, which may be provided in some implementations. In implementations which do software TLB handling they should be used only by the TLB handler to avoid conflict with this anticipated future usage.

3

Addressing and Access Control

Introduction

HP Precision Architecture processors use byte addressing to fetch instructions and data from main memory or the I/O registers. The byte addresses may be either physical addresses or virtual addresses. Physical addresses are used directly and no protection or access rights check is performed. Virtual addresses are translated to physical addresses and undergo protection and rights checking.

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

Three *levels* of the processor architecture have been defined: *zero*, *one* and *two*. Level zero zero systems are fundamentally different from level one and two systems. Level zero systems support only absolute addressing and have no space registers. Level one and two systems provide virtual addressing through the use of 16-bit or 32-bit space registers.

Virtual memory is structured as a set of address spaces, each containing 2^{32} bytes (4 gigabytes). Level one processors have 2^{16} address spaces and level two processors have 2^{32} address spaces. Level one and two systems provide virtual addressing and differ only in the number of significant bits in space identifiers. A level one system implements 16-bit space identifiers to provide 48 bit addresses. A level two system implements 32-bit space identifiers and allows 64-bit virtual addresses. All processor resources that hold space IDs are 32 bits wide. Resources that hold space IDs are implemented as 16-bit registers.

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. The concatenation of a space identifier and an offset within the space form the virtual address.

For memory management purposes, the address space is logically subdivided into pages of 2K (2048) bytes in length. On a level one system, the virtual page number is represented by 37 bits in the virtual address and on a level two system by 53 bits. In both cases, the byte index into the page is specified by the rightmost 11 bits of the virtual address. Figure 3-1 illustrates the structure of spaces, pages and offsets.

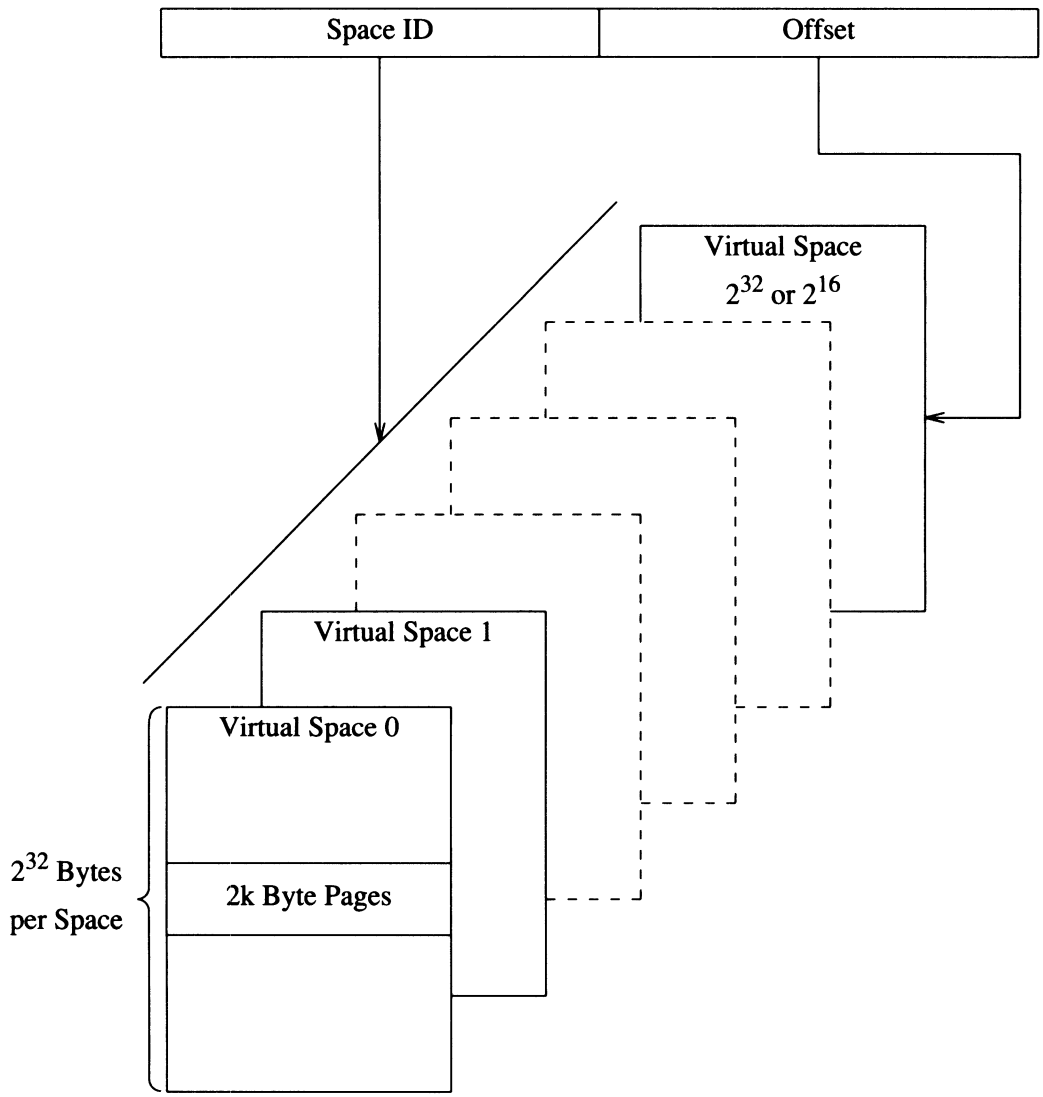


Figure 3-1. Structure of Spaces, Pages and Offsets.

Pointers and Address Specification

Eight space registers, the instruction address (IA) 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.

Space registers 0 through 7 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. Instruction addresses are aligned on word (4 byte) boundaries. When an instruction address is computed, the 3-bit s-field in the instruction selects the space register to be used and the rightmost two bits of the offset are used to hold the privilege level.

The current instruction address (IA) consists of a space identifier, a 32-bit byte offset (of a word aligned address), and, in its two rightmost bit positions, the current privilege level. This privilege level controls both instruction and data references. The current instruction address is maintained in the front element of the IA space and offset 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 sign-extended displacement.

The space identifier, for data references, is selected from space registers 1 through 7 by the following procedure. When nonzero, the 2-bit s-field selects corresponding space registers 1, 2, or 3. When the s-field is zero, the two leftmost bits (bits 0 and 1) of the base register are used to select one of the space registers (4 through 7). Adding four to these two bits generates the selected space register. 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 the region corresponds to the quadrant selected by the upper two bits. For example, if a base register contains the value X'40001000, space register 5 is used as the space identifier and the second quadrant of the space is directly addressable.

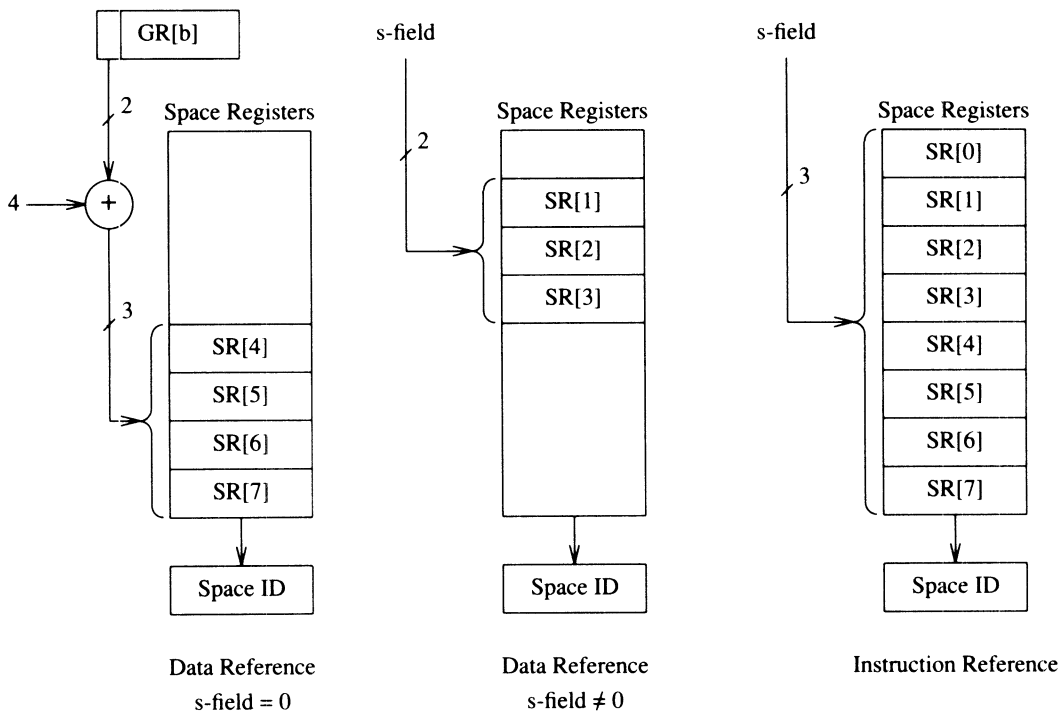


Figure 3-2. Space Identifier Selection.

Address Resolution and the TLB

Virtual addresses are translated to physical addresses using a hardware structure called the translation lookaside buffer (TLB). A TLB is a hardware table that accepts a virtual page number and returns the corresponding physical address. This table is not large enough to hold all the current translations. All the translations are stored in a memory structure called the Physical Page Directory (PDIR). The TLB is organized as two parts. The instruction TLB (ITLB) is only used for all instruction references, while the data TLB (DTLB) is only used for all data references. A system may implement a *combined* TLB which is used for both instruction and data references.

Given a virtual address, the selected TLB is searched for an entry matching the virtual page number. If the entry exists, the 21-bit absolute page number (contained in the TLB entry) is concatenated with the original 11-bit page index to form a 32-bit absolute address. If no such entry exists, the TLB is updated by one of two mechanisms depending on whether the system implements software TLB handling or hardware TLB handling.

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

In implementations that provide hardware for TLB miss handling, the hardware performs a translation using a translation table in memory, places the translation and protection fields into the appropriate instruction TLB or data TLB. No interruption occurs unless the translation does not exist in the TLB. This condition is normally a page fault.

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.

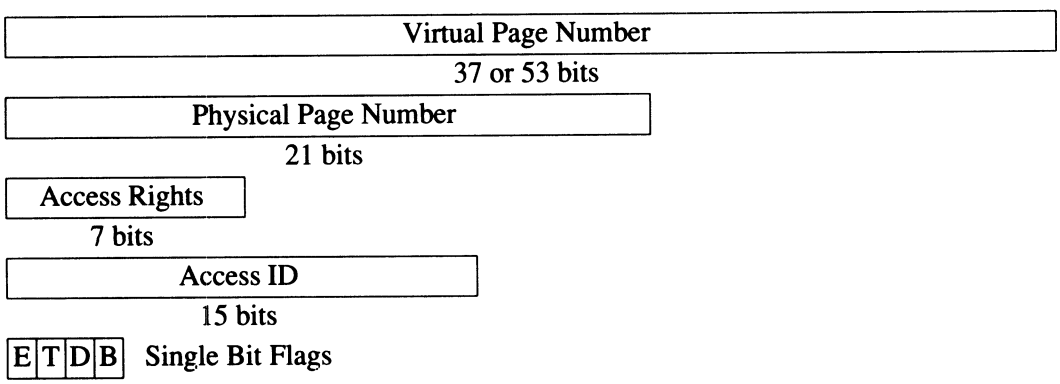


Figure 3-3. TLB Fields.

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

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

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

D Dirty. When 0, stores and semaphore instructions cause a TLB dirty bit trap on systems with software TLB handling. When 0, stores and semaphore instructions cause the D-bit in the DTLB entry and the PDIR to be set on systems with hardware TLB handling. When 1, no trap or update occurs. The D-bit is 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 interruption. Stores, cache purge, 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 T, D, and B bits are only implemented in the DTLB or a combined TLB. The translation buffer is managed by a mixture of hardware and software mechanisms. Translations are brought into the TLB by either hardware or software when a translation misses. On systems which provide hardware for TLB miss handling, the PDIR holds all the necessary information needed by the TLB. For systems with software TLB 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 ITLB ADDRESS and INSERT ITLB PROTECTION instructions place the complete translation and access control information into the ITLB. An equivalent pair (IDTLBA, IDTLBP) places the complete translation and access control information and also initializes the O/S and debugging support bit fields in the DTLB.

Several mechanisms remove translations from the ITLB or DTLB. First, a translation may be displaced when another translation is placed in the TLB. Second, a specific virtual address may be used to remove (purge) the associated translation from the TLB. The PURGE INSTRUCTION TLB and PURGE DATA TLB instructions perform this function. There is no instruction to remove a translation from both the instruction and data TLBs simultaneously. Also, a translation may be removed by another processor in a multiprocessor system.

Translations may also be removed from the TLB using the PURGE INSTRUCTION TLB ENTRY and the PURGE DATA TLB ENTRY instructions. These remove some machine specific entry in the TLB without regard for the translation. These instructions are used by system software in loops to clear the entire instruction or data TLB.

A virtual address cannot translate to two different physical addresses nor can two virtual addresses map to the same physical address. It is the responsibility of memory management software to prevent these occurrences.

Caches are required to permit a physical memory location to be accessed by both an absolute and virtual address for virtual addresses that meet the following conditions:

1. The virtual address has a space identifier equal to 0, and
2. The virtual address has a virtual offset equal to its absolute address.

A virtual address meeting these conditions is said to be equivalently-mapped. A absolute address is equivalent to a virtual address formed by concatenating space identifier 0 with the absolute address.

Caches are not required to detect that the same physical memory location is accessed by different virtual addresses or by both a absolute and a virtual address, except for equivalently-mapped addresses. Since this condition, loosely called aliasing, can be caused only by software running at the most privileged level, it is the responsibility of such software to avoid the ambiguities it may create. This requires flushing the affected address range from the caches prior to any of the following:

1. Changing the address mapping in the TLB's.
2. Making a absolute access to a location which might reside in the caches as a result of access by a virtual address that was not equivalently-mapped.
3. Making a virtual access to a location which might reside in the caches as a result of access by its absolute address that was not equivalently-mapped.

The following rules are guaranteed to prevent address aliasing:

1. Before a PDIR entry is modified to contain a new virtual page, the current virtual page, which maps to the physical page is flushed from the data cache. If the PDIR access rights include execute permission, the virtual page is also flushed from the instruction cache. Instruction prefetch restricts the immediate reuse of virtual addresses. For example, execute access instructions or interrupt return instructions must occur after a translation is changed.
2. Any page that is accessed in absolute mode is equivalently mapped in the PDIR. (Access to the virtual address may be restricted through protection ID or access rights.)

Access Control

A set of mechanisms are available to system software to provide a secure and protected environment for user processes. These mechanisms are collectively known as access control and 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 translation is enabled, and is done on a page basis.

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 memory reference is completed. If the access is not validated, the instruction is terminated with a protection trap. Violations of an instruction access are reported with an instruction memory protection trap. Data read and write access violations are reported with a data memory protection/unaligned data reference trap. 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 identifier control 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. The privilege level is kept in bits 30 and 31 of the current instruction's address. 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 check.
- Access type - The access type is either read, write, or execute. Load, semaphore, and read probe instructions are a *read access* of their operands. Store, semaphore, cache purge, and write probe

instructions are a *write access* of their operands. The semaphore instructions are both a read and write access type. The only *execute access* occurs when the current instruction is fetched for execution.

- Protection IDs - The four control registers CR8, CR9, CR12 and CR13 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 rightmost bit is the write-disable (WD) bit. When 0, write accesses that match that protection ID are allowed. The remaining 15 bits hold the protection ID.

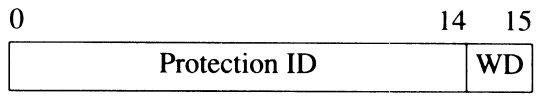


Figure 3-4. Protection ID.

- PSW access attributes - The PSW protection validation (P-bit), code translation (C-bit), and data translation (D-bit) bits further qualify the process attributes. When the P-bit is a 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 translation and access check is enabled. When 1, translation is performed and execute access checks are made. When 0, no translation is performed and the access is always allowed. Read and write accesses use the D-bit in an equivalent manner.

For each entry in the TLB, the *access ID* and the *access rights* field determines if an access is allowed. The access ID is a 15-bit field in the TLB that is used with the protection IDs in the protection ID check.

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 lower bound. The three sub-fields *type*, privilege level 1 (*PL1*), and the privilege level 2 (*PL2*) 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.

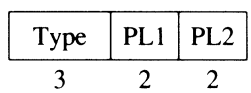


Figure 3-5. Access Rights Field.

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 an lower and upper bound 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. A page containing this branch instruction may promote the current process to a higher 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 following instruction for some implementations. Software cannot depend on the privilege level of the following 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. The access ID is a 15-bit field in the TLB. This check is validated if any of the protection ID's compare equal with the access ID. In case of a write access, the write disable bit of any 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 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 (protection ID validation) determines whether protection ID check is performed. When 0, no protection check occurs and only access rights check is performed. Figure 3-6 illustrates the rights and protection ID check 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 ≤ PL1 write: Not allowed execute: Not allowed
001	Read/Write: dynamic data page	read: PL ≤ PL1 write: PL ≤ PL2 execute: Not allowed
010	Read/Execute: Normal code page	read: PL ≤ PL1 write: Not allowed execute: PL2 ≤ PL ≤ PL1
011	Read/Write/Execute: Dynamic code page	read: PL ≤ PL1 write: PL ≤ PL2 execute: PL2 ≤ PL ≤ PL1
100	Execute: promote to privilege level 0*	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1
101	Execute: promote to privilege level 1*	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1
110	Execute: promote to privilege level 2*	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1
111	Execute: promote to privilege level 3*	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1

* Promotion 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.

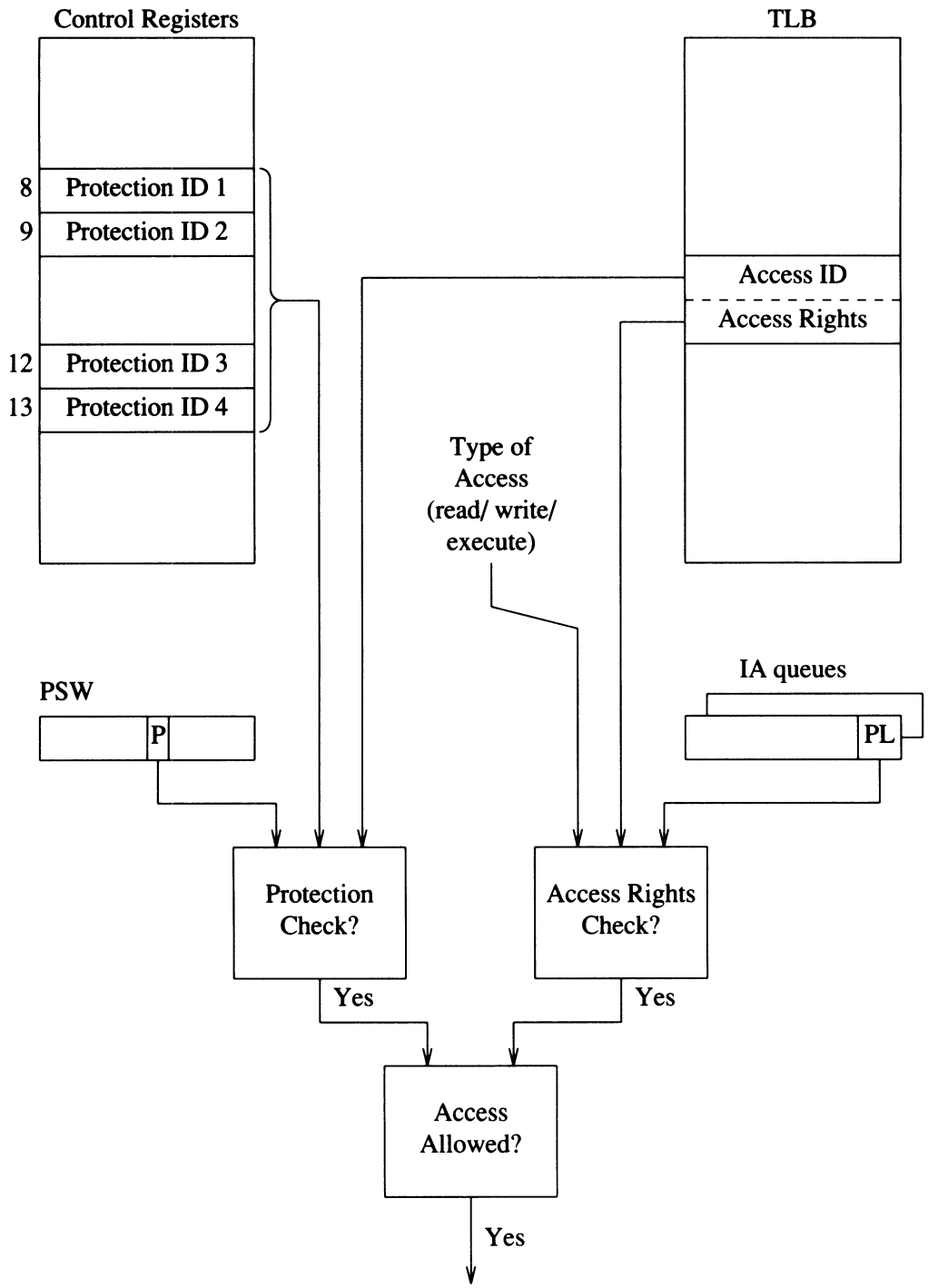


Figure 3-6. Access Control Checks.

Software Virtual Address Translation

Systems that implement software TLB handling perform virtual address translation in software. Once the virtual address of a page has been translated to a physical address, the translation is entered into the TLB along with other information associated with the page. One such method involves the use of two tables, the hash table (HT) and the physical page directory (PDIR). Following is an outline of this method. Due to efficiency requirements, several of the temporary control registers are used to service TLB misses in software.

The translations are stored in the PDIR as a physical to virtual table. The most common use of the PDIR is to translate a virtual address to a physical address after a TLB miss. To provide the inverse access (virtual to physical) the table must be efficiently searched. The hash table is indexed by some function of the virtual address bits to select a pointer into the PDIR. Collisions created by multiple addresses hashing to the same entry are resolved using a sequentially searched linked list.

The PDIR and hash tables are located by descriptors that define the absolute starting address. For efficiency, these descriptors are kept in control registers (assumed to be CR 24 for the PDIR address and CR 25 for the HT address). Note that both tables are required to be memory-resident and each must be allocated in contiguous physical memory.

The number of entries in the hash table is typically a multiple of the number of entries in the PDIR, rounded up to the nearest power of two. The format of a hash table entry is given in Figure 3-7.

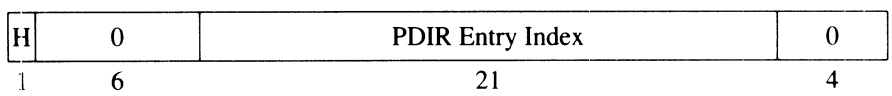


Figure 3-7. Hash Table Entry.

The fields are:

H	indicates whether this entry is in use. If H = 0, then one or more virtual pages hash to this entry. If H = 1, then the hash chain is empty
PDIR Entry Index	indicates the index of the first entry in the PDIR that corresponds to this hash value (if H = 0).

Note that the PDIR Entry Index is negative if it represents a page dedicated to I/O. Thus it needs to be sign extended when used.

The physical page directory (PDIR) contains one entry for each page of physical memory, plus one for each page corresponding to a physical or virtual I/O device. The entries for physical pages are at nonnegative offsets from the location pointed to by CR 24, and the I/O entries are at negative offsets. The entry for a physical page can be located by indexing the table by the physical page number. The PDIR Entry Index in the hash table is used to select the PDIR entry for the first virtual page which hashes to that hash class; further virtual pages that hash to the same value may be found by sequentially searching the collision chain which begins with the *Next PDIR Entry Index* field found in this entry.

The format of a PDIR entry is given in Figure 3-8.

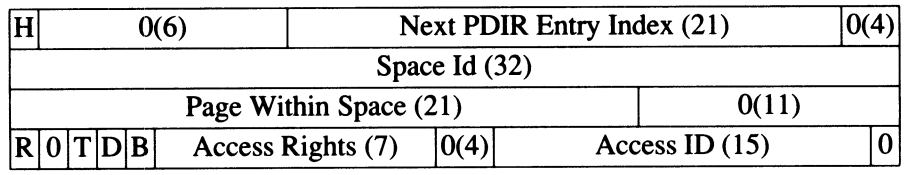


Figure 3-8. Physical Page Directory (PDIR) Entry.

The fields are:

H	indicates the end of hash chain. If H = 1, there are no more entries in the PDIR corresponding to the hash value.
Next PDIR Entry Index	is the index of the next PDIR entry that hashes to the same hash value as this one (if H = 0).
Space ID	is the space identifier of the virtual address space containing this page.
Page Within Space	identifies the page number within the virtual space to which this physical page corresponds.
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. For systems with software TLB miss handling, this bit is managed by the software and not directly set by the hardware.

Other fields correspond to those described above for TLB entries (see the earlier discussion of TLBs in this chapter).

Each virtual space is required to be an integral number of pages in length. When an entry for a virtual page exists in the PDIR, any reference to that page is valid, subject to access control validation.

The purpose of the hash algorithm is to translate virtual addresses to a smaller, more uniform name space. The algorithm is implementation-dependent. Implementations with hardware TLB updating must provide the LOAD HASH ADDRESS instruction, which computes the hash table entry address corresponding to a given virtual address.

Figure 3-9 illustrates this software translation mechanism. In this example, 64-bit virtual addressing is assumed.

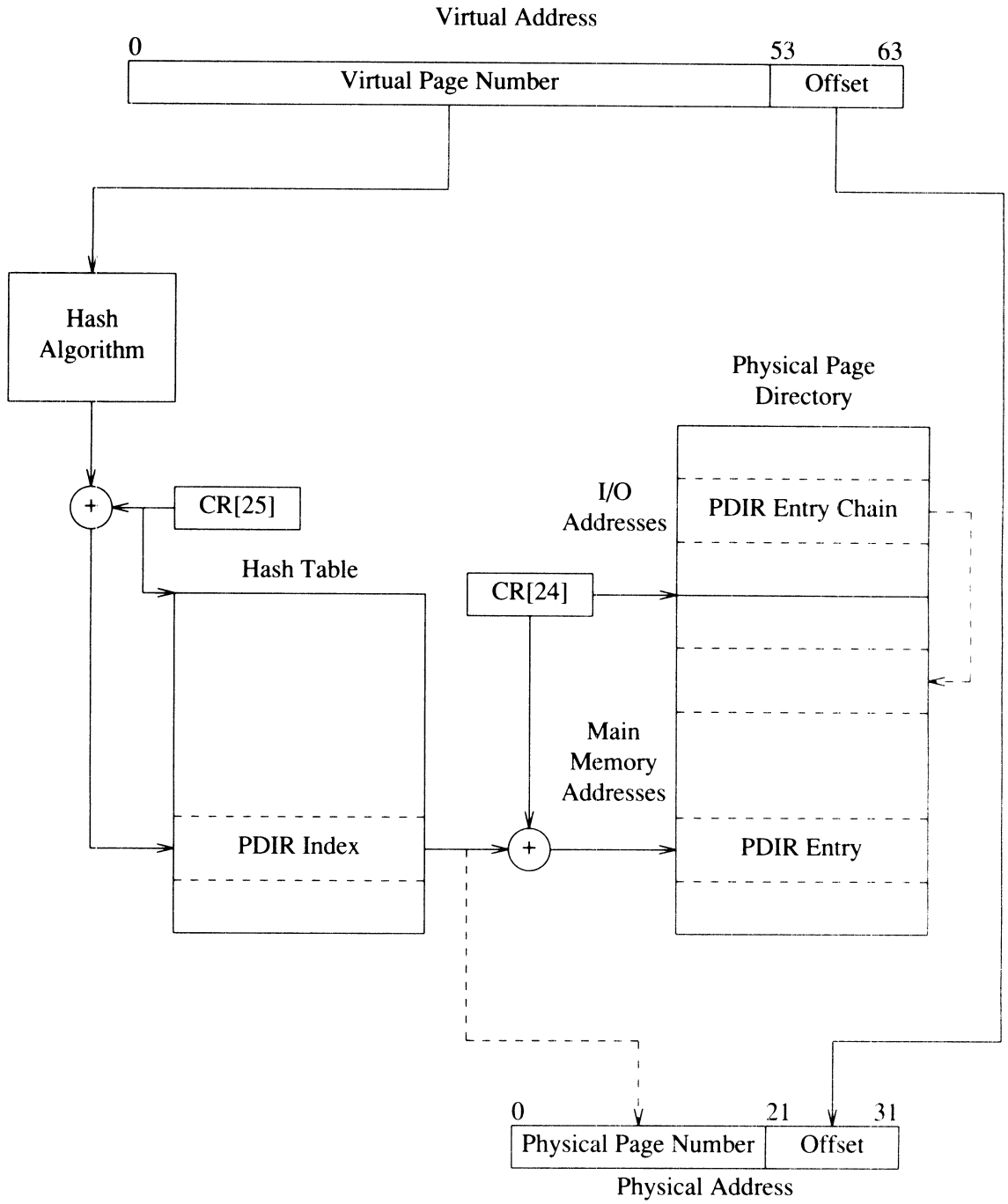


Figure 3-9. Virtual to Physical Address Translation.

Caches, Multiprocessing and I/O

All TLB's in a multiprocessor configuration 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.

A multiprocessor system may include only HP Precision Architecture processors, or may include other processors as well. The subset of processors running the standard operating system is called *symmetric*, the remainder of the system is called *asymmetric*.

The symmetric part of a multiprocessing 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 purges and flushes, except for FDCE and FICE. Such purges and flushes 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 still responsible for modifications to the code stream.

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

I/O adapters process data in memory; this data can be inconsistent with processor caches. Software is required to ensure that: 1) the contents of cache are flushed to main memory prior to an I/O output operation, and 2) the appropriate caches are purged to synchronize with in I/O input transfer.

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". A cache line which was stored into in such a way that it was unchanged may be considered dirty.

4

Control Flow and Interruptions

Introduction

The instruction execution model described in this chapter provides a logical view of the steps involved in instructions execution. The sections on nullification, branching, and interruptions show how control flow 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. The above process performs 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.

1. If the PSW M-bit is 0, then high priority machine checks (HPMCs) may occur.
2. The processor checks for:
   a. A power failure interrupt that is not masked by the PSW I-bit (External, power failure, and low priority machine check interruption unmask).
   b. A recovery counter trap. This trap is enabled when the PSW R-bit (Recovery Counter Enable) is 1 and the high 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.
3. The current instruction is fetched and executed. If a group 3 interruption occurs during execution, the processor rolls back the effect of the current instruction and take the interruption. Depending on the state of the PSW N-bit, one of two events occur:
   a. If the current instruction *is not nullified* (PSW N-bit is 0), then the instruction is fetched using the front elements of the instruction address (IA) space and offset queues. If the PSW C-bit (Code address translation enable) is set, virtual address translation of the instruction address is performed. The PSW P-bit (Protection ID validation enable) enables protection checking. On a split TLB system, the ITLB 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 set. The current instruction is executed and the PSW X-bit (Data memory break disable) is cleared. If the next instruction is nullified, the PSW N-bit (nullify) is set, 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 queue is advanced, the back of the queue is loaded with the target address, the privilege level and the offset are computed by the branch instruction, and the PSW B-bit (Taken branch in previous cycle) is set.
- For a not-taken branch: the Instruction Address queue is advanced, the back of the 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 cleared.
- If the current instruction is a RETURN FROM INTERRUPTION, the IA queues and the PSW have been updated with the new values and the following instruction is executed based on these new values.

b. If the current instruction *is nullified*, the Instruction Address queue is advanced and the back of the queue is written with the new front element plus 4. The privilege level is the same as the new front element. The PSW X-bit, N-bit, and B-bit are cleared.

4. Group 4 traps (higher-privilege transfer trap, lower-privilege transfer trap, taken branch trap) 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 (Lower-privilege transfer trap enable) was set, 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 (Higher-privilege transfer trap enable) was set, a Higher-privilege transfer trap is taken. The "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 (Taken branch trap enable) was set, then a Taken branch trap occurs.

   Group 4 traps use the PSW values for the just completed instruction. Any new PSW values, as a result of the just completed instruction, affect the following instruction.

Instruction Pipelining

The architecture permits implementations to prefetch up to seven instructions (including branch prediction) beyond the instruction currently executing. Modification of resources used for instruction fetch affect instructions that are fetched eight instructions later (worst case) or after the next RETURN FROM INTERRUPTION, whichever occurs first. Instruction fetch resources include protection ID registers, PSW, TLB entries, and cache entries.

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.
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 instruction or execute a RETURN FROM INTERRUPTION.

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 queue advances and the B-bit, N-bit, and X-bit in the PSW are set to 0). The recovery counter *is not* decremented for a nullified instruction.

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

Branching

Branches are another way of altering control flow during program execution. The processor 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 test is true.

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 4-1.

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

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 4-1. Delayed Branching.

Conditional and Unconditional Branches

The explicit branch instructions are called *unconditional* branches. In these, the branch is performed unconditionally, and is not dependent on the outcome of any test operation. The implicit branches provide a mechanism to branch *conditionally* 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 an *interspace* or *external* branches.

Target Address Computation

The target of a branch instruction, just like any instruction address, consists of a space ID and an offset. Depending on whether the branch is interspace or intraspace, a space ID calculation is performed or the space ID remains unchanged. 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 IA space queue 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 GR. 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 left shifted by two before adding to the base register. Interspace branches are always base relative.

In the case of intraspace branches, the space ID does not change. 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 left shifted 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 left shifted 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 left shifted 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 allows users to start case tables immediately following the delay instruction.

Privilege Level Changes

Branch instructions may change 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 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. 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. The *type* field is located in the TLB entry for the page from which the GATEWAY instruction is fetched.

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 always takes effect at the target instruction. The GATEWAY instruction is an exception; change of privilege level for a GATEWAY instruction can occur either at the delay slot or 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 changes are defined to take effect.

Consider the case when 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 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 code below:

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 has no relation to 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].

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, 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 two-deep queues. The *front* element points to the currently executing instruction and the *back* element points to the following instruction that will be executed. *next* is temporary element that holds the address of the next instruction that will enter the queues when they are updated. The queues are said to be updated when back enters front and next enters back.

For unconditional branches and taken conditional branches, the IA queues get updated with the target of the branch. Both the word offset and the privilege level are updated. IAOQ_Next gets the value of the target of the branch. For conditional branches that are not taken, IAOQ_Next gets IAOQ_Back + 4. The privilege level is obtained from the back element of the queue.

Consider the situation shown in Figure 4-2; 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, N-bit, and B-bit are cleared.

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 4-2. Branch in the Delay of a Branch.

Traps Associated with Branches

Branch instructions may cause various traps based on the setting of PSW bits. If the PSW T-bit is set, and a branch is taken, the *taken branch trap* occurs. This trap may be used for the purposes of debugging. If the PSW H-bit is set, and a branch instruction raises the privilege level, the *higher-privilege transfer trap* occurs. If the PSW L-bit is set, and a branch instruction lowers the privilege level, the *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, and if the next instruction is a GATEWAY, an *illegal instruction trap* occurs.

Interruptions

Interruptions are anomalies that occur during instruction processing, causing the control flow to be passed to an interruption handling routine. In the process, certain processor state saves are made automatically by the hardware. Upon completion of interruption processing, a RETURN FROM INTERRUPTION 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.

Traps, faults, checks, and interrupts are the different types 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 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 are synchronous or asynchronous with respect to the instruction stream.

All four classes of interruptions are handled in the same way. There are 25 defined interruptions which 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
Group 3:	6	Instruction TLB miss fault/instruction page fault
	7	Instruction memory protection 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
18 Data memory protection trap/Unaligned data reference trap
19 Data memory break trap
20 TLB dirty bit trap
21 Page reference 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 within each group determines the priority of simultaneous interrupts (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 Interruption PSW register (IPSW). For group 2 or 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 instruction executes.
2. The defined bits in the PSW are cleared to zero if the interruption is not a high-priority machine check (interruptions are disabled, absolute addressing mode is enabled, etc.). If the interruption is a high-priority machine check, all defined bits of the PSW except the M-bit are set to zero and the M-bit is set to 1.
3. IA information in the IIA queues is frozen (as a result of clearing 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 Offset queues, which are two entries deep, containing the addresses and privilege levels of these instructions. The IIA queues are kept up-to-date whenever the Q-bit in the PSW is set. When an interruption is taken, the address of the pending instructions are preserved in the queues. The elements of the queues may be obtained by reading the IIASQ and IIAOQ registers (CR's 17 and 18, respectively).
4. The current privilege level is set to the highest privilege level (zero).
5. 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 translation is not enabled, only the ISR's contents are undefined. The value loaded into the IOR includes all 32 bits of the offset.

6. Execution begins at the address given by:

   Interruption Vector Address + (32 * interruption_number)

   Interruption_number is a unique integer value (1 through 25) assigned to that particular interruption. Vectoring is accomplished by performing an indexed branch into the Interruption Vector Table indexed by this integer. The value in the Interruption Vector Address register (CR 14) must be aligned on a 1024-byte boundary.

Instruction Recoverability

When execution of instructions is interrupted, the maximal processor state that is required to be saved and restored is that necessary to properly 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 - i.e. 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 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, then 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. To avoid TLB misses and page faults, portions of interruption handlers must run in absolute mode and have their code and data present in memory.

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.

The scheme by which interruptions in Groups 2, 3, and 4 are signalled and processed is shown in Figure 4-3.

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.

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.

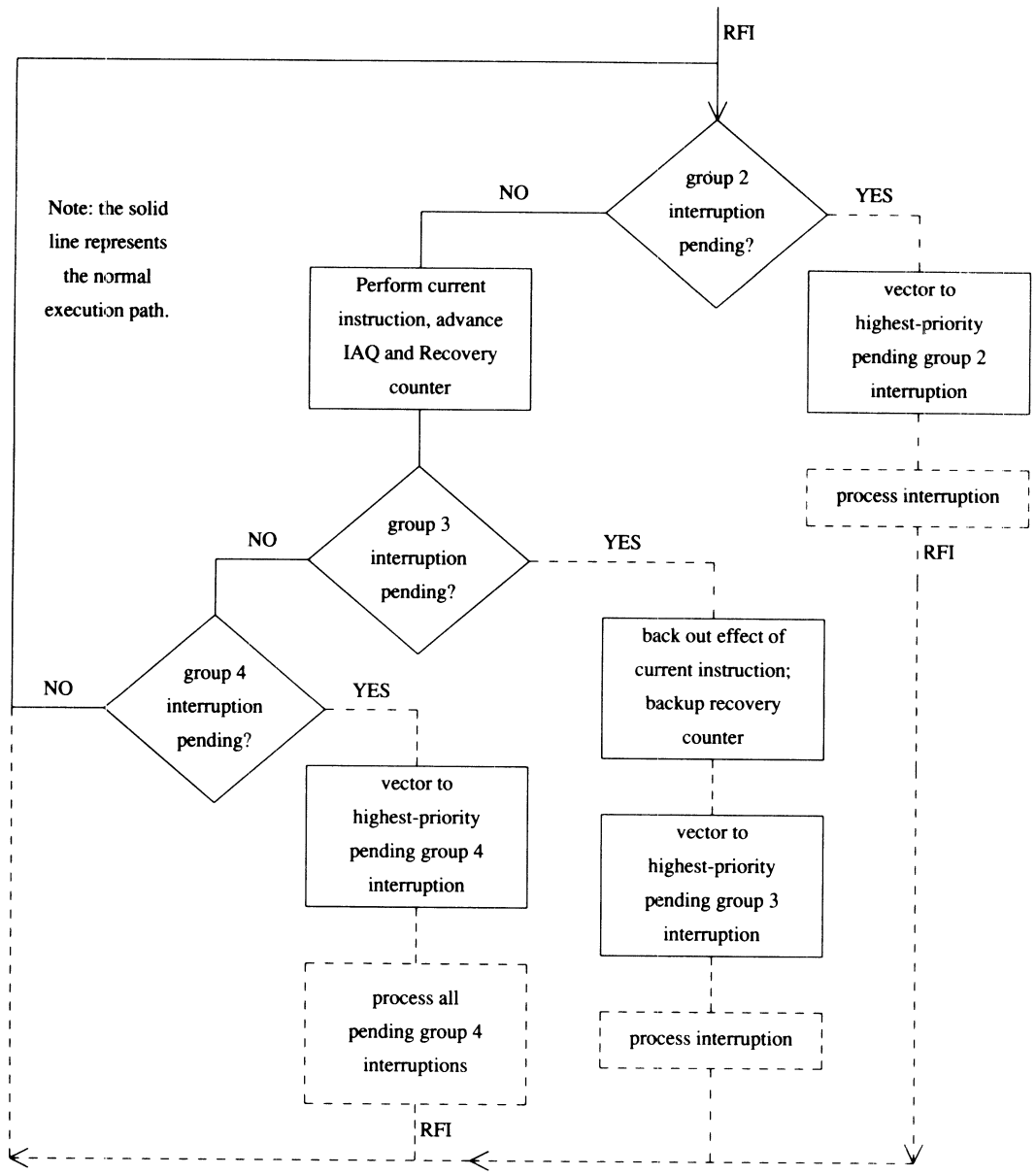


Figure 4-3. Interruption Processing.

Return from Interruption

The RETURN FROM INTERRUPTION instruction restores the PSW and instruction address queues. If the old PSW stored in IPSW (CR 22) has interruptions enabled, interruptions are re-enabled before execution of the first of the continuation instructions. The Q-bit may be turned on only by an RFI instruction. Attempting to turn on the Q-bit with an SSM or MTSM instruction is undefined. Hardware may not rely on any previous content of the IA space queues when executing with the PSW C-bit off (no code translation.)

Descriptions of Interruptions

Group 1 Interruptions

Name	**High-priority machine check** (1)
Cause	A hardware fault 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 fault is detected depend on the seriousness of the fault. 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, so that machine checks within the machine check handler can be prevented. The causes of high-priority and low-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 a 1 and PSW R 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 the broadcast IO_EIR register, or the interval timer compares equal to its associated comparator 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), one bit in the External Interrupt Request Register (CR 23). When an module causes data to be written into the EIR register, the bit position corresponding to the written value is set. If any of the corresponding bits in CR 15 are set and the PSW I-bit is set, then the external interrupt is taken. If the interrupt is masked, it is kept pending.

Interrupt handling software clears bits in the EIR 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 properly respond to all of the interrupting sources.

Name — **Low-priority machine check** (5)

Cause — A recoverable hardware fault has been detected

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 — The actions taken depend on the seriousness of the detected fault. Faults 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 high-priority and low-priority machine checks are implementation-dependent, as is the means of controlling their reporting.

Group 3 Interruptions

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

Cause — Instruction TLB entry needed by instruction fetch is absent

Parameters — none

IIA Queue — Front — Address of the instruction causing the fault

Back — Address of the following instruction

Notes — This interruption will not occur if TLB misses are handled by hardware. Instead, the same interruption vector entry is used for instruction page faults. Prefetched and nullified instructions do not cause ITLB misses. Only if an instruction is to be executed can an ITLB miss fault occur.

Name	**Instruction page fault** (6)
Cause	Page needed by instruction fetch is absent from main memory
Parameters	none
IIA Queue	Front — Address of the instruction causing the fault
	Back — Address of the following instruction
Notes	This interruption will not occur if TLB misses are handled by software. Instead, the interruption vector entry for instruction TLB misses is used.

Name	**Instruction memory protection trap** (7)
Cause	Invalid access rights or invalid protection ID and the PSW P-bit is set for an instruction fetch
Parameters	none
IIA Queue	Front — Address of the instruction causing the trap
	Back — Address of the following instruction
Notes	This trap is not detected in absolute addressing mode.

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 (taken branch) set
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. On some implementations, DIAG, LHA, and LPA may be illegal instructions.

Name	**BREAK instruction trap** (9)
Cause	Break instruction executed
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 (PL =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 current list of privileged instructions: IDTLBA, IDTLBP, IITLBA, IITLBP, PDTLB, PDTLBE, PITLB, PITLBE, MTSM, SSM, RSM, RFI, LDWAX, LDWAS, STWAS, DIAG, LHA, LPA.

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 (PL =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 MOVE TO SPACE REGISTER, MOVE TO CONTROL REGISTER, or MOVE FROM CONTROL REGISTER.

Name	**Overflow trap** (12)
Cause	An overflow is detected in a *trap on overflow* instruction
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 a *trap on condition* instruction
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 - the SFU or coprocesso instruction that was executing when an exception is reported with a trap. It may or may not be related to the condition causing the trap.
IIA Queue	Front — Address of the instruction causing the trap
	Back — Address of the following instruction

Name	**Data TLB miss fault** (15)
Cause	Data TLB entry needed by operand access of a load, store, or semaphore instruction is absent
Parameters	ISR - space identifier of data address IOR - offset of data address IIR - The instruction that caused the fault
IIA Queue	Front — Address of the instruction causing the fault Back — Address of the following instruction
Notes	This interruption does not occur if TLB misses are handled by hardware. Instead, the data page faults interruption vector is used.

Name	**Data page fault** (15)
Cause	Page needed by operand access of a load, store, semaphore instruction is absent from main memory
Parameters	ISR - space identifier of data address IOR - offset of data address IIR - The instruction that caused the fault
IIA Queue	Front — Address of the instruction causing the fault Back — Address of the following instruction
Notes	This interruption will not occur if TLB misses are handled by software. Instead, the data TLB miss fault interruption vector is used.

Name	**Non-access instruction TLB miss fault** (16)
Cause	The instruction TLB entry needed by a FLUSH INSTRUCTION CACHE instruction is not present
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	This interruption source is distinguished from other TLB misses because a page fault resulting from any of them should not result in reading the faulting page from disk. This interruption does not occur if TLB misses are handled by hardware.

Name	**Non-access data TLB miss 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, PURGE DATA CACHE, FLUSH INSTRUCTION CACHE, or a FLUSH DATA CACHE instruction

is not present

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 if TLB misses are handled by hardware. On systems with a direct mapped ITLB, the FIC instruction must use the DTLB, provided translation is performed.

Name
Non-access data page fault (17)

Cause
Page information needed by PROBE WRITE ACCESS, PROBE WRITE ACCESS IMMEDIATE, PROBE READ ACCESS, or PROBE READ ACCESS IMMEDIATE is absent from the PDIR

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
This interruption will not occur if TLB misses are handled by software. Instead, the same interruption vector is used for non-access data TLB misses.

Name
Data memory protection trap/Unaligned data reference trap (18)

Cause
An access rights check or a protection ID check that fails of an operand reference for load, store, semaphone, and PURGE DATA CACHE instructions; any load or store instruction with virtual address translation to unaligned data

Parameters
ISR - space identifier of the virtual address
IOR - offse 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
Data memory protection traps are not detected in absolute addressing mode. Only unaligned virtual memory loads and stores (include coprocessor loads and stores) are defined to terminate with this trap. Semaphore instructions with unaligned (16 byte boundaries) addresses are undefined.

Name
Data memory break trap (19)

Cause
Stores, LOAD AND CLEAR WORD INDEXED, LOAD AND CLEAR WORD SHORT, or PURGE DATA CACHE to a page with the B-bit set in the data TLB

Parameters — ISR - space identifier of virtual address
IOR - offset of 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 the PSW X-bit is set. It is not detected in absolute addressing mode or for STORE WORD ABSOLUTE SHORT instruction.

Name — **TLB dirty bit trap** (20)

Cause — Store, LOAD AND CLEAR WORD INDEXED, or LOAD AND CLEAR WORD SHORT to a page whose data TLB entry has a zero dirty bit

Parameters — ISR - space identifier of data address
IOR - offset of 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. This trap will not occur if TLB misses are handled by hardware. It is not detected in absolute addressing mode.

Name — **Page reference trap** (21)

Cause — Load, store, or semaphore instructions to a page with the T-bit set in its data TLB entry

Parameters — ISR - space identifier of virtual address
IOR - offset of virtual address
IIR - The instruction executing at the time of the trap

IIA Queue — Front — Address of the instruction causing the trap

Back — Address of the following instruction

Notes — This trap is not detected in absolute addressing mode.

Name — **Assist emulation trap** (22)

Cause — An attempt is being made to execute an SFU instruction for an unimplemented SFU or to execute a coprocessor instruction for a coprocessor whose corresponding bit in the Coprocessor Configuration Register (CR 10) is 0

Parameters — ISR - space identifier of data address
IOR - offset of data address
IIR - 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 currently completed instruction and the H-bit in the PSW is set

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 previous instruction and the L-bit in the PSW is set

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 — An unconditional branch or a taken conditional branch was executed, and the PSW T-bit was set

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 set in this case).

5

Instruction Set

Introduction

The HP Precision Architecture instruction set consists of defined, undefined, 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. Computational instructions.
5. System Control instructions.
6. Assist instructions.

Instructions are always 32 bits in length. A 6-bit major opcode is always the first field. Source registers, if specified, are always 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, additional opcode qualifiers, conditions, and nullification.

Undefined Instructions

Not all of the 64 possible major opcodes of the instruction set are defined as valid instructions. (See Appendix D 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 implementer, but system integrity must not be 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.

Attempting to execute an optional instruction that is not a special operation or a coprocessor instruction, and is not implemented in a particular processor, causes an Illegal instruction trap. 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.

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 queue advances and the PSW B-bit, N-bit, and X-bit are set to zero. Null instructions most commonly occur in level zero systems. For example, in a level zero 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. 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 branch instructions have the negation controlled by the opcode.

The 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*), if necessary.

Conditions affect control flow by checking the result of the operation performed by the current instruction and then:

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.

Table 5-1. Arithmetic/Logical Operation Conditions.

c	Description
0	never (nothing)
1	all bits zero
2	(leftmost bit equals one) xor signed overflow
3	all bits zero or (leftmost bit equals one xor signed overflow)
4	no unsigned overflow
5	all bits zero or unsigned overflow
6	signed overflow
7	rightmost bit equals one

In the above table, c is the machine language value. 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 5-2. Overflow Results.

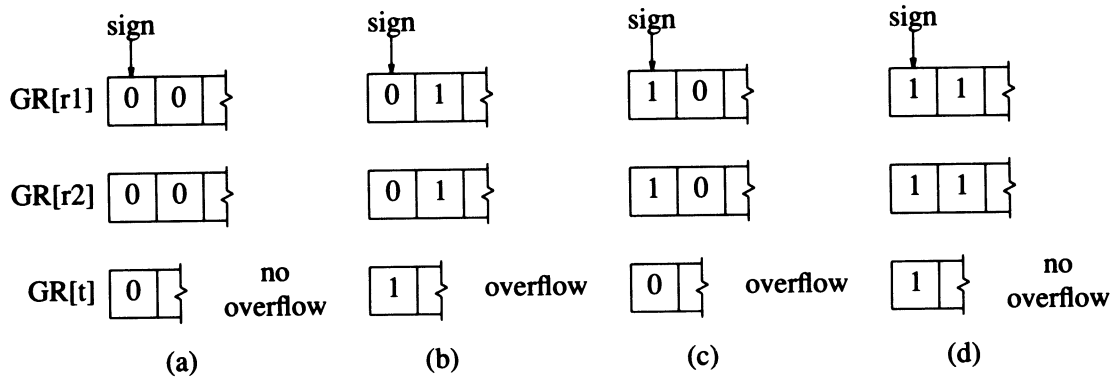
Instructions	Unsigned Overflow	Signed Overflow
Adds	The result of an unsigned additon is greater than $2^{32}-1$.	The result of signed additon is not representable in 32-bit two's complement notation (both source operands have the same sign and the sign of the 32-bit result is different).
Subtracts and Compares	The result of an unsigned subtraction is less than zero (i.e., *b* is greater than *a* in the operation *c* = *a* - *b*).	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 a one, 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 that is generated during the pre-shift operation and XORed into conditions 2 and 3, does not have to be included. 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 numbers addition and indicates the signed overflow condition when both operands are: (a) small positive numbers, (b) large positive numbers, (c) large negative numbers, or (d) 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.

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 or equal to 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

In the above table, cond is in assembly language format and c and f are in machine language format.

opd1 denotes operand 1 (an immediate or a register) in the assembly language instruction format and *opd2* denotes operand 2 (a register). The condition, <, "opd1 is less than opd2 (unsigned)" is equivalent to "no carry" in Table 5-1. In the description, the negative of a number means the two's complement of that number.

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 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, the *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 Table 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.

Table 5-5. Logical Instruction Conditions.

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

In the above table, cond is in assembly language format and c and f are in machine language format. Other values of the condition field are undefined for the logical operations.

Unit Conditions

The operations concerned with subunits 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.

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

In the above table, cond is in assembly language format and c and f are in machine language format. Other values of the condition field are undefined for the unit operations.

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.

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

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

In the above table, cond is in assembly language format and c is in machine language format.

The MOVE AND BRANCH instructions use these conditions. The BRANCH ON BIT instructions use only the "<" (bit set) and ">=" (bit clear) conditions.

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 assembler language programs, please refer to the Assembly Language Reference Manual.

Notation of queues

The operation definitions of branch instructions are described in terms of two queues. Each queue is two entries deep, one is for the offsets and the other for the space IDs. See Figure 5-1.

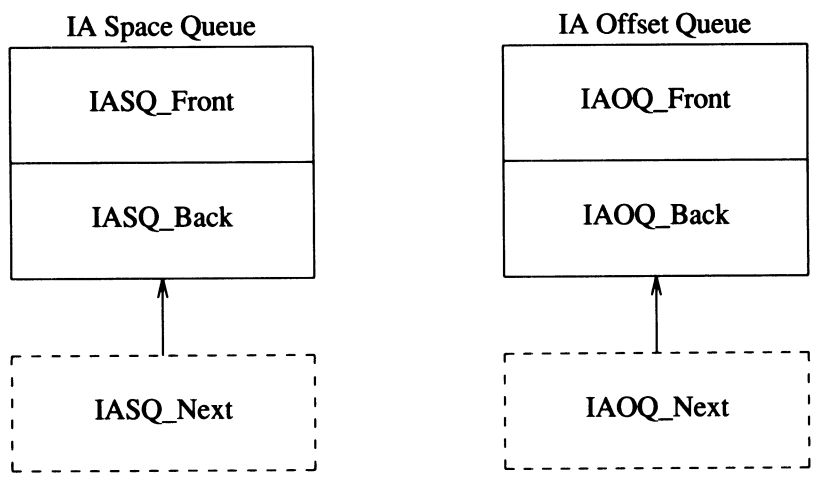


Figure 5-1. IA Space and IA Offset Queues.

The front entries of each queue correspond to the current values of the instruction offset and instruction space. The queues are shifted once for each instruction if there are no Group 1, 2 or 3 interruptions pending. The two entries of the queues are the front and back entries of the IA Offset and IA Space queues respectively. The front entries, IASQ_Front and IAOQ_Front, refer to the space and offset of the current instruction. The back entries, IASQ_Back and IAOQ_Back, refer to the space and offset of the following instruction. In addition, the two terms IAOQ_Next and IASQ_Next are used to denote the next values that will enter the IA Offset and IA Space queues respectively. These will enter the back of the

queues when the queues are advanced. The values entered depend on whether or not the current instruction is a taken branch.

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.

An exception to this occurs in the addressing of memory. For example, the range denoted by "Mem[addr]{8..15}" denotes the byte after the one denoted by "Mem[addr]", rather than a part of the same byte. See below for details.

Memory and Addressing

Main memory is denoted by the name "Mem" when it is addressed virtually and "PhysMem" when it is addressed physically.

Mem[addr], where addr is a virtual address, denotes the byte of virtual memory with address "addr". Mem[addr]{m..n} denotes a range of bits of virtual memory from the m'th bit beyond the beginning of byte Mem[addr] through the n'th bit beyond the beginning of byte Mem[addr]. Thus, for example, "Mem[X'0000340200004000]{0..31}" denotes the word of virtual memory beginning at byte offset X'4000 in the space with space ID X'03402.

PhysMem[addr] (where addr is a 32-bit absolute address) denotes the byte of physical memory beginning with address addr. The meaning of PhysMem[addr]{m..n} corresponds to that of Mem[addr]{m..n}.

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. If "regname" is a register name, then "regname{m}" denotes its m'th bit (numbering from 0 on the left). The term "regname{m..n}" denotes the range of bits of regname from the m'th through the n'th.

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

Register	Range	Description
GR[t]	t = 0..31	General registers
SR[t]	t = 0..7	Space registers
CR[t]	t = 0, 8..3	Control registers
CPR[uid][t]	t = 0..31	Coprocessor "uid" registers
FPR[t]	t = 0..15	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 zero 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:

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

All operators are binary, except that "~" is unary and "–" is both binary and unary, depending on 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 it's 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, *n1* through *xn*.
lshift(arg1,arg2)	*arg1* is logically shifted left by the number of bits specified in *arg2*.
mem_load	See *Memory Reference Instructions*.
mem_store	See *Memory Reference Instructions*.
store_in_memory(space,offset,low,high,data)	The function store_in_memory is identical to mem_store except that it forces the data to be stored into memory. The data may optionally remain in the cache.
mod(arg1,arg2)	Produces the remainder when *arg1* is divided by *arg2*.
rshift(arg1,arg2)	*arg1* is logically shifted right by *arg2* bits.
send_to_copr(u,t,priv)	Sends the 5-bit value *t* and the 2-bit privilege level *priv* 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.
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))

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));
	}

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 architectural.

Function	Description
alloc_DTLB(space,offset,entry)	Allocates an entry in the data TLB based on the space and offset arguments. The position of the entry is returned through the entry pointer.
alloc_ITLB(space,offset,entry)	Same as alloc_DTLB, except that a new entry is allocated in the instruction TLB.
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.
flush_data_cache(addr)	An address translation is performed, and if the cache line containing the address is present it is invalidated, and if dirty it is written back to main memory.
flush_data_cache_entry(addr)	If the cache line containing the address is present it is invalidated, and if dirty it is written back to main memory.
flush_instruction_cache(addr)	An address translation is performed, and if the cache line containing the address is present it is invalidated, and if dirty it is written back to main memory.

flush_instruction_cache_entry(addr)	The instruction cache entry specified by a portion of the effective address (model dependent) is flushed from the cache.
flush_data_cache_entry(addr)	The data cache entry specified by a portion of the effective address (model dependent) is flushed from the cache.
hardware_tlb_fault_handling	Boolean; set if TLB-miss handling is performed in hardware.
hash_table_entry_address(space,offset)	Returns the physical address of the hash table entry corresponding to the passed virtual address.
physical_address(space,offset)	Returns the physical address corresponding to the passed virtual address.
read_access_allowed(space,offset,x)	Determines if write access is allowed to the address, *addr*, at the privilege level given by the two rightmost bits of GR *x*.
purge_data_cache(space,offset)	If the cache line specified by the effective address is in cache, it is invalidated, no write back is performed.
purge_data_TLB_entry(space,offset)	The data TLB entry specified by a portion of the effective address (model-dependent) is invalidated.
purge_instruction_TLB_entry(space,offset)	The instruction TLB entry specified by a portion of the effective address (model-dependent) is invalidated.
read(space,offset,x)	Determines if read access is allowed to the address, *addr*, at the privilege level given by the two rightmost bits of GR *x*.
search_DTLB(space,offset,entry)	Searches the data TLB for a translation of the virtual address passed to it and returns true if it exists. As a side effect, if the translation exists, it sets the variable *entry* to point to the TLB entry containing this translation.
search_ITLB(space,offset,entry)	Same as search_DTLB, except that the instruction TLB is searched.
sfu_operation(code,opcode,priv)	An SFU specific operation is performed based on the arguments and the current state of the special function unit.
write_access_allowed(space,offset,x)	Determines if write access is allowed to the address, *addr*, at the privilege level given by the two rightmost bits of GR *x*.

Instruction Descriptions

Figure 5-2 illustrates the information presented in each of the instruction descriptions.

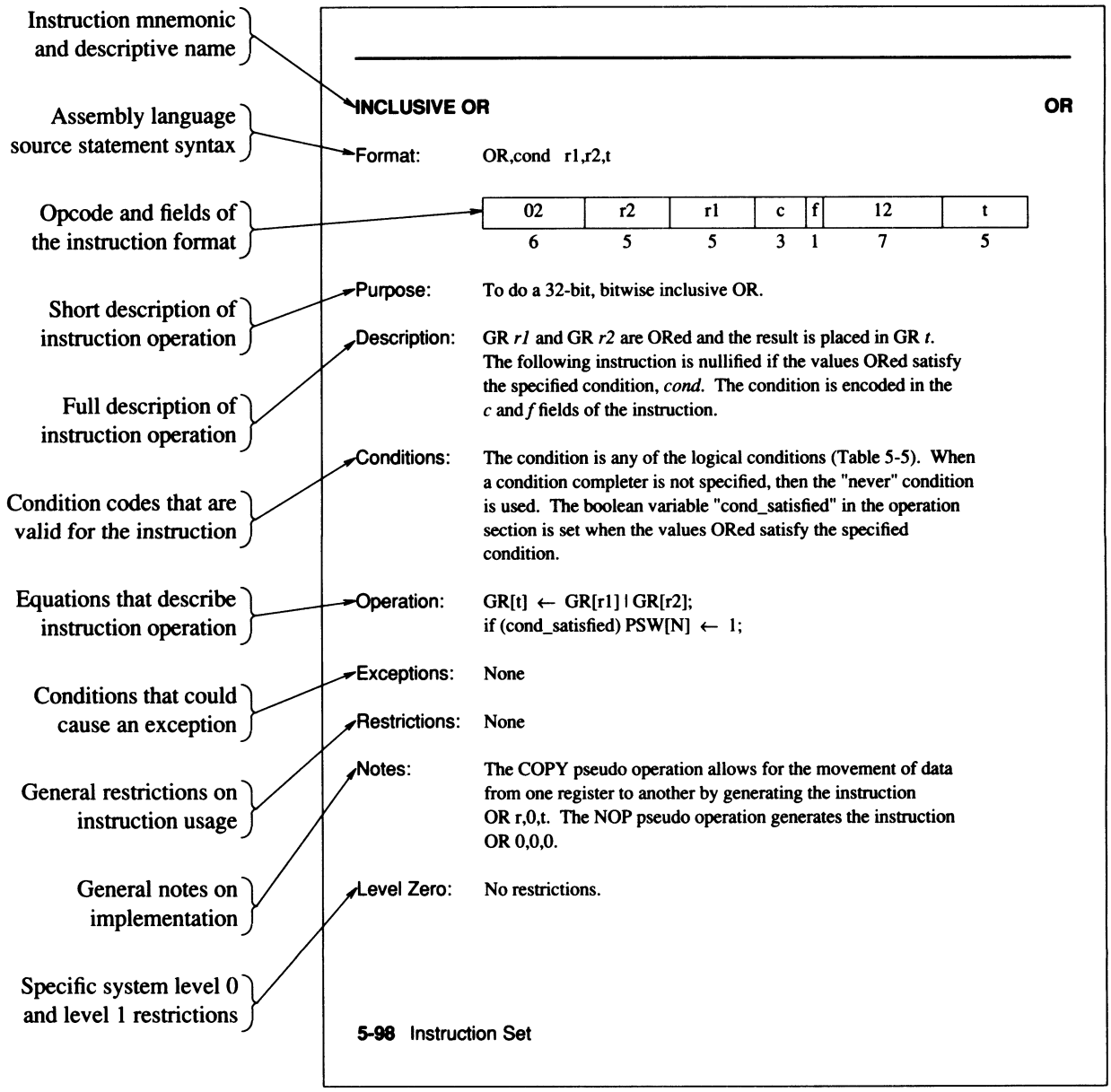


Figure 5-2. Instruction Description Example.

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 nonzero rightmost bits of computed word and halfword addresses are not ignored. Unaligned load and store instructions with data translation to halfwords, words, or doublewords cause a data memory protection/unaligned data reference trap. Semaphores and absolute addressing mode references to unaligned data are undefined operations. Only word accesses of I/O registers are defined; byte, halfword, doubleword, store bytes, and semaphore references to the I/O address space are undefined operations.

Memory reference instructions work directly between the registers and main memory. They also can operate between the registers and the D-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 for cache and TLB operations require cognizance of the D-cache (see *System Control Instructions*). The minimum architectural cache line size is 16 bytes; the maximum is 64 bytes. The maximum physical cache line size is 2048 bytes (one page).

Depending on the state of the D-bit (data address translation bit) in the PSW, all load and store instructions reference virtual memory (D-bit=1) or physical memory (D-bit=0). LOAD WORD ABSOLUTE INDEXED, LOAD WORD ABSOLUTE SHORT, and STORE WORD ABSOLUTE SHORT always reference physical memory. Memory is accessed using the following procedures:

```
mem_load(space,offset,low,high)
    {
    addr ← cat(space, offset);
    if (PSW[D] == 1)
        return(Mem[addr]{low..high});
    else
        return(PhysMem[addr{32..63}]{low..high});
    }
mem_store(space,offset,low,high,data)
    {
    addr ← cat(space, offset);
    if (PSW[D] == 1)
        Mem[addr]{low..high} ← data;
    else
        PhysMem[addr{32..63}]{low..high} ← data;
    }
```

The function store_in_memory is similar to mem_store except that it forces the data to be stored into memory. The data may optionally remain in the cache.

LOAD OFFSET does not reference memory, but saves the computed address offset in a general register. LOAD OFFSET and LOAD IMMEDIATE LEFT are not memory reference instructions.

PROGRAMMING NOTE

Execution is faster if software avoids dependence on register interlocks. Instruction scheduling to avoid the need for interlocking is recommended. This does not restrict the delay a load instruction may incur in a particular system to a single execution cycle; in fact, the delay will 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 is performed to a page with the B-bit set in its TLB entry and the X-bit off in the PSW.

Loads and Stores

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 *t* 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 transfered 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, and whether base register modification is to take place.

The format of the load and store instructions is:

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

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 zero, 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*.

The effective offset is the sum of the contents of GR *b* and the sign-extended displacement *d*, with the appropriate number of rightmost bits ignored.

The address calculation is shown in Figures 5-3 and 5-4 in two parts: one showing space identifier selection and the other, offset computation. The effective virtual address is formed by the concatenation of the space identifier and the 32-bit offset. In the absolute addressing mode, the address computation is the same; only the offset portion is used to reference memory.

NOTE

An Unaligned data reference trap is taken if the appropriate number of rightmost bit(s) of the effective virtual address are not zero for the LOAD WORD, LOAD HALFWORD, STORE WORD, and STORE HALFWORD instructions.

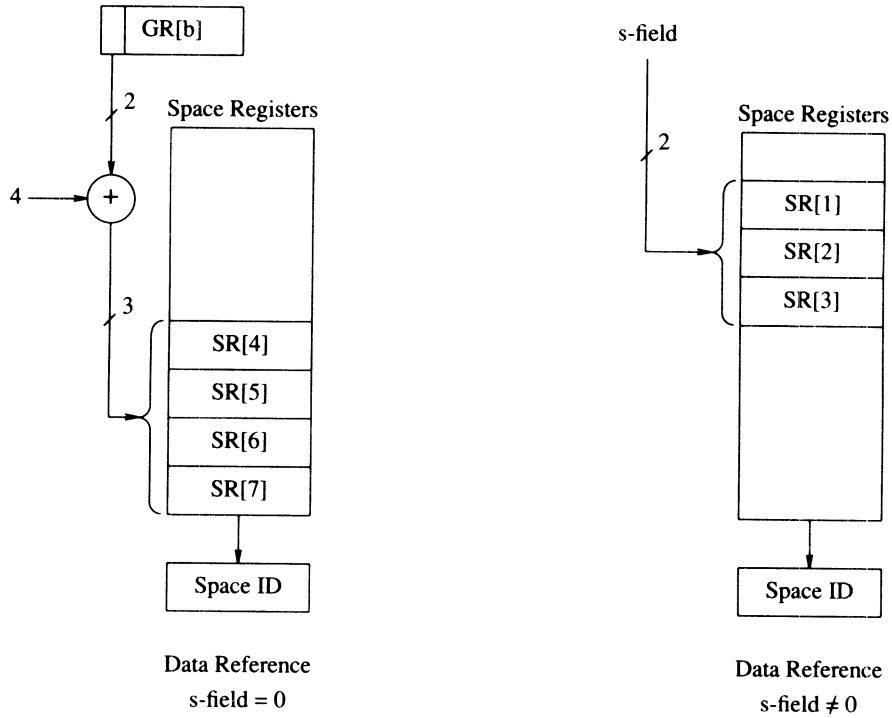


Figure 5-3. Space Identifier Selection.

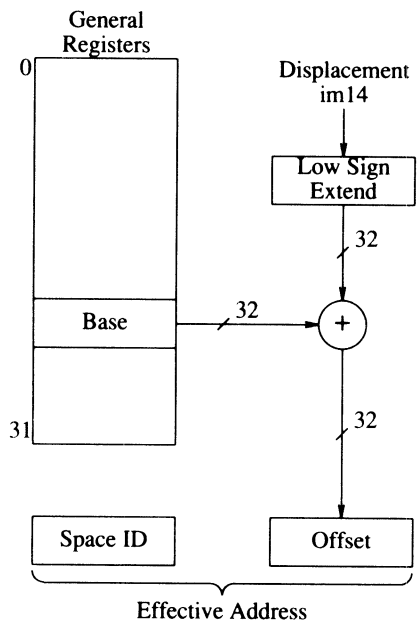
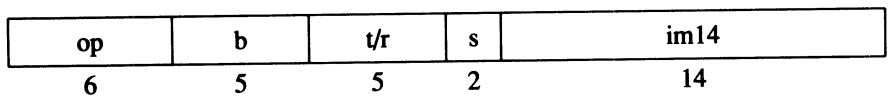


Figure 5-4. Loads and Stores.

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 *t* 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 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:



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 offset computation is shown in Table 5-8. The effective virtual address is formed by the concatenation of the 32-bit space identifier and the 32-bit offset.

NOTE

An Unaligned data reference trap is taken if the rightmost two bits of the effective virtual address are not zero for the LOAD WORD AND MODIFY and STORE WORD AND MODIFY instructions.

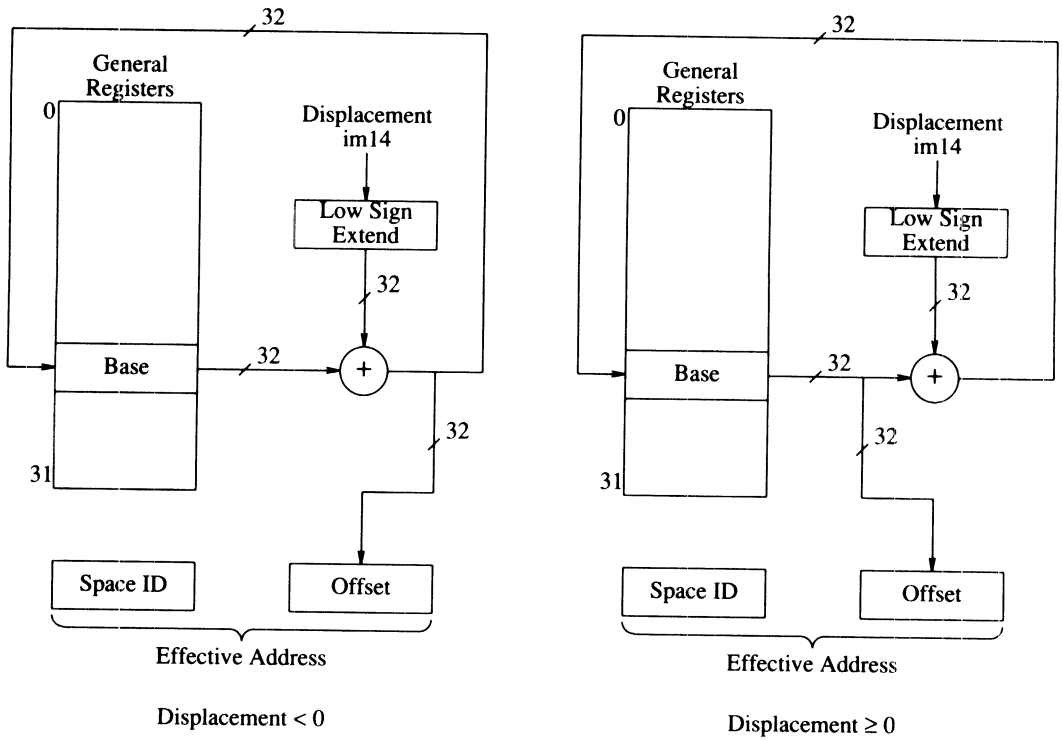
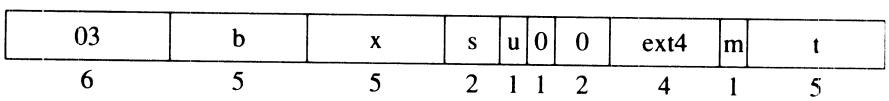


Figure 5-5. Load and Store Word Modify.

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 through 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:



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*.

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 5-8.

Table 5-8. Indexed Load and Store Completers.

cmplt	Description	u	m
	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	Shift index by data size, modify base register	1	1
,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. The calculation of the offset portion of the effective address for different completers is shown in Figure 5-6.

In absolute addressing mode, the address computation is the same, but only the offset portion is used to reference memory.

NOTE

An Unaligned data reference trap is taken if the appropriate number of rightmost bits of the effective virtual address are not zero for the LOAD WORD INDEXED and LOAD HALFWORD INDEXED instructions.

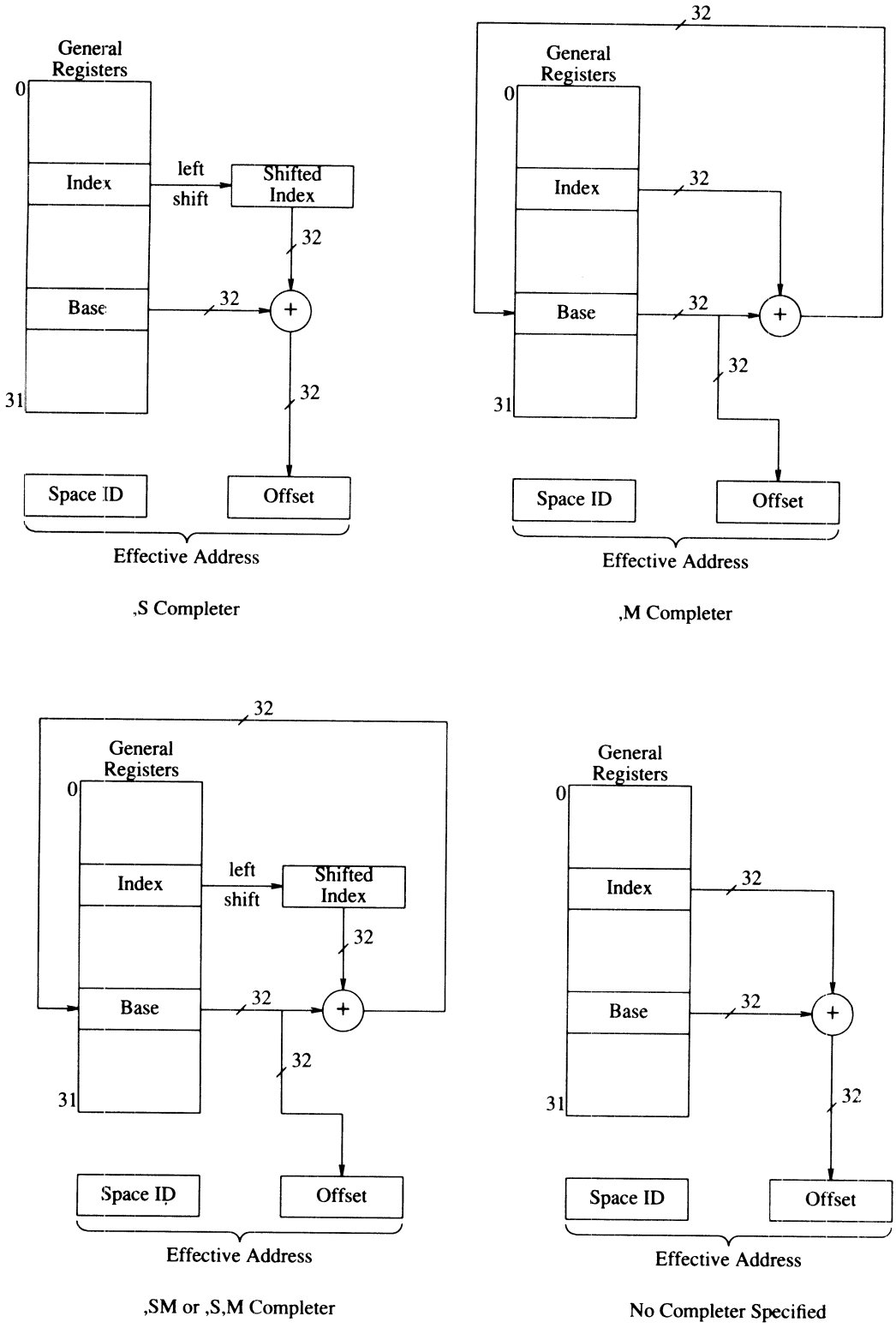
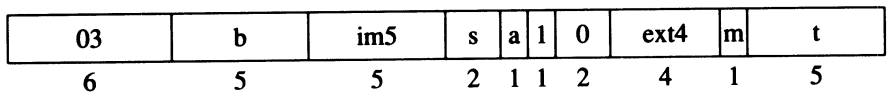


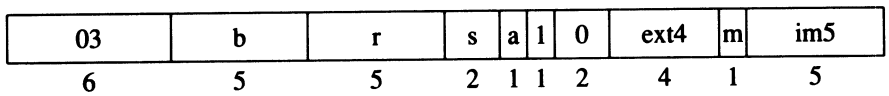
Figure 5-6. Effective Address Computation For Indexed Loads.

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. The format of the short displacement load instructions is:



and that of the short displacement stores is:



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.

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 5-9 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 5-9. Short Displacement Load and Store Completers.

cmplt	Description	a	m
	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 u and m are in machine language format. In absolute addressing mode, the address computation is the same, but only the offset portion is used to reference memory.

PROGRAMMING NOTE

An Unaligned data reference trap is taken if the appropriate number of rightmost bits of the effective virtual address are not zero for the LOAD WORD SHORT, LOAD HALFWORD SHORT, STORE WORD SHORT, and STORE HALFWORD SHORT instructions.

The space identifier is computed like any other data memory reference. The calculation of the offset portion of the address for different completers is shown in Figure 5-7.

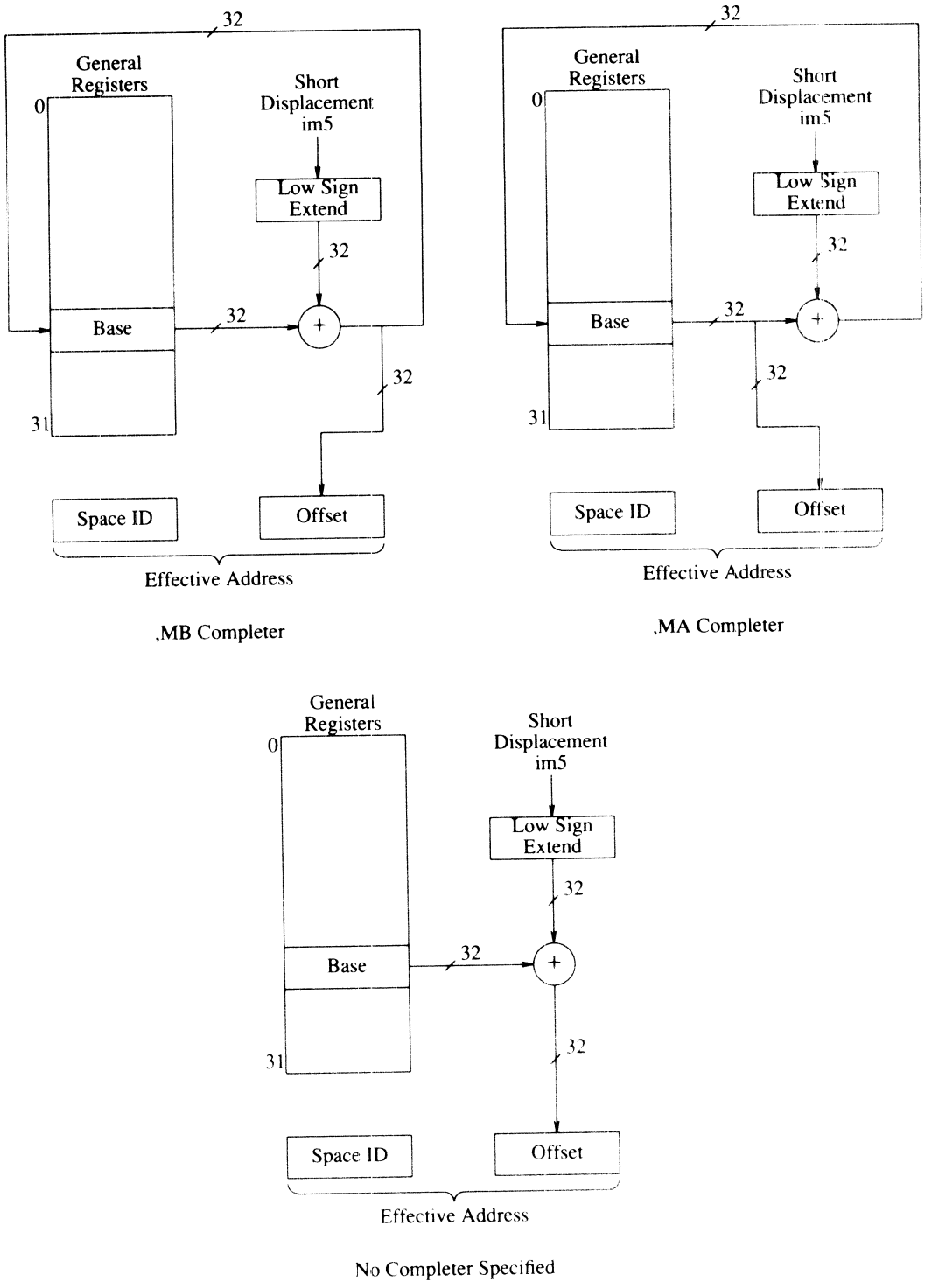
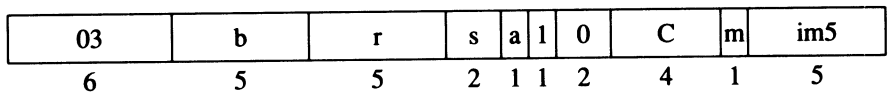


Figure 5-7. Short Displacement Loads and Stores.

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:



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.

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 5-10 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 5-10. Store Bytes Short Completers.

cmplt	Description	a	m
	Beginning case, don't modify base register	0	0
,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. In absolute addressing mode, the address computation is the same, but only the offset portion is used to reference memory. The space identifier is computed like any other data memory reference. The calculation of the offset portion of the address for different completers is shown in Figure 5-8.

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

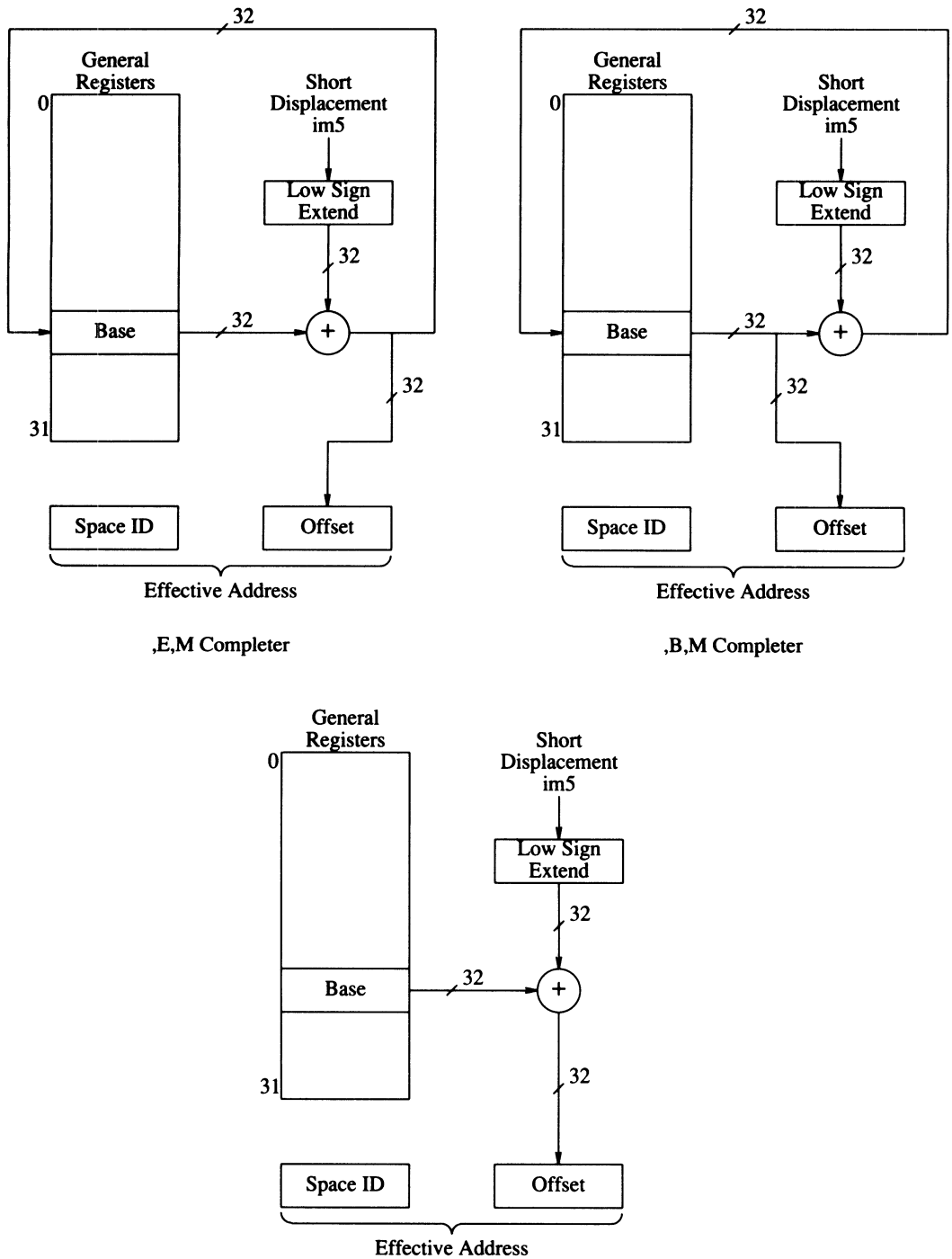
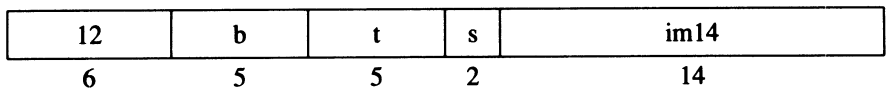


Figure 5-8. Store Bytes Short.

LOAD WORD

LDW

Format: LDW d(s,b),t



Purpose: To load a word into a general register.

Description: The aligned word 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:

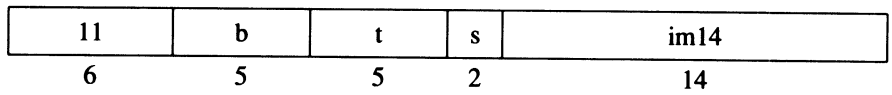
```
offset ← GR[b] + low_sign_ext(im14,14);
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
GR[t] ← mem_load(space,offset,0,31);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

LOAD HALFWORD

LDH

Format: LDH d(s,b),t



Purpose: To load a halfword into a general register.

Description: The aligned halfword 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.

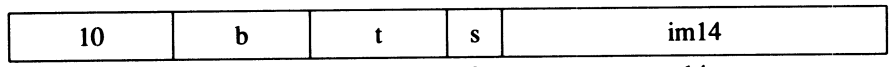
Operation:

```
offset ← GR[b] + low_sign_ext(im14,14);
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
GR[t] ← zero_ext(mem_load(space,offset,0,15),16);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

LOAD BYTE

Format: LDB d(s,b),t



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);
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
GR[t] ← zero_ext(mem_load(space,offset,0,7),8);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap
Page reference trap

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.

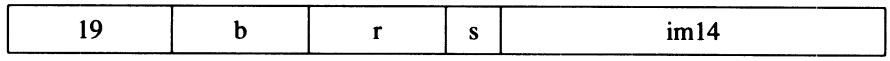
Description: GR *r* is stored in the aligned word at the effective address. The base register, *b*, plus displacement, *d*, forms the offset. The displacement is encoded into the *im14* field.

Operation:
```
offset ← GR[b] + low_sign_ext(im14,14);
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
mem_store(space,offset,0,31,GR[r]);
```

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

STORE HALFWORD STH

Format: STH r,d(s,b)



19	b	r	s	im14
6	5	5	2	14

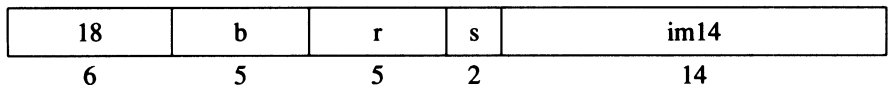
Purpose: To store a halfword from a general register.

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);
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
mem_store(space,offset,0,15,GR[r]{16..31});
```

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

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.

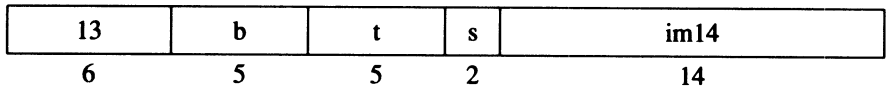
Operation:
```
offset ← GR[b] + low_sign_ext(im14,14);
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
mem_store(space,offset,0,7,GR[r]{24..31})
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap
Data memory break trap
TLB dirty bit fault
Page reference 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 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.

Operation:

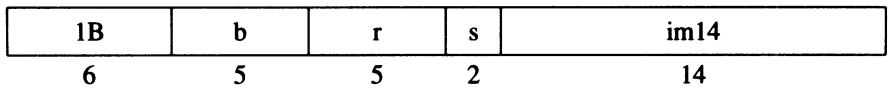
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
if (low_sign_ext(im14,14) < 0)
   offset ← GR[b] + low_sign_ext(im14,14);
else
   offset ← GR[b];
GR[b] ← GR[b] + low_sign_ext(im14,14);
GR[t] ← mem_load(space,offset,0,31);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

Restrictions: The value loaded is undefined if $b = t$.

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.

Operation:

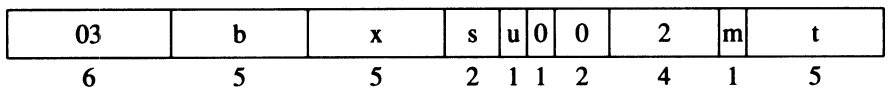
```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (low_sign_ext(im14,14) <0)
    offset ← GR[b] + low_sign_ext(im14,14);
else
    offset ← GR[b];
GR[b] ← GR[b] + low_sign_ext(im14,14);
mem_store(space,offset,0,31,GR[r]);
```

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

Restrictions: The value stored is undefined if $b = r$.

LOAD WORD INDEXED LDWX

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



Purpose: To load a word into a general register.

Description: The aligned word is loaded into GR *t* from 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 5-8 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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);
```

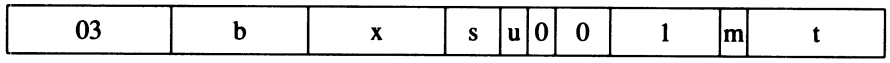
Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

LOAD HALFWORD INDEXED

LDHX

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



Purpose: To load a halfword into a general register.

Description: The aligned halfword is zero-extended and loaded into GR *t* from 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 1. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-8 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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),16);
```

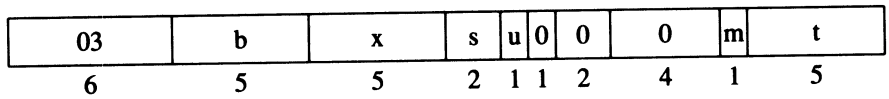
Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

LOAD BYTE INDEXED

LDBX

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



Purpose: To load a byte into a general register.

Description: The byte from the effective address is zero-extended and loaded into GR *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 or the base register plus the index register. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification. (See Table 5-8 for the assembly language completer mnemonics.)

Operation:

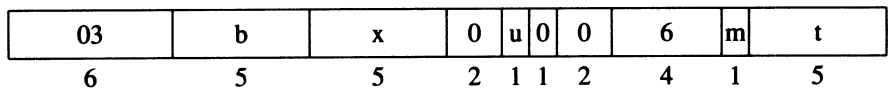
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',S':     offset ← GR[b] + GR[x];                    /*u=1, m=0*/
                  break;
   case ',M':     offset ← GR[b];                            /*u=0, m=1*/
                  GR[b] ← GR[b] + GR[x];
                  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),8);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

LOAD WORD ABSOLUTE INDEXED — LDWAX

Format: LDWAX,cmplt x(b),t



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

Description: The aligned word in physical memory from the absolute address given by the offset is loaded into GR *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. This operation is only defined if the address is aligned on a 4 byte boundary. (See Table 5-8 for the assembly language completer mnemonics.)

Protection is not checked when this instruction is executed.

Operation:

```
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] ← PhysMem[offset];
```

Exceptions: Privileged operation trap

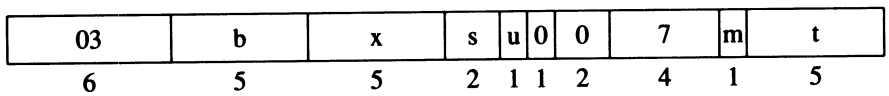
Restrictions: This instruction may be executed only by code running at the most privileged level. The value loaded is undefined if base register modification is specified and $b = t$.

Notes: On level zero systems, this instruction functions identically to LOAD WORD INDEXED.

LOAD AND CLEAR WORD INDEXED

LDCWX

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



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 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 5-8 for the assembly language completer mnemonics.)

If the cache line containing that address is present in the data cache and is dirty, the line is written back to main memory. The addressed word in main memory is copied into GR *t* then cleared to zero. The actions after the effective address calculation are indivisible; the remaining steps of this instruction are non-interruptible. This operation is only defined if the address is aligned on a 16-byte boundary.

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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;
   }
addr ← cat(space, offset);

(indivisible)
flush_data_cache(addr);
GR[t] ← mem_load(space,offset,0,31);
store_in_memory(space,offset,0,31,0);
```

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

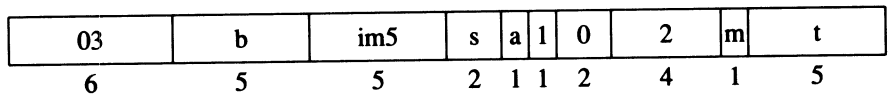
Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

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

LOAD WORD SHORT

LDWS

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



03	b	im5	s	a	1	0	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 is loaded into GR *t* from 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 5-9 for the assembly language completer mnemonics.)

Operation:

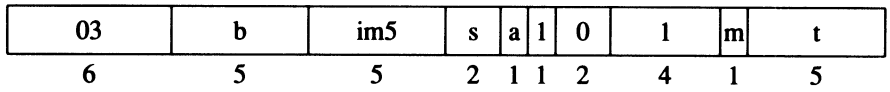
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);      /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                            /*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);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

LOAD HALFWORD SHORT — LDHS

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



Purpose: To load a halfword into a general register.

Description: The aligned halfword is zero-extended and loaded into GR *t* from 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 5-9 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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),16);
```

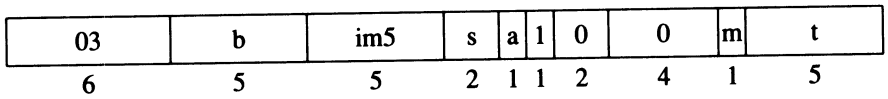
Exceptions: Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

LOAD BYTE SHORT

LDBS

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



Purpose: To load a byte into a general register.

Description: The byte from 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 5-9 for the assembly language completer mnemonics.)

Operation:

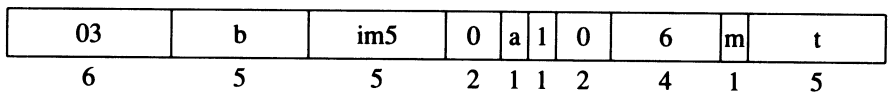
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                              /*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),8);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

LOAD WORD ABSOLUTE SHORT — LDWAS

Format: LDWAS,cmplt d(b),t



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

Description: The aligned word in physical memory is loaded into GR *t* from the absolute address given by the offset. 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 5-9 for the assembly language completer mnemonics.)

Protection is not checked when this instruction is executed.

Operation:

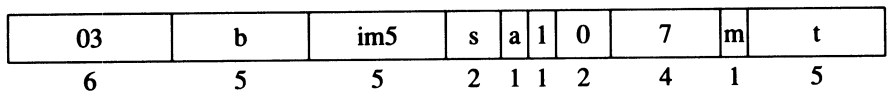
```
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);      /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                            /*a=0, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   default:      offset ← GR[b] + low_sign_ext(im5,5);      /*m=0*/
                 break;
   }
GR[t] ← PhysMem[offset];
```

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only by code running at the most privileged level. The value loaded is undefined if base register modification is specified and $b = t$.

LOAD AND CLEAR WORD SHORT — LDCWS

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



03	b	im5	s	a	1	0	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. The completer, encoded in the *a* and *m* fields of the instruction, also specifies base register modification. (See Table 5-9 for the assembly language completer mnemonics.)

If the cache line containing that address is present in the data cache and dirty, the line is written back to main memory. The addressed word in main memory is copied into GR *t* and then cleared to zero. The actions after the effective address calculation are indivisible; the remaining steps of the instruction are non-interruptible. This operation is only defined when the address is on a 16-byte boundary.

Operation:

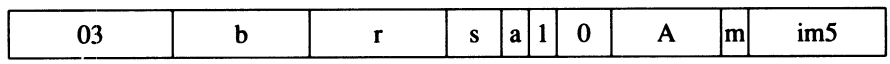
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                              /*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;
   }
addr ← cat(space,offset);

indivisible
  flush_data_cache(addr);
  GR[t] ← mem_load(space,offset,0,31);
  store_in_memory(space,offset,0,31,0);
```

Exceptions: Data TLB miss fault/data page fault
Data memory protection trap
Data memory break trap

TLB dirty bit fault
Page reference trap

Restrictions: The value loaded is undefined if base register modification is specified and $b = t$.

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



03	b	r	s	a	1	0	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 5-9 for the assembly language completer mnemonics.)

Operation:

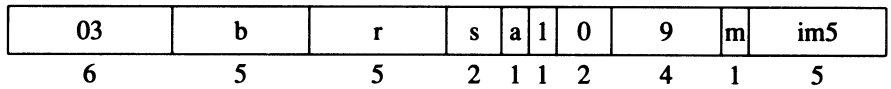
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);      /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                            /*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,GR[r]);
```

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

Restrictions: The value stored is undefined if base register modification is specified and $b = r$.

STORE HALFWORD SHORT — STHS

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



03	b	r	s	a	1	0	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 5-9 for the assembly language completer mnemonics.)

Operation:

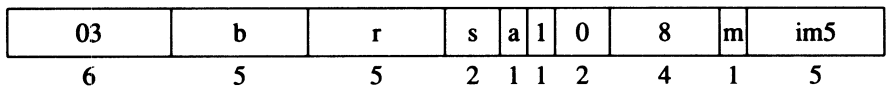
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);      /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                            /*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,0,15,GR[r]{16..31});
```

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

Restrictions: The value stored is undefined if base register modification is specified and $b = r$.

STORE BYTE SHORT STBS

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



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 5-9 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,0,7,GR[r]{24..31});
```

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

Restrictions: The value stored is undefined if base register modification is specified and $b = r$.

STORE WORD ABSOLUTE SHORT — STWAS

Format: STWAS,cmplt r,d(b)

03	b	r	0	a	1	0	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 word in physical memory at the absolute address given by the offset. 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 5-9 for the assembly language completer mnemonics.)

Protection is not checked when this instruction is executed.

Operation:

```
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;
    }
PhysMem[offset] ← GR[r];
```

Exceptions: Privileged operation trap

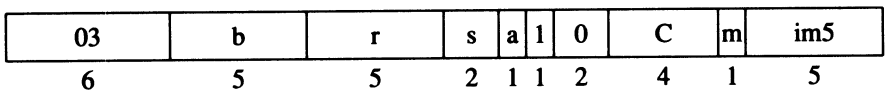
Restrictions: This instruction may be executed only by code running at the most privileged level. The value stored is undefined if base register modification is specified and $b = r$.

Notes: On level zero systems, this instruction function identically to STORE WORD SHORT.

STORE BYTES SHORT

STBYS

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



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

Description: If 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 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 still checked.

If base register modification is specified through completer "M", GR *b* is updated and then truncated to a word address. (See Table 5-10 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (cmplt == ',B,M')
    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*/
                mem_store(space,offset,pos,31,GR[r]{pos..31});
                break;
    case ',E':                                                    /*a=1, m=0*/
                mem_store(space,offset,0,pos-1,GR[r]{0..pos-1});
                break;
    case ',B,M':                                                  /*a=0, m=1*/
                mem_store(space,offset,pos,31,GR[r]{pos..31});
                GR[b] ← (GR[b] + low_sign_ext(im5,5)) & 0xFFFFFFFC;
                break;
    case ',E,M':                                                  /*a=1, m=1*/
                mem_store(space,offset,0,pos-1,GR[r]{0..pos-1});
                GR[b] ← (GR[b] + low_sign_ext(im5,5)) & 0xFFFFFFFC;
                break;
}
```

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

Restrictions: The value stored is undefined if base register modification is specified and $r = b$.

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.

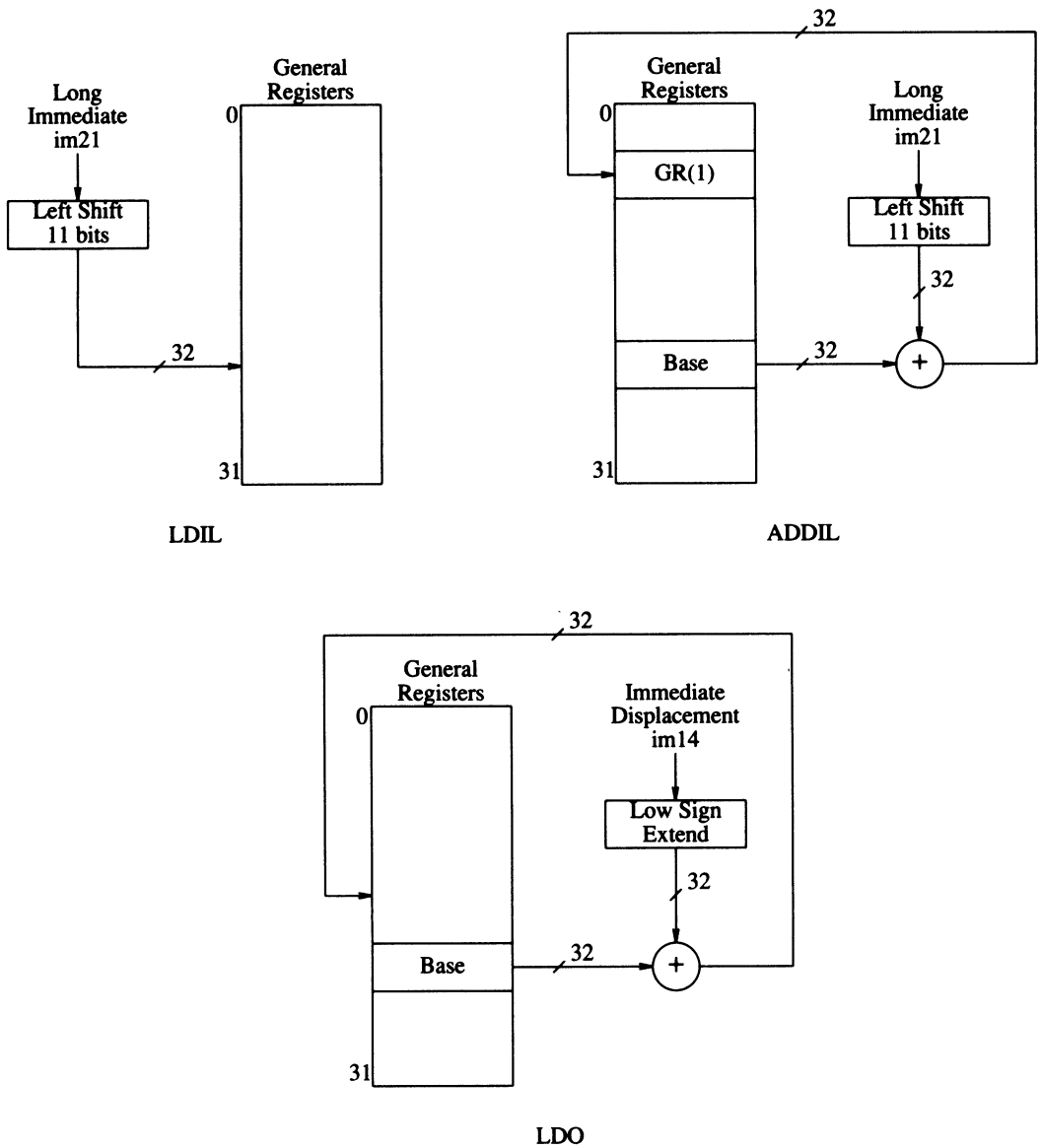
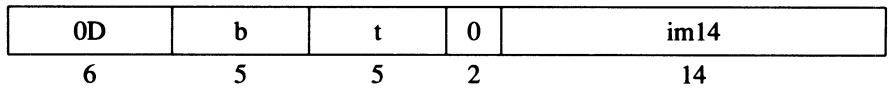


Figure 5-9. Immediate Instructions.

LOAD OFFSET LDO

Format: LDO d(b),t



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.

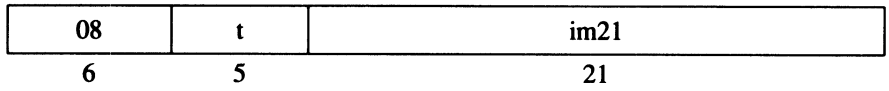
Operation: GR[t] ← GR[b] + low_sign_ext(im14,14);

Exceptions: None

Notes: Memory is not referenced so TLB and protection faults are not possible. The LDI 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



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] ← lshift(assemble_21(im21),11);

Exceptions: None

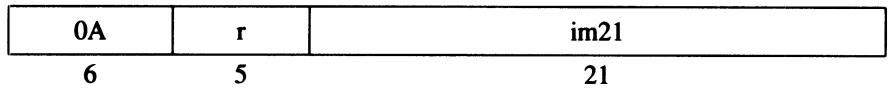
Notes: Memory is not referenced.

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
```

Format: ADDIL i,r



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] ← 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(S,GR1),GRt
```

Branch Instructions

Branch instructions are classified into three major categories: unconditional local branches, unconditional external branches, and conditional branches. Within these categories there is sub-classification based on how the target address is computed, whether or not a return address is returned, and whether or not privilege changes can occur. Not all of the options are available for each category. The following sections describe the branch types. 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. The IA-relative with static displacement uses the current IA offset plus a 17-bit signed word displacement. This allows for a plus or minus 256k byte branch target range within a space.
2. The IA-relative with dynamic displacement uses the current IA offset plus a shifted index register.
3. The base-relative with dynamic displacement uses 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 IA 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 base-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 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 instruction 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 256k byte branch range across space boundaries.

BRANCH AND LINK EXTERNAL is used for interspace procedure calls. It places the IA offset of the return point in GR 31 and copies IA space 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.

Conditional Branches

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

There are four categories of conditional 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 the 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 5-10 categorizes the characteristics of the branch instructions.

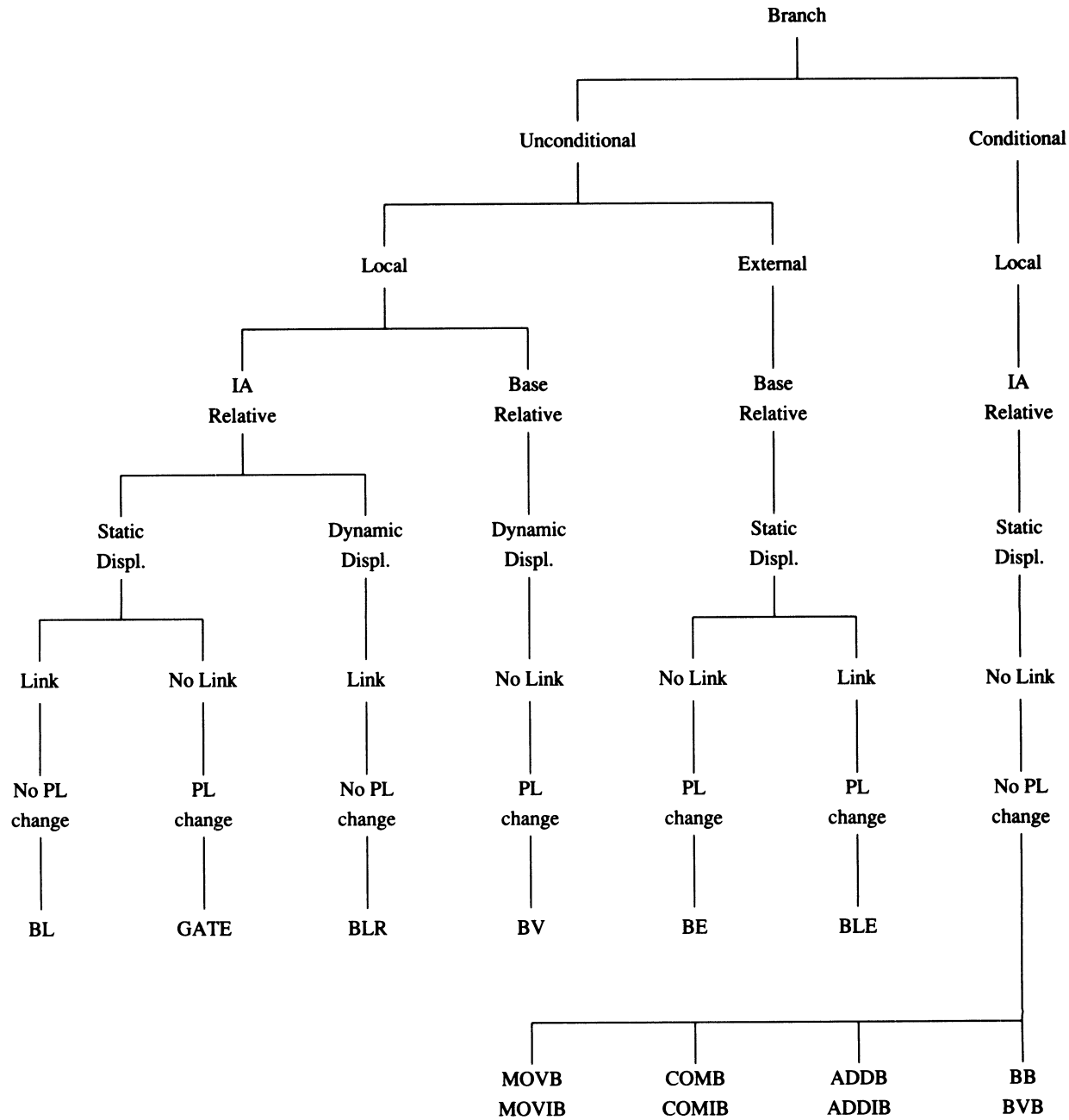


Figure 5-10. Classification of Branch Instructions.

PROGRAMMING NOTE

The following shows example instruction sequences which perform the different types of procedure calls. 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 IA space is observed:

```
call:   BL      target,rp               or      LDIL    L%target,rp
        BLE     R%target(GRr,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:

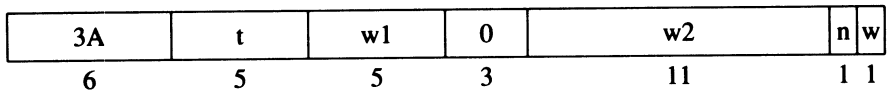
```
call:   LDW     space_id,GR1            ; space id for target
        MTSP    GR1,SR4
        LDIL    L%target,rp
        BLE     target(SR4,rp)
        OR      GR31,0,rp

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



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 offset to form the target offset. The offset of the return point is placed in GR *t*. The return point is the 4 bytes beyond the following instruction.

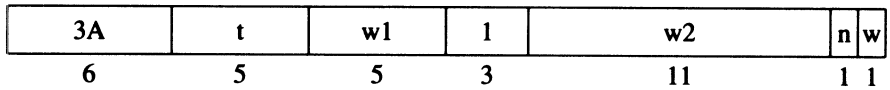
The instruction following the BRANCH AND LINK will be executed unless nullification is requested. The branch target, *target*, in the assembly language format is used by the assembler to derive the *w*, *w1*, and *w2* fields.

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.

Format: GATE,n target,t



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.

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.

An Illegal instruction trap is taken if a GATEWAY instruction and the PSW B-bit is set.

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});
    tmp ← 0;
    if (PSW[C])
        if (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

PROGRAMMING NOTE

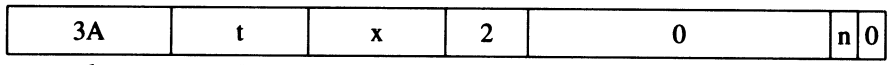
The privilege level checking for fetching and executing the instruction following a GATEWAY (which might be in a different page from the GATEWAY itself) may be done against either the old or the new privilege level. Software should ensure that both checks are equally valid.

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.

BRANCH AND LINK REGISTER

BLR

Format: BLR,n x,t



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 offset 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;
```

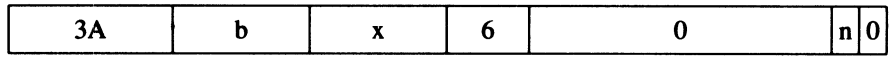
Exceptions: Taken branch trap

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. The addition of 8 to the shifted index is for the case that the index contains zero.

BRANCH VECTORED BV

Format: BV,n x(b)



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 delay-slot instruction.

Operation:

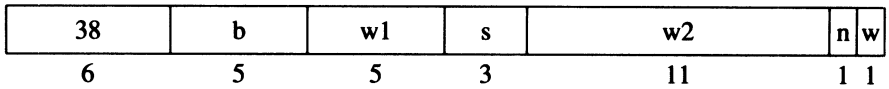
```
IAOQ_Next{0..29} ← (GR[b] + lshift(GR[x],3)){0..29};
if (IAOQ_Front{30..31} < GR[b]{30..31})
   IAOQ_Next{30..31} ← GR[b]{30..31};
else
   IAOQ_Next{30..31} ← IAOQ_Front{30..31};
if (n) PSW[N] ← 1;
```

Exceptions: Taken branch trap

Level Zero: On level zero systems, this instruction demotes the privilege level to any nonzero value, if it changes it.

BRANCH EXTERNAL — BE

Format: BE,n wd(sr,b)



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 offse of the target instruction. SR *sr* (which is assembled from the *s* field of the instruction) becomes the space ID 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 delay-slot instruction. When a BRANCH EXTERNAL is executed in absolute addressing mode, 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

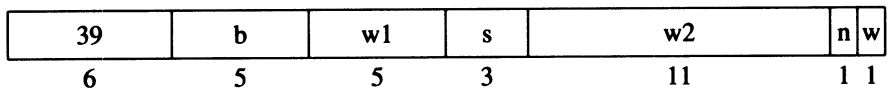
Level Zero: On level zero systems, this instruction executes as usual. The IASQ is a nonexistent register and updating has no effect. Also, privilege level is demoted to any nonzero value, if the instruction changes it.

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 IA Space 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)



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 delay-slot instruction. When a BRANCH AND LINK EXTERNAL is executed in absolute addressing mode,
the effects on IASQ and SR 0 are not defined.

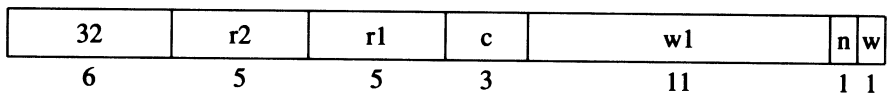
Operation:

```
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 Zero: On level zero systems, this instruction executes as usual. The IASQ and SR 0 are non existent entities and updating has no effect. Also, privilege level is demoted to any nonzero value, if the instruction changes it.

Format: MOVB,cond,n r1,r2,target



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.

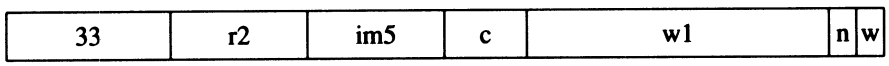
Conditions: The condition, *cond*, is any of the extract/deposit conditions shown in Table 5-7 (=, <, OD, TR, <>, >=, EV). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the value moved satisfies the specified condition.

Operation:

```
GR[r2] ← GR[r1];
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
 if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Format: MOVIB,cond,n i,r2,target



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 *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. When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the value moved satisfies the specified condition.

Operation:

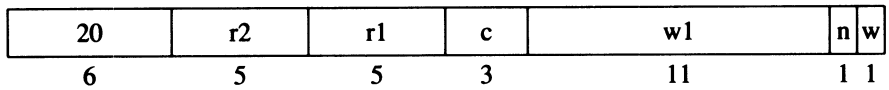
```
GR[r2] ← low_sign_ext(im5,5);
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

PROGRAMMING NOTE

Note that, since *i* is known at the time a MOVE IMMEDIATE AND BRANCH instruction is written, conditions other than "TR" and the empty condition are of no use.

Format: COMBT,cond,n r1,r2,target



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, 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 non-negated compare or subtract conditions shown in Table 5-3 (never, =, <, <=, << <<=, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values compared satisfy the specified condition.

Operation:

```
GR[r1] + (~GR[r2]) + 1;
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

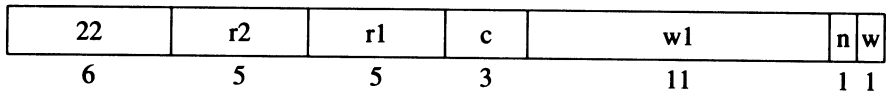
Exceptions: Taken branch trap

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



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, 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 non-negated (f = 0) compare or subtract conditions shown in Table 5-3 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values compared fail to satisfy the specified condition.

Operation:

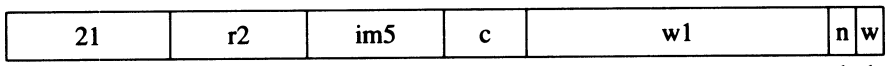
```
GR[r1] + (~GR[r2]) + 1;
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

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 IMMEDIATE AND BRANCH IF TRUE — COMIBT

Format: COMIBT,cond,n i,r2,target



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 *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, 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 non-negated compare or subtract conditions shown in Table 5-3 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values compared satisfy the specified condition.

Operation:

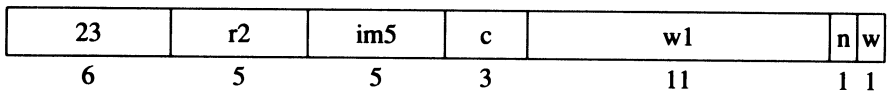
```
low_sign_ext(im5,5) + (~GR[r2]) + 1;
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

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,r2,target



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 withGR *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, 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 non-negated (f = 0) compare or subtract conditions shown in Table 5-3 (never, =, <, <=, <<, <<=, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values compared fail to satisfy the specified condition.

Operation:

```
low_sign_ext(im5,5) + (~GR[r2]) + 1;
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

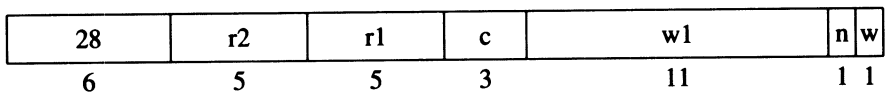
Exceptions: Taken branch trap

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



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, 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 non-negated add conditions shown in Table 5-4 (never, =, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

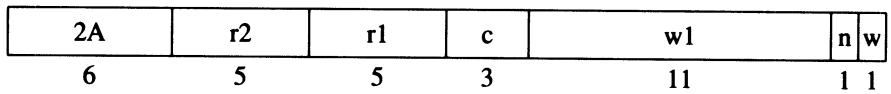
```
GR[r2] ← GR[r1] + GR[r2];
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

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 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, 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 non-negated (f = 0) add conditions shown in Table 5-4 (never, =, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values compared fail to satisfy the specified condition.

Operation:

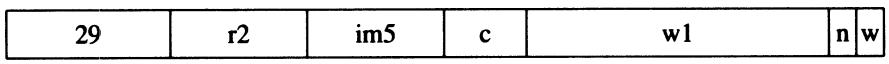
```
GR[r2] ← GR[r1] + GR[r2];
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

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,r2,target



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 *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, 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 non-negated add conditions shown in Table 5-4 (never,=, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values added satisfy the specified condition.

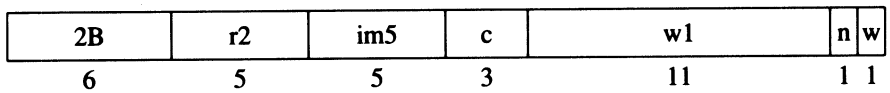
Operation:

```
GR[r2] ← low_sign_ext(im5,5) + GR[r2];
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    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.

Format: ADDIBF,cond,n i,r2,target



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 *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, 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 non-negated (f = 0) add conditions shown in Table 5-4 (never, =, <, <=, NUV, ZNV, SV, OD). When a condition completer is not specified, then the "never" condition is used. The boolean variable "condition_satisfied" in the operation section is set when the values compared fail to satisfy the specified condition.

Operation:

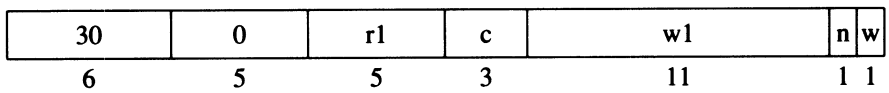
```
GR[r2] ← low_sign_ext(im5,5) + GR[r2];
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

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 r1,target



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 *r1*, 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, 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.

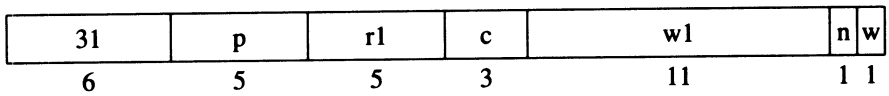
Conditions: The condition, *cond*, is either "<" (bit is 1), or ">=" (bit is 0) from the extract/deposit conditions (Table 5-7). Use of other conditions is an undefined operation.

Operation:

```
GR[r0] ← lshift(GR[r2], CR[11]);
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    IAOQ_Next ← IAOQ_Front + disp + 8;
    }
if (n)
    if (disp < 0)
        PSW[N] ← !cond_satisfied;
    else
        PSW[N] ← cond_satisfied;
```

Exceptions: Taken branch trap

Format: BB,cond,n r1,p,target



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 *r1* 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, 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 either "<" (bit is 1), or ">=" (bit is 0) from the extract/deposit conditions (Table 5-7). Use of other conditions is an undefined operation.

Operation:

```
GR[r0] ← lshift(GR[r1],p);
if (cond_satisfied)
    {
    disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
    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. The exceptions are the instructions that use the extract/deposit conditions. Most computation instructions encode the condition completers in the 3-bit *c*-field and 1-bit *f*-field of the instructions. 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 stores 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 were 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.

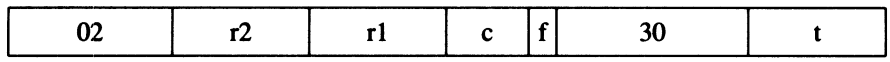
Shift Double, Extract, and Deposit Instructions

The double shift operations allow for a concatenation of two registers followed by a shift of 1 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.

The Deposit instructions take a right-justified field from a source and deposit it into any portion of the target. Deposits either zero the rest of the target or leave it unchanged (merge operation). 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. These instructions provide left shifts, and simple multiplications by powers of two.

Format: ADD,cond r1,r2,t



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.

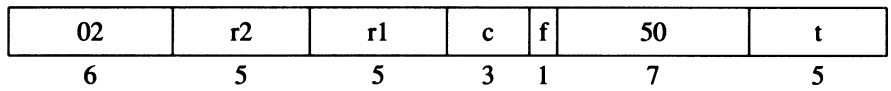
The following instruction is nullified if the values added satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then 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] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

Format: ADDL,cond r1,r2,t



02	r2	r1	c	f	50	t
6	5	5	3	1	7	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 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.

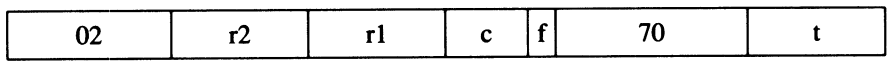
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:
```
GR[t] ← GR[r1] + GR[r2];
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

ADD AND TRAP ON OVERFLOW — ADDO

Format: ADDO,cond r1,r2,t



02	r2	r1	c	f	70	t
6	5	5	3	1	7	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.

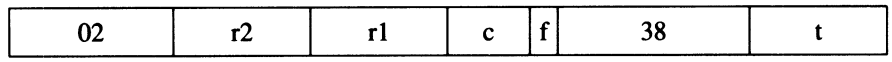
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← GR[r1] + GR[r2];
if (!overflow)
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: Overflow trap

Format: ADDC,cond r1,r2,t



02	r2	r1	c	f	38	t
6	5	5	3	1	7	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.

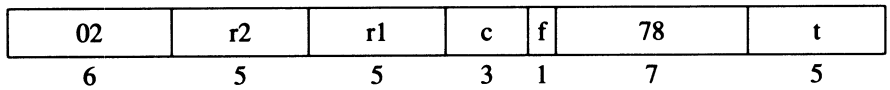
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

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



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.

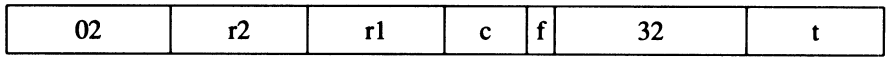
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← GR[r1] + GR[r2] + PSW[C/B]{0};
if (!overflow)
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: Overflow trap

Format: SH1ADD,cond r1,r2,t



02	r2	r1	c	f	32	t
6	5	5	3	1	7	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 a one 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 shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

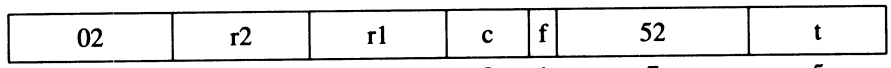
Operation:
```
GR[t] ← lshift(GR[r1], 1) + GR[r2];
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

SHIFT ONE AND ADD LOGICAL

SH1ADDL

Format: SH1ADDL,cond r1,r2,t



02	r2	r1	c	f	52	t
6	5	5	3	1	7	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 a one or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

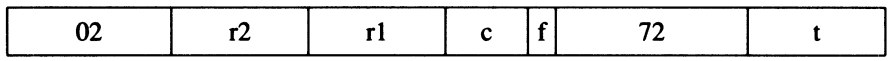
Conditions: The condition is any of the add conditions (Table 5-4). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:
```
GR[t] ← lshift(GR[r1], 1) + GR[r2];
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

SHIFT ONE, ADD AND TRAP ON OVERFLOW — SH1ADDO

Format: SH1ADDO,cond r1,r2,t



02	r2	r1	c	f	72	t
6	5	5	3	1	7	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 a one or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

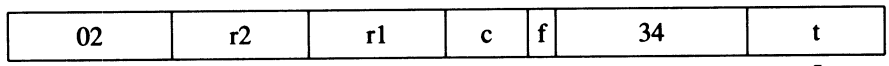
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← lshift(GR[r1], 1) + GR[r2];
if (!overflow)
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: Overflow trap

Format: SH2ADD,cond r1,r2,t



02	r2	r1	c	f	34	t
6	5	5	3	1	7	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 a one or an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

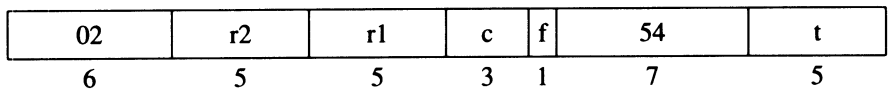
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

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



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 a one or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

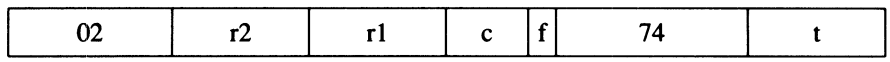
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:
```
GR[t] ← lshift(GR[r2],2) + GR[r2];
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

SHIFT TWO, ADD AND TRAP ON OVERFLOW — SH2ADDO

Format: SH2ADDO,cond r1,r2,t



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 a one or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

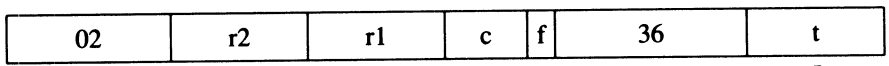
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← lshift(GR[r1],2) + GR[r2];
if (!overflow)
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: Overflow trap

Format: SH3ADD,cond r1,r2,t



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 a one 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 shown in Table 5-4. When a condition completer is not specified, then 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];
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

Format: SH3ADDL,cond r1,r2,t

02	r2	r1	c	f	56	t
6	5	5	3	1	7	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 a one or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

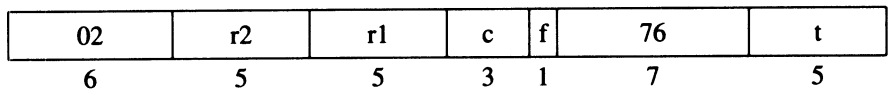
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

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	76	t
6	5	5	3	1	7	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 a one or that an ordinary unsigned overflow occurred during the addition. The conditions take on special interpretations since the shift operation participates in overflow determination.

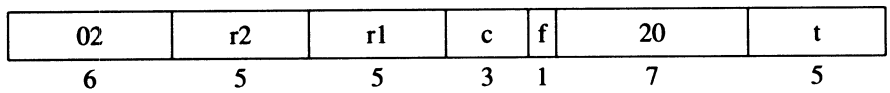
Conditions: The condition is any of the add conditions shown in Table 5-4. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

Operation:

```
res ← lshift(GR[r1],3) + GR[r2];
if (!overflow)
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: Overflow trap

Format: SUB,cond r1,r2,t



02	r2	r1	c	f	20	t
6	5	5	3	1	7	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.

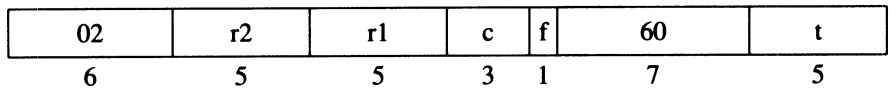
Conditions: The condition is any of the compare or subtract conditions shown in Table 5-3. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

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	60	t
6	5	5	3	1	7	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. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

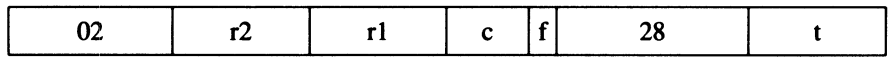
Operation:

```
res ← GR[r1] + (~GR[r2]) + 1;
if (!overflow)
    {
    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	28	t
6	5	5	3	1	7	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. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

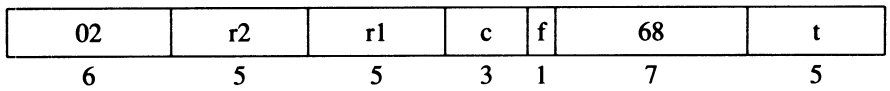
Operation:
```
GR[t] ← GR[r1] + (~GR[r2]) + PSW[C/B]{0};
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

SUBTRACT WITH BORROW AND TRAP ON OVERFLOW

SUBBO

Format: SUBBO,cond r1,r2,t



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. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

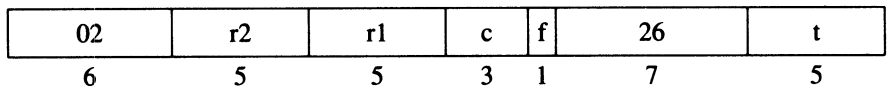
Operation:

```
res ← GR[r1] + (~GR[r2]) + PSW[C/B]{0};
if (!overflow)
    {
    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	26	t
6	5	5	3	1	7	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. When a condition completer is not specified, then 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)
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
```

Exceptions: Conditional trap

SUBTRACT AND TRAP ON CONDITION OR OVERFLOW

Format: SUBTO,cond r1,r2,t

02	r2	r1	c	f	66	t
6	5	5	3	1	7	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 overflow occurs, an overflow trap is taken; if 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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

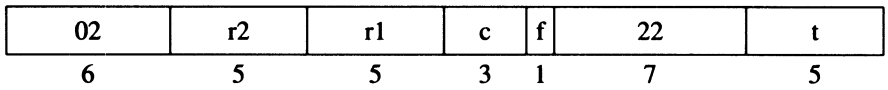
Operation:

```
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



Purpose: To provide the primitive operation for integer division.

Description: This instruction performs a single-bit nonrestoring 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 values divided 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 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 a one 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.

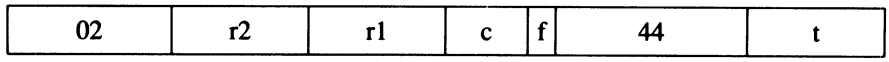
Conditions: The condition is any of the compare or subtract conditions (Table 5-3). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation:

```
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

Format: COMCLR,cond r1,r2,t



02	r2	r1	c	f	44	t
6	5	5	3	1	7	5

Purpose: To compare two registers, clear a register, 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 cleared 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). When a condition completer is not specified, then 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[r1] + (~GR[r2]) + 1;
GR[t] ← 0;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

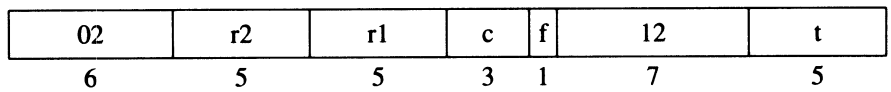
PROGRAMMING NOTE

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. For example,

```
COMCLR,<>     rb,rc,ra
LDO           1(0),ra
```

will set ra to 1 if rb and rc are equal, and to 0 if not.

Format: OR,cond r1,r2,t



02	r2	r1	c	f	12	t
6	5	5	3	1	7	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.

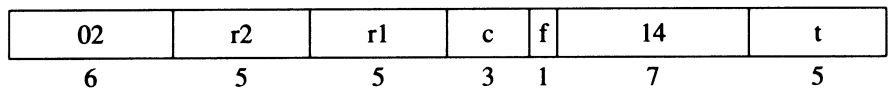
Conditions: The condition is any of the logical conditions (Table 5-5). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ORed satisfy the specified condition.

Operation:
```
GR[t] ← GR[r1] | GR[r2];
if (cond_satisfied) PSW[N] ← 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.

Format: XOR,cond r1,r2,t



Purpose: To do a 32-bit, bitwise exclusive OR.

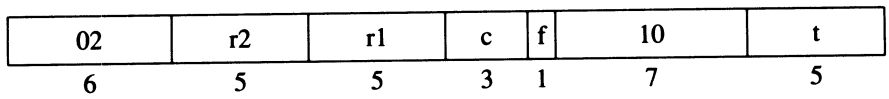
Description: GR *r1* and GR *r2* are XORed and the result is placed in GR *t*. The following instruction is nullified if the values XORed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the logical conditions (Table 5-5). When a condition completer is not specified, then 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

Format: AND,cond r1,r2,t



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.

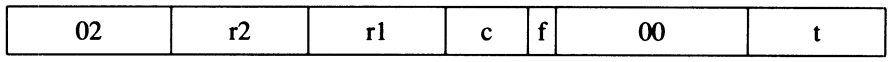
Conditions: The condition is any of the logical conditions (Table 5-5). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ANDed satisfy the specified condition.

Operation:
```
GR[t] ← GR[r1] & GR[r2];
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

AND COMPLEMENT — ANDCM

Format: ANDCM,cond r1,r2,t



02	r2	r1	c	f	00	t
6	5	5	3	1	7	5

Purpose: To do a 32-bit bitwise AND with complement.

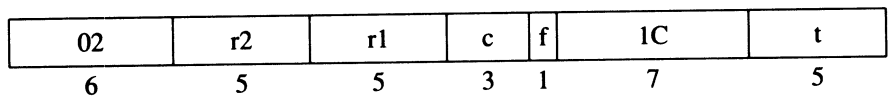
Description: GR *r1* is ANDed with the one's complement of GR *r2* and the result is placed in GR *t*. The following instruction is nullified if the values ANDed satisfy the specified condition, *cond*. The condition is encoded in the *c* and *f* fields of the instruction.

Conditions: The condition is any of the logical conditions (Table 5-5). When a condition completer is not specified, then 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] ← GR[r1] & ~GR[r2];
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

Format: UXOR,cond r1,r2,t



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.

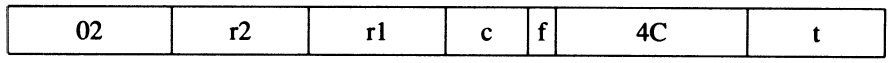
Conditions: The only meaningful conditions are the unit conditions "never", SBZ, SHZ, TR, NBZ and NHZ, since the others all involve carries. When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the 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	4C	t
6	5	5	3	1	7	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). When a condition completer is not specified, then 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:

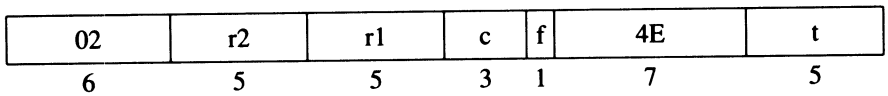
```
GR[t] ← GR[r1] + ~GR[r2];
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

UNIT ADD COMPLEMENT AND TRAP ON CONDITION

UADDCMT

Format: UADDCMT,cond r1,r2,t



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 complemented 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). When a condition completer is not specified, then 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

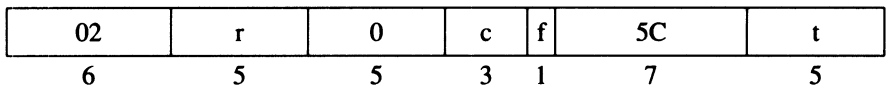
PROGRAMMING NOTE

UNIT ADD COMPLEMENT AND TRAP ON CONDITION can be used to check decimal validity and to pre-bias decimal numbers as follows:

```
NINES          register containing the number X'99999999
R              register containing the number to be checked
T              register to contain number plus bias
UADDCMT,SDC    R,NINES,T
```

DECIMAL CORRECT DCOR

Format: DCOR,cond r,t



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 off in PSW[C/B] has 6 subtracted from it. The result is stored in GR *t*. The following instruction is nullified if GR *r* 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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when GR *r* 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

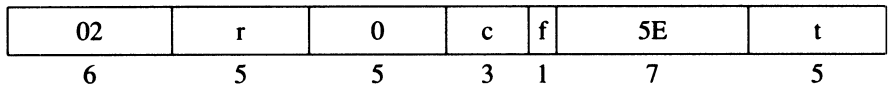
PROGRAMMING NOTE

DECIMAL CORRECT can be used to sum four 32-bit decimal integers as follows:

A,B,C,D	registers containing operands	
X	register to contain result (A+B+C+D)	
NINES	register containing the number X'99999999	
UADDCM	A,NINES,X	; pre-bias first operand
ADD	X,B,X	; add B
IDCORX,X		; correct result, retaining bias
ADD	X,C,uUX	; add C
IDCOR	X,X	; correct result, retaining bias
ADD	X,D,X	; add D
DCOR	X,X	; final correction

INTERMEDIATE DECIMAL CORRECT — IDCOR

Format: IDCOR,cond r,t



02	r	0	c	f	5E	t
6	5	5	3	1	7	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] has 6 added to it. The result is stored in GR *t*. The following instruction is nullified if GR *r* 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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when GR *r* satisfies the specified condition.

Operation:

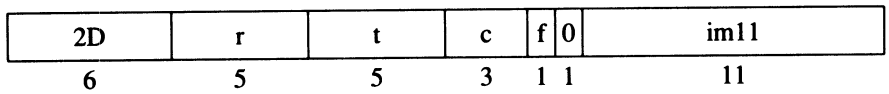
```
GR[t] ← GR[r] + cat(
    0x6*PSW[C/B]{0}, 0x6*PSW[C/B]{1},
    0x6*PSW[C/B]{2}, 0x6*PSW[C/B]{3},
    0x6*PSW[C/B]{4}, 0x6*PSW[C/B]{5},
    0x6*PSW[C/B]{6}, 0x6*PSW[C/B]{7});
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

ADD TO IMMEDIATE

ADDI

Format: ADDI,cond i,r,t



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.

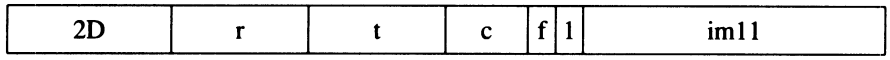
Conditions: The condition is any of the add conditions (Table 5-4). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

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



Purpose: To add an immediate value and 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*; if signed overflow occurs, an overflow trap is taken instead, 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. 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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

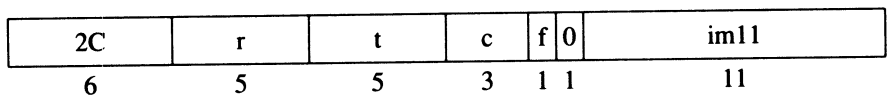
Operation:

```
res ← low_sign_ext(im11,11) + GR[r];
if (!overflow)
    {
    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



Purpose: To add an immediate value and 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). When a condition completer is not specified, then 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:

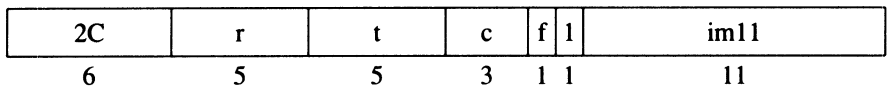
```
res ← low_sign_ext(im11,11) + GR[r];
if (cond_satisfied)
    conditional_trap;
else
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
```

Exceptions: Conditional trap

ADD TO IMMEDIATE AND TRAP ON CONDITION OR OVERFLOW

ADDITO

Format: ADDITO,cond i,r,t



Purpose: To add an immediate value and 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). When a condition completer is not specified, then 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:

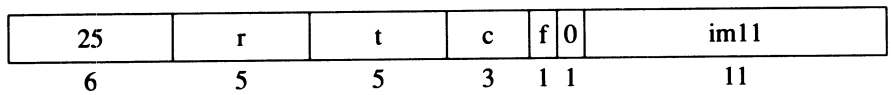
```
res ← low_sign_ext(im11,11) + GR[r];
if (overflow)
    overflow_trap;
else if (cond_satisfied)
    conditional_trap;
else
    {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    }
```

Exceptions: Overflow trap
Conditional trap

SUBTRACT FROM IMMEDIATE

SUBI

Format: SUBI,cond i,r,t



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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

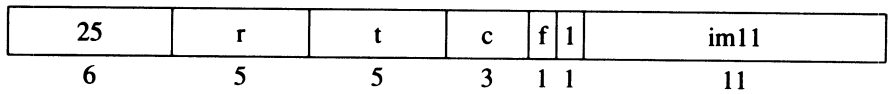
Operation:
```
GR[t] ← low_sign_ext(im11,11) + (~GR[r]) + 1;
PSW[C/B] ← carry_borrows;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

SUBTRACT FROM IMMEDIATE AND TRAP ON OVERFLOW

SUBIO

Format: SUBIO,cond i,r,t



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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

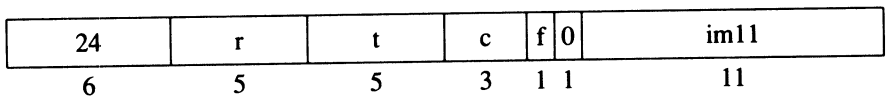
Operation:

```
res ← low_sign_ext(im11,11) + (~GR[r]) + 1;
if (!overflow)
    {
    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



Purpose: To compare an immediate value with the contents of a register, clear a register, and conditionally nullify the following instruction.

Description: The sign-extended immediate and GR *r* are compared and GR *t* is cleared 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.

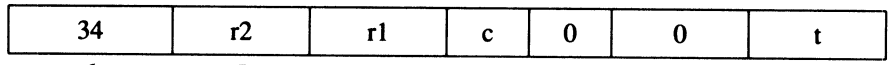
Conditions: The condition is any of the compare or subtract conditions (Table 5-3). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values compared satisfy the specified condition.

Operation:
```
GR[0] ← low_sign_ext(im11,11) + (~GR[r]) + 1;
GR[t] ← 0;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

VARIABLE SHIFT DOUBLE

Format: VSHD,cond r1,r2,t



34	r2	r1	c	0	0	t
6	5	5	3	3	5	5

Purpose: To shift a concatenated pair of registers by a variable amount and conditionally nullify the following instructions.

Description: The rightmost 31 bits of GR *r1* and GR *r2* are concatenated 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* and *f* fields of the instruction.

Conditions: The condition is any of the extract/deposit conditions (Table 5-7). When a condition completer is not specified, then 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]){31..62};
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

PROGRAMMING NOTE

If *r1* and *r2* name the same register, its contents are rotated and placed in GR *t*. See SHIFT DOUBLE for an example.

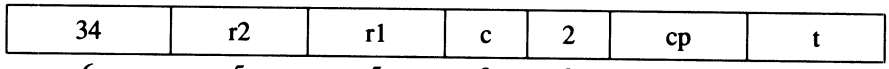
A logical right shift of GR *r* by a variable amount contained in GR *p* leaving the result in GR *t* may be done by the following sequence:

```
MTCTL   p,cr11
VSHD    0,r,t
```

An arithmetic right shift of GR *r* by a variable amount contained in GR *p* leaving the result in GR *t* (and using GR 1 as a temporary) may be done by the following sequence:

```
EXTRS   r,0,1,r1
MTCTL   p,cr11
VSHD    r1,r,t
```

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 concatenated pair of registers by a fixed amount and conditionally nullify the following instruction.

Description: The rightmost 31 bits of GR *r1* and GR *r2* are concatenated 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* and *f* fields 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). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation statisfies the condition.

Operation:
```
GR[t] ← rshift(cat(GR[r1]{1..31},GR[r2]),p){31..62};
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

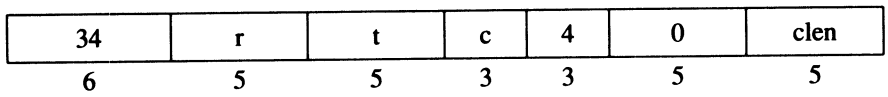
PROGRAMMING NOTE

If *r1* and *r2* name the same register, its contents are rotated and stored in GR *t*. For example the following rotates the contents of ra right by 8 bits:

```
SHD     ra,ra,8,ra
```

VARIABLE EXTRACT UNSIGNED — VEXTRU

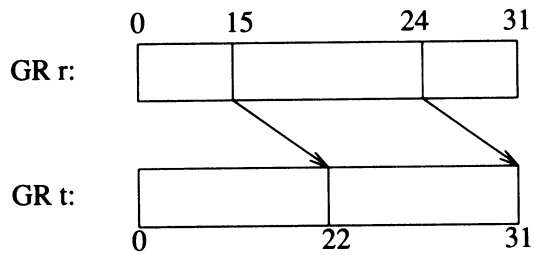
Format: VEXTRU,cond r,len,t



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 (CR11) 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* and *f* fields 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). When a condition completer is not specified, then 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.

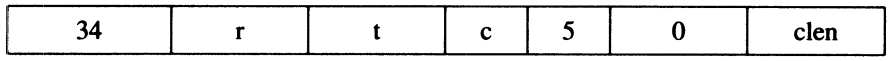
Operation:

```
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 (CR11) 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* and *f* fields of the instruction.

The length *len* in the assembly language format is represented in the machine instruction by *clen*, whose value is 32-*len*.

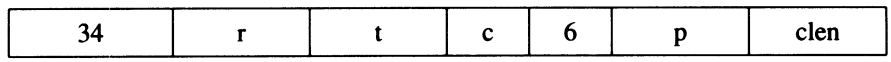
Conditions: The condition is any of the extract/deposit conditions (Table 5-7). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

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

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* and *f* fields 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). When a condition completer is not specified, then 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.

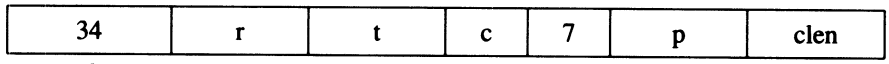
Operation:

```
if ((1 + p) >= len)
    {
    tmp ← lshift(zero_ext_64(GR[r],32),(1+p)){0..31};
    GR[t] ← zero_ext(tmp{32-len..31},len);
    }
else
    undefined;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

EXTRACT SIGNED — EXTRS

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* and *f* fields 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). When a condition completer is not specified, then 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 ((1 + p) >= len)
    {
    tmp ← lshift(sign_ext_64(GR[r],32),(1+p)){0..31};
    GR[t] ← sign_ext(tmp{32-len..31},len);
    }
else
    undefined;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

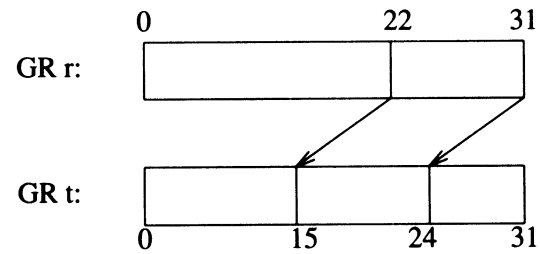
Format: VDEP,cond r,len,t

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 (CR11) 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* and *f* fields 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). When a condition completer is not specified, then 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

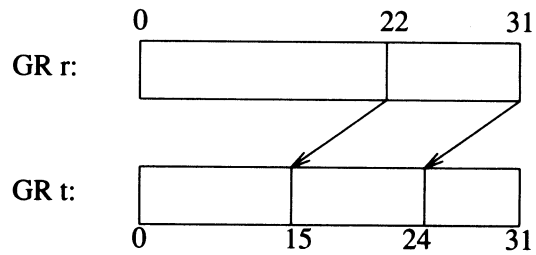
Format: DEP,cond r,p,len,t

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* and *f* fields 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*.

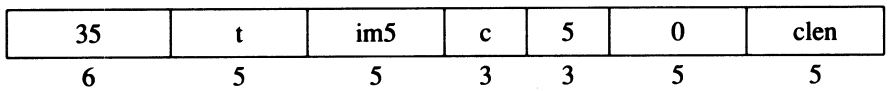
Conditions: The condition is any of the extract/deposit conditions (Table 5-7). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:
```
if (p >= (len-1))
   GR[t]{p-len+1..p} ← GR[r]{32-len..31};
else
   undefined;
if (cond_satisfied) PSW[N] ← 1;
```

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* and *f* fields 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). When a condition completer is not specified, then 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:

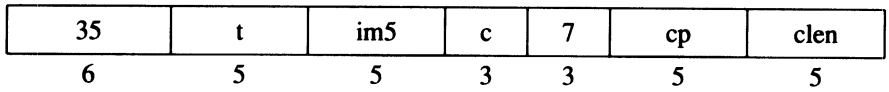
```
tpos ← CR[11];
ival ← low_sign_ext(im5,5);
if (tpos < (len-1))
   GR[t] ← cat(ival{31-tpos..31},GR[t]{tpos+1..31});
else
   GR[t] ← cat(GR[t]{0..tpos-len},ival{32-len..31},GR[t]{tpos+1..31});
if (cond_satisfied) PSW[N] ← 1;
```

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* and *f* fields 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). When a condition completer is not specified, then 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:

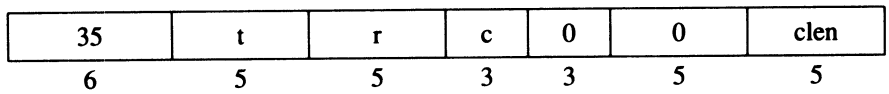
```
ival ← low_sign_ext(im5,5);
if (p >= (len-1))
   GR[t] ← cat(GR[t]{0..p-len},ival{32-len..31},GR[t]{p+1..31});
else
   undefined;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

ZERO AND VARIABLE DEPOSIT

ZVDEP

Format: ZVDEP,cond r,len,t



Purpose: To zero a register, deposit a value into it at a variable position, and conditionally nullify the following instruction.

Description: GR *t* is zeroed 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* and *f* fields 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). When a condition completer is not specified, then 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

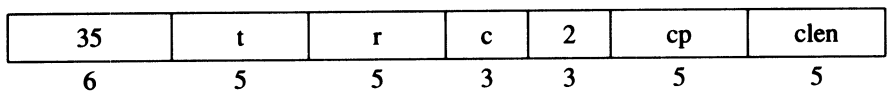
PROGRAMMING NOTE

A left shift of GR *r* by a variable amount given by GR *p* leaving the result in GR *t* (and using GR 1 as a temporary) may be done by

```
SUBI    31,p,1
MTCTL   1,11
ZVDEP   r,32,t
```

Note that this provides no indication of a possible overflow.

Format: ZDEP,cond r,p,len,t



35	t	r	c	2	cp	clen
6	5	5	3	3	5	5

Purpose: To zero a register and deposit a value into it at a constant position.

Description: GR *t* is zeroed 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* and *f* fields 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*.

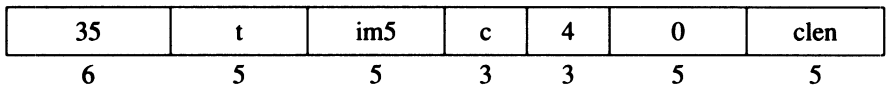
Conditions: The condition is any of the extract/deposit conditions (Table 5-7). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:
```
if (p >= (len-1))
    GR[t] ← cat(0{0..p-len},GR[r]{32-len..31},0{p+1..31});
else
    undefined;
if (cond_satisfied) PSW[N] ← 1;
```

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 zero a register and deposit an immediate value into it at a variable position.

Description: GR *t* is zeroed 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* and *f* fields 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*.

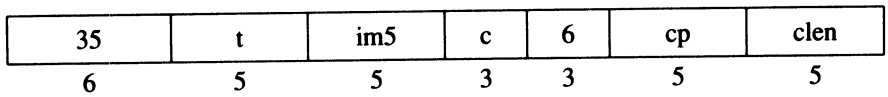
Conditions: The condition is any of the extract/deposit conditions (Table 5-7). When a condition completer is not specified, then the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

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

Format: ZDEPI,cond i,p,len,t



35	t	im5	c	6	cp	clen
6	5	5	3	3	5	5

Purpose: To zero a register and deposit an immediate value into it at a constant position.

Description: GR *t* is zeroed 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* and *f* fields 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). When a condition completer is not specified, then 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,len);
if (p >= (len-1))
    GR[t] ← cat(0{0..p-len},ival{32-len..31},0{p+1..31});
else
    undefined;
if (cond_satisfied) PSW[N] ← 1;
```

Exceptions: None

System Control Instructions

The system control instructions provide special register moves, system mask control, return from interruption, hash address computation, 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 5-11.

Table 5-11. System Control Completers.

cmplt	Description	m
	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 Instruction TLB Address and Insert Instruction 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 level for which access is to be validated, and do not perform address modification.

The space identifier is computed in one of two different ways and is shown in Figure 5-11. The calculation of the offset portion of the address is shown in Figure 5-12.

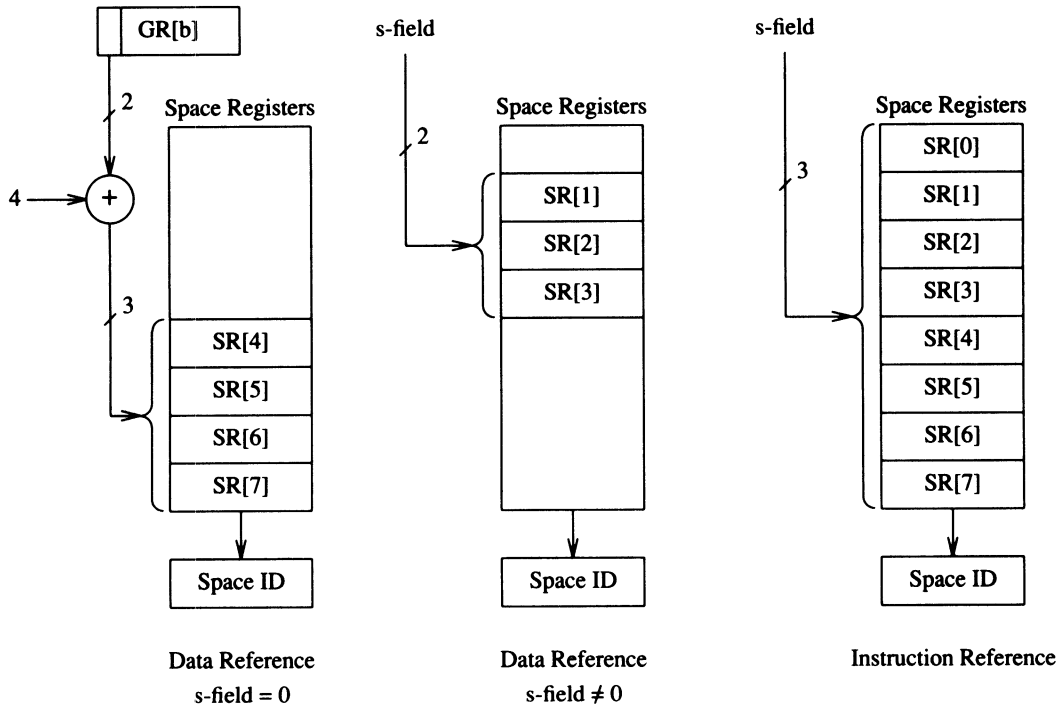


Figure 5-11. Space Identifier Selection.

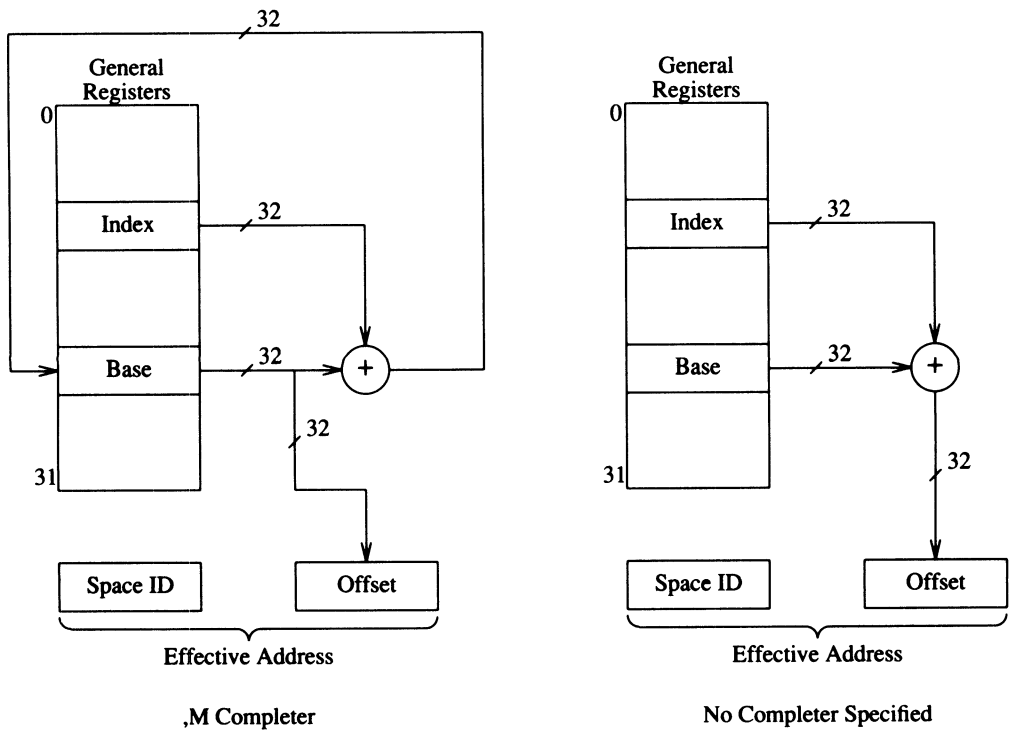
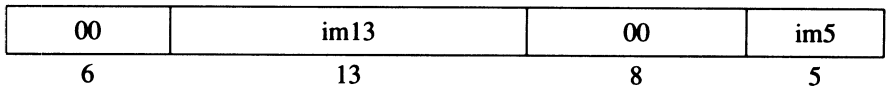


Figure 5-12. Effective Address Computation For System Operations.

Format: BREAK im5,im13



Purpose: To cause a Break instruction trap for debugging purposes.

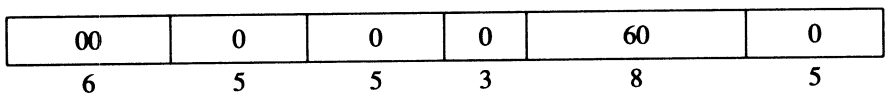
Description: A Break instruction trap occurs when this instruction is executed.

Operation: Break instruction trap

Exceptions: None

Notes: *im5* and *im13* can be used as parameters to the "BREAK" processing code.

Format: RFI



Purpose: To restore processor state and restart execution of an interrupted instruction stream.

Description: The PSW register contents are restored from the IPSW register and not modified by this instruction. The IA Space and IA Offset queues are restored from the Interrupt IA Space and Offset Queues. Execution continues at the locations loaded into the IA queues.

Execution of an RFI with the IPSW Q-bit (Interruption state collection enable) 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 with PSW Q, I, L, H, or R bits set is undefined. This is the only instruction that can turn the PSW Q-bit on.

Operation:

```
PSW   ← IPSW;
IAOQ_Back   ← IIAOQ_Back;          /* CR[18] */
IASQ_Back   ← IIASQ_Back;          /* CR[17] */
IAOQ_Front  ← IIAOQ_Front;         /* CR[18] */
IASQ_Front  ← IIASQ_Front;         /* CR[17] */
```

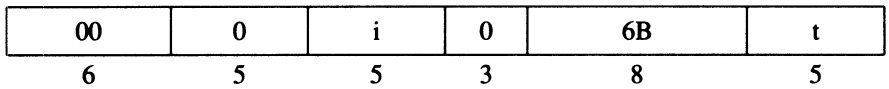
Exceptions: Privileged operation trap

Restrictions: This instruction can only be executed by code running 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.

Level Zero: On level zero systems, this instruction executes as usual except that the effect of updating the nonexistent IA space queue and the unimplemented bits in the PSW is undefined.

Format: SSM i,t



00	0	i	0	6B	t
6	5	5	3	8	5

Purpose: To selectively set bits in the system mask.

Description: The current value of PSW bits 27 through 31 is saved in GR *t* and then the immediate value *i* is ORed with PSW bits 27 through 31. PSW bit 28 (Q-bit) cannot be set by this instruction and doing so is an undefined operation.

Operation:

```
if ((PSW[Q] == 0) && (i{1}))
   undefined;
else
   {
   GR[t]    ← cat(0{0..26},PSW{27..31});
   PSW[R]   ← PSW[R] | i{0};
   PSW[P]   ← PSW[P] | i{2};
   PSW[D]   ← PSW[D] | i{3};
   PSW[I]   ← PSW[I] | i{4};
   }
```

Exceptions: Privileged operation trap

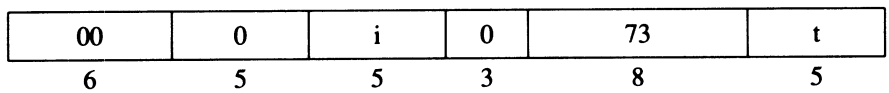
Restrictions: This instruction can be executed only by code running at the most privileged level.

Notes: While it is possible to execute this instruction with the bit in *i* in the position corresponding to PSW[Q] having the value 1, the effect of doing so is undefined if PSW[Q] is not already 1.

RESET SYSTEM MASK

RSM

Format: RSM i,t



00	0	i	0	73	t
6	5	5	3	8	5

Purpose: To selectively reset bits in the system mask.

Description: The current values of PSW bits 27 through 31 are saved in GR *t* and then the complement of the immediate field *i* is ANDed with PSW bits 27 through 31.

Operation:

```
GR[t]   ← cat(0{0..26},PSW{27..31});
PSW[R]  ← PSW[R] & (~i{0});
PSW[Q]  ← PSW[Q] & (~i{1});
PSW[P]  ← PSW[P] & (~i{2});
PSW[D]  ← PSW[D] & (~i{3});
PSW[I]  ← PSW[I] & (~i{4});
```

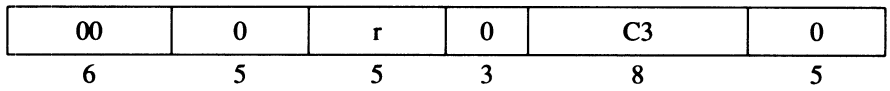
Exceptions: Privileged operation trap

Restrictions: This instruction can be executed only by code running at the most privileged level.

Notes: The state of the IPRs, IIA queues, and the IPSW is undefined when system control instructions (RSM or MTSM) are used to turn the Q-bit off.

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 five rightmost bits of GR *r* replace PSW bits 27 through 31. The PSW Q-bit may not be changed to a 1 by this instruction; that operation is undefined.

Operation:

```
if ((PSW[Q] == 0) && (GR[r]{28} == 1))
   undefined;
else
   {
   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 can only be executed by code running at the most privileged level.

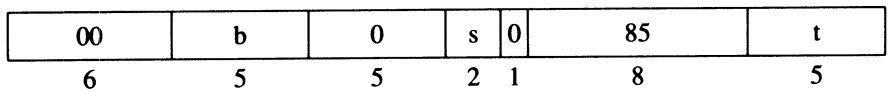
Notes: While it is possible to execute this instruction with the bit in field *r* in the position corresponding to PSW[Q] having the value 1, the effect of doing so is undefined if PSW[Q] is not already 1.

The state of the IPRs, IIA queues, and IPSW is undefined when system control instructions (RSM or MTSM) are used to turn the Q-bit off.

LOAD SPACE IDENTIFIER

LDSID

Format: LDSID (s,b),t



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:

```
if (s == 0)
   GR[t] ← SR[GR[b]{0..1} + 4];
else
   GR[t] ← SR[s];
```

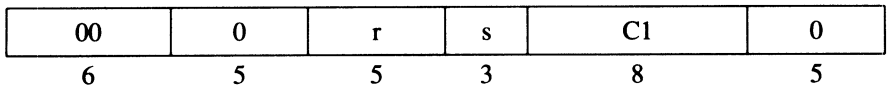
Exceptions: None

Level Zero: On level zero systems, the value 0 is written into the specified GR.

Notes: On level one 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 loaded into the specified SR.

MOVE TO SPACE REGISTER — MTSP

Format: MTSP r,sr



00	0	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:
sr ← assemble_3(s);
SR[sr] ← GR[r];

Exceptions: Privileged register trap

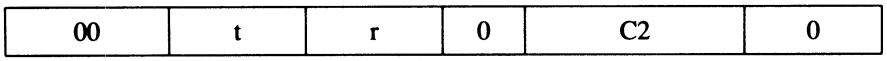
Restrictions: SRs 5, 6 and 7 may be changed only by software running at the most privileged level.

Level Zero: On level zero systems, this instruction executes as a null instruction.

Notes: On level one systems, the leftmost 16 bits are nonexistent bits.

MOVE TO CONTROL REGISTER — MTCTL

Format: MTCTL r,t



00	t	r	0	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:

```
switch(t)
    {
    case 0:  case 14: case 15: case 16: case 22: case 24: case 25:
    case 26: case 27: case 28: case 29: case 30: case 31:
            CR[t] ← GR[r];
            break;
    case 1:  case 2: case 3: case 4: case 5: case 6: case 7: case 19:
    case 20: case 21:
            undefined;
            break;
    case 17: case 18:
            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]{24..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];
    }
```

Exceptions: Privileged register trap

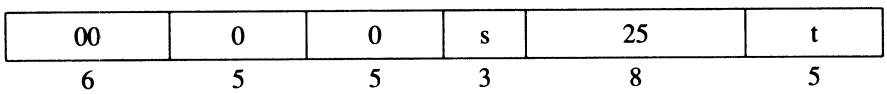
Restrictions: System control registers other than the shift amount register (CR 11) may be written only by code running at the most privileged level. CR 11 can be written by code running at any privilege level.

Notes: The MTSAR pseudo operation generates an MTCTL r,cr11 to copy a general register to the shift amount register (CR 11).

Level Zero: On level zero systems, 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	0	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:
sr ← assemble_3(s);
GR[t] ← SR[sr];

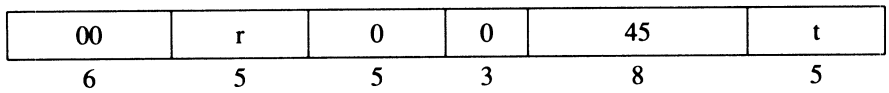
Exceptions: None

Level One: On level zero systems, the value 0 is written into the specified general register.

Notes: On level one 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 loaded into the specified space register.

MOVE FROM CONTROL REGISTER — MFCTL

Format: MFCTL r,t



00	r	0	0	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:

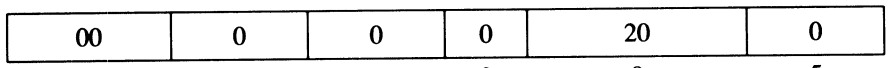
```
if ((r == 0) || (r >= 8))
    GR[t] ← CR[r];
else
    undefined;                                  /* CRs 1 – 7 */
```

Exceptions: Privileged register trap

Restrictions: System control registers other than the shift amount register (CR 11) and the interval timer (CR 16) may be read only by code running at the most privileged level. CRs 11 and 16 can be read by code running at any privilege level.

Level Zero: On level zero systems, if the source control register is CR 8, 9, 12, 13, 17, or 20, a 0 is written into the specified general register.

Format: SYNC



00	0	0	0	20	0
6	5	5	3	8	5

Purpose: To synchronize the caches with memory and instruction execution by causing the completion of all pending I-cache and D-cache operations.

Description: Instruction execution is suspended until the completion of all I-cache and D-cache operations resulting from preceding FLUSH DATA CACHE, FLUSH DATA CACHE ENTRY, FLUSH INSTRUCTION CACHE, FLUSH INSTRUCTION CACHE ENTRY, and PURGE DATA CACHE instructions.

Operation: complete all D-cache operations;
complete all I-cache operations

Exceptions: None

Notes: In systems that do not have a cache or in which all cache operations are performed synchronously, this instruction is executed as a null instruction.

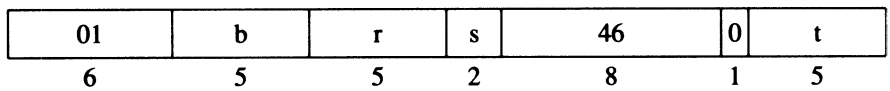
PROGRAMMING NOTE

Execution of this instruction guarantees that any reference to data following its execution will await the completion of previous cache operations. Since the architecture permits instruction prefetching, the following is the minimum spacing that is guaranteed to work for "self-modifying code":

```
        LDW     newinstr(0,r0),r1
        STW     r1,instr(0,r0)
        FDC     instr(0,r0)
        SYNC
        FIC     instr(0,r0)
        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.

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 set, the protection ID is also checked. The instruction uses data address translation regardless of the state of the PSW D-bit.

Operation:

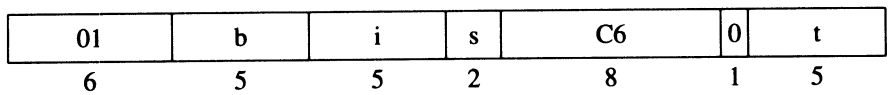
```
offset ← GR[b];
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (read_access_allowed(space,offset,GR[r]))
  GR[t] ← 1;
else
  GR[t] ← 0;
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Level Zero: On level zero systems, this instruction always sets the target general register to one.

PROBE READ ACCESS IMMEDIATE — PROBERI

Format: PROBERI (s,b),i,t



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 set, it also checks the protection ID. This instruction uses data address translation regardless of the state of the PSW D-bit.

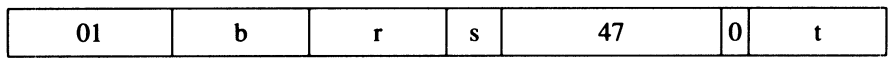
Operation:

```
offset ← GR[b];
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (read_access_allowed(space,offset,i))
  GR[t] ← 1;
else
  GR[t] ← 0;
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Level Zero: On level zero systems, this instruction always sets the target general register to one.

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 set, the protection ID is also checked. This instruction uses data address translation regardless of the state of the PSW D-bit.

Operation:

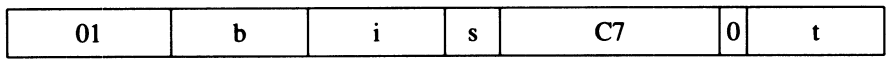
```
offset ← GR[b];
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (write_access_allowed(space,offset,GR[x]))
  GR[t] ← 1;
else
  GR[t] ← 0;
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Level Zero: On level zero systems, this instruction always sets the target general register to one.

PROBE WRITE ACCESS IMMEDIATE — PROBEWI

Format: PROBEWI (s,b),i,t



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, it also checks the protection ID. This instruction uses data address translation regardless of the state of the PSW D-bit.

Operation:

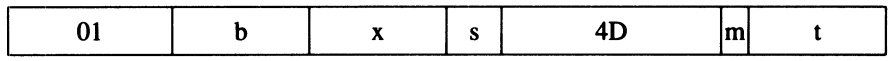
```
offset ← GR[b];
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (write_access_allowed(space,offset,i))
  GR[t] ← 1;
else
  GR[t] ← 0;
```

Exceptions: Non-access data TLB miss fault/non-access data page fault

Level Zero: On level zero systems, this instruction always sets the target general register to one.

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 physical 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 for the assembly language completer mnemonics.) GR *t* receives the physical address corresponding to the given virtual address. If the page is not present, GR *t* is set to 0.

In systems with separate data and instruction TLBs, the physical address is obtained from the data TLB. This instruction uses data address translation regardless of the state of the PSW D-bit.

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[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 (hardware_TLB_fault_handling)
   {
   if (search_DTLB(space,offset,&entry))
      GR[t] ← physical_address(space,offset);
   else
      GR[t] ← 0;
   }
else if (search_DTLB(space,offset,&entry))
   GR[t] ← physical_address(space,offset);
else
   non-access_data_TLB_miss_fault();
```

Exceptions: Privileged operation trap
Illegal instruction trap
Non-access data TLB miss fault

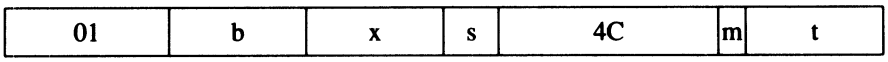
Restrictions: The result of LPA is ambiguous for an address which maps to physical address 0.

This instruction may be executed only at the most privileged level. The result is undefined if address modification is specified and $b = t$.

Level Zero: On level zero systems, this instruction causes an Illegal instruction trap.

LOAD HASH ADDRESS LHA

Format: LHA,cmplt x(s,b),t



01	b	x	s	4C	m	t
6	5	5	2	8	1	5

Purpose: To determine the physical address of the hash table entry corresponding to a virtual address.

Description: An effective address is calculated, and GR *t* is set to the physical address of the hash table entry corresponding to the given virtual address. 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 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[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*/
   }
GR[t] ← hash_table_entry_address(space,offset);
```

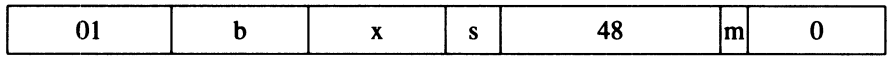
Exceptions: Privileged operation trap
Illegal instruction trap

Restrictions: This instruction may be executed only at the most privileged level. The result is undefined if address modification is specified and $b = t$.

Notes: This instruction is implemented only on systems that perform TLB fault handling in hardware. On systems with TLB handling done in software, this instruction produces an Illegal instruction trap.

Level Zero: On level zero systems, this instruction causes an Illegal instruction trap.

Format: PDTLB,cmplt x(s,b)



01	b	x	s	48	m	0
6	5	5	2	8	1	5

Purpose: To invalidate a data TLB entry.

Description: The data TLB entry (if any) for the page specified by the effective address generated by the instruction is invalidated. 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 for the assembly language completer mnemonics.)

In a multiprocessor system sharing a single PDIR, this instruction must broadcast to all data TLBs. The other processors must complete the purge before the issuing processor continues.

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[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_DTLB(space, offset{0..20},&entry))
   DTLB[entry].ENTRY_VALID ← false;
```

Exceptions: Privileged operation trap

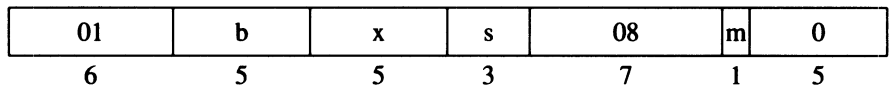
Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries in single-TLB systems.

Level Zero: On systems that do not have TLBs, this instruction executes as a null instruction.

PURGE INSTRUCTION TLB — PITLB

Format: PITLB,cmplt x(sr,b)



Purpose: To invalidate an instruction TLB entry.

Description: The instruction TLB entry (if any) for the page specified by the effective address generated by the instruction is 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, also specifies base register modification. (See Table 5-11 for the assembly language completer mnemonics.)

In a multiprocessor system sharing a single PDIR, this instruction must broadcast to all instruction TLBs. The other processors must complete the purge before the issuing processor continues.

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*/
    }
if (search_ITLB(space,offset{0..21},&entry))
    ITLB[entry].ENTRY_VALID ← false;
```

Exceptions: Privileged operation trap

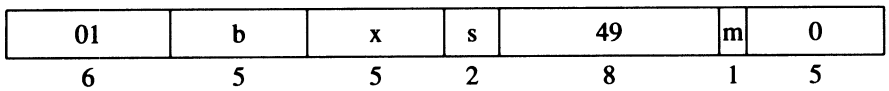
Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries in single-TLB systems.

Level Zero: On systems that do not have TLBs, this instruction executes as a null instruction.

PURGE DATA TLB ENTRY — PDTLBE

Format: PDTLBE,cmplt x(s,b)



Purpose: To invalidate a data TLB entry without matching the address portion.

Description: A data TLB entry specified by a portion (model-dependent) of the effective address generated by the instruction is 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. (See Table 5-11 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.

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[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*/
   }
purge_data_TLB_entry(space,offset);
```

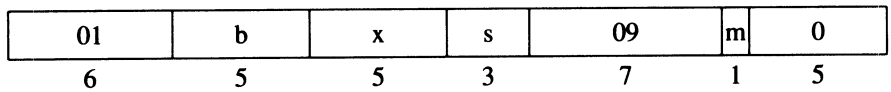
Exceptions: Privileged operation trap

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

Notes: This instruction may be used to purge both instruction entries and data entries in single-TLB systems.

Level Zero: On level zero systems, this instruction executes as a null instruction.

Format: PITLBE,cmplt x(sr,b)



Purpose: To invalidate an instruction TLB entry without matching the address portion.

Description: An instruction TLB entry specified by a portion (model dependent) of the effective address generated by the instruction is 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. (See Table 5-11 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.

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*/
    }
purge_instruction_TLB_entry(space,offset);
```

Exceptions: Privileged operation trap

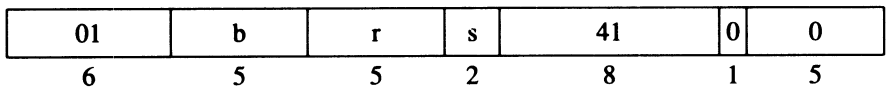
Restrictions: This instruction may be executed only by code running at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries in single-TLB systems.

Level Zero: On level zero systems, this instruction executes as a null instruction.

INSERT DATA TLB ADDRESS — IDTLBA

Format: IDTLBA r,(s,b)



Purpose: To begin adding an entry to the data TLB (INSERT DATA TLB PROTECTION completes the process).

Description: An entry is found in the data TLB to place a new translation and this entry is invalidated. If the data TLB already contains an entry with this address, the entry is reused. The base register, *b*, forms the effective 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:

```
offset ← GR[b];
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
if (!search_DTLB(space,offset,&entry))
    alloc_DTLB(space,offset,&entry);
DTLB[entry].ENTRY_VALID ← false;
DTLB[entry].VIRTUAL_ADDR ← cat(space,offset{0..20});
DTLB[entry].PHY_PAGE_NO ← GR[r]{7..27};
```

Exceptions: Privileged operation trap

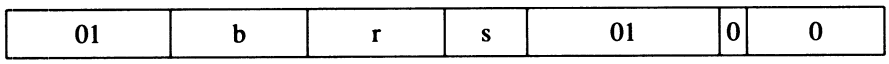
Restrictions: This instruction may be executed only by code running at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries in single-TLB systems.

Level Zero: On level zero zero systems, this instruction executes as a null instruction.

INSERT INSTRUCTION TLB ADDRESS — IITLBA

Format: IITLBA r,(sr,b)



Purpose: To begin adding an entry to the instruction TLB (INSERT INSTRUCTION TLB PROTECTION completes the process).

Description: An entry is found in the instruction TLB to place a new translation and this entry is invalidated. If the instruction TLB already contains an entry with this address, the entry is reused. The base register, *b*, forms the effective 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:

```
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..20});
ITLB[entry].PHY_PAGE_NO ← GR[r]{7..27};
```

Exceptions: Privileged operation trap

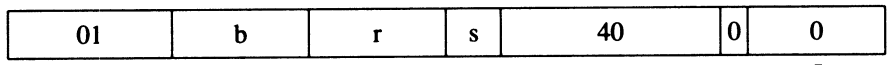
Restrictions: This instruction may be executed only by code running at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries in single-TLB systems.

Level Zero: On level zero systems, this instruction executes as a null instruction.

INSERT DATA TLB PROTECTION — IDTLBP

Format: IDTLBP r,(s,b)



Purpose: To finish adding an entry to the data TLB (INSERT DATA TLB ADDRESS begins the process).

Description: The address computed from the instruction is checked for a match against entries in the data 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 (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
offset ← GR[b];
if (search_DTLB(space,offset,&entry))
    {
    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]{16..30};
    DTLB[entry].ENTRY_VALID ← true;
    }
```

Exceptions: Privileged operation trap

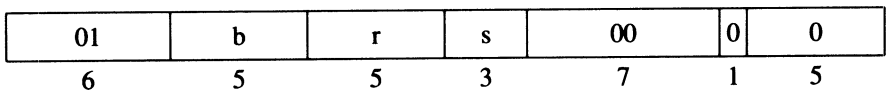
Restrictions: This instruction may be executed only by code running at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries in single-TLB systems.

Level Zero: On level zero systems, this instruction executes as a null instruction.

INSERT INSTRUCTION TLB PROTECTION — IITLBP

Format: IITLBP r,(sr,b)



Purpose: To finish adding an entry to the instruction TLB (INSERT INSTRUCTION TLB ADDRESS begins the process).

Description: The address computed from the instruction is checked for a match against entries in the instruction 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:

```
space ← SR[assemble_3(s)];
offset ← GR[b];
if (search_ITLB(space,offset,&entry))
    {
    ITLB[entry].ACCESS_RIGHTS ← GR[r]{5..11};
    ITLB[entry].ACCESS_ID ← GR[r]{16..30};
    ITLB[entry].ENTRY_VALID ← true;
    }
```

Exceptions: Privileged operation trap

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

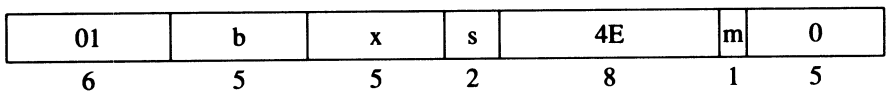
Notes: This instruction may be used to insert both instruction entries and data entries in single-TLB systems. The T, D, and B bits should be set to the appropriate bits of GR *r* in that case.

Level Zero: On level zero systems, this instruction executes as a null instruction.

PURGE DATA CACHE

PDC

Format: PDC,cmplt x(s,b)



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. 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 for the assembly language completer mnemonics.) No write-back is performed.

The process must have write access to the data. The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

In a multiprocessor system that shares a single PDIR, this instruction must broadcast to all data caches.

Operation:

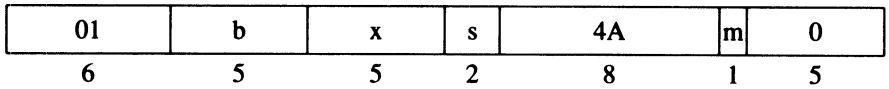
```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
switch (cmplt)
    {
    case ',M': offset ← GR[b];                          /*m=1*/
               GR[b] ← GR[b] + GR[x];
               break;
    default:   offset ← GR[b] + GR[x];                  /*m=0*/
    }
purge_data_cache(space,offset);
```

Exceptions:
Data memory protection trap
Non-access data TLB miss fault
Data memory break trap

Notes: For systems that do not have a cache, this instruction is executed as a null instruction.

FLUSH DATA CACHE FDC

Format: FDC,cmplt x(s,b)



01	b	x	s	4A	m	0
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 for the assembly language completer mnemonics.) The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

A dirty line is a line in which any byte has been written to since it was read from memory.

In a multiprocessor system that shares a single PDIR, this instruction must broadcast to all data caches.

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[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_data_cache(space,offset);
```

Exceptions: Non-access data TLB miss fault

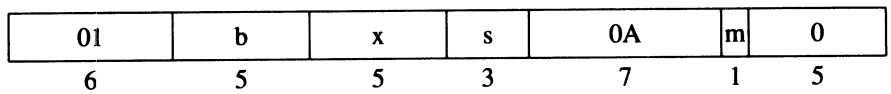
Notes: In single-cache systems, this instruction may be used to flush both data and instruction lines from the cache.

For systems that do not have a cache, this instruction is executed as a null instruction.

FLUSH INSTRUCTION CACHE

FIC

Format: FIC,cmplt x(sr,b)



01	b	x	s	0A	m	0
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 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.

For systems with single set instruction TLBs, the data TLB must be used when translation is needed. In this case, a TLB fault is reported using a non-access data TLB miss fault.

In a multiprocessor system that shares a single PDIR, this instruction must broadcast to all instruction 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);
```

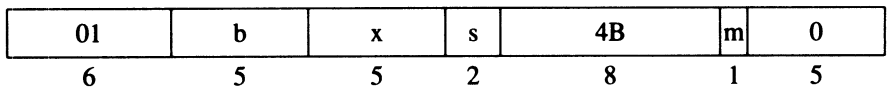
Exceptions:
Non-access instruction TLB miss fault
Non-access data TLB miss fault

Notes: In single-cache systems, this instruction may be used to flush both instruction and data lines in the cache, including writing them back to main memory, if it is dirty.

For systems that do not have a cache, this instruction is executed as a null instruction.

FLUSH DATA CACHE ENTRY — FDCE

Format: FDCE,cmplt x(s,b)



Purpose: To cause a data cache line to be invalidated in the D-cache without requiring knowledge of its virtual address.

Description: At least one cache line (if there are any) specified by a portion (model-dependent) of the effective address is written back to main memory, if and only if it is dirty, and invalidated in 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. No translation is performed. (See Table 5-11 for the assembly language completer mnemonics.)

This is an implementation-dependent instruction that can be used to flush the entire cache.

Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[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_data_cache_entry(space,offset);
```

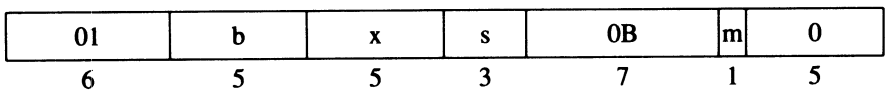
Exceptions: None

Notes: In single-cache systems, this instruction may be used to flush both data and instruction lines from the cache.

For systems that do not have a cache, this instruction is executed as a null instruction.

FLUSH INSTRUCTION CACHE ENTRY — FICE

Format: FICE,cmplt x(sr,b)



01	b	x	s	0B	m	0
6	5	5	3	7	1	5

Purpose: To cause an instruction cache entry to be invalidated in the I-cache without requiring knowledge of its virtual address.

Description: At least one cache line (if there are any) specified by a portion (model-dependent) of the effective address 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 for the assembly language completer mnemonics.) The space register, *sr*, is encoded in the *s* field of the instruction. No translation is performed.

This is an implementation-dependent instruction that can be used to flush the entire cache.

Operation:

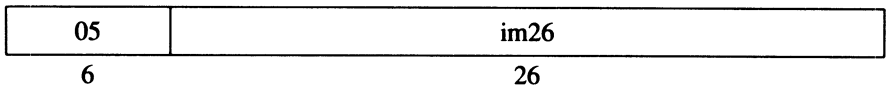
```
switch (cmplt)
   {
   case ',M': offset ← GR[b];                          /*m=1*/
              GR[b] ← GR[b] + GR[x];
              break;
   default:   offset ← GR[b] + GR[x];                  /*m=0*/
   }
space ← SR[assemble_3(s)];
flush_instruction_cache_entry(space,offset);
```

Exceptions: None

Notes: In single-cache systems, this instruction flushes entries in the cache, including writing them back to main memory, if it is dirty.

For systems which do not have a cache, this instruction is executed as a null instruction.

Format: DIAG i



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: Implementation-dependent.

Exceptions: Implementation-dependent.

Assist Instructions

The HP Precision Architecture 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. The floating-point coprocessor is the only assist that is architecturally specified.

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 all HP Precision Architecture 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 HP Precision Architecture instruction set contains all supported instructions, including those for floating-point operations. Particular models may 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 are 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.

Architected Assist Units

Some assist processors are architecturally 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 architected 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.

A 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.

Architected special function units will conform to the requirements of the architected 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 architected 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 implementations. It must be implemented.

Coprocessor Instructions

The coprocessor mechanism is intended for special-purpose data manipulations, especially those which handle data larger than single words. The interconnection method is intended to permit instruction set extensions with minimal affect on the instruction execution rate, while maintaining short data communication paths between the coprocessors and the rest of the system. When the appropriate bit of the CCR (coprocessor configuration register) is set, coprocessor instructions are passed to the coprocessor and the defined operation occurs. When the CCR bit is 0, the instruction is terminated with an Assist emulation trap. No Assist exception trap is allowed.

When caches are implemented, coprocessors are connected to the CPU-cache interface. On cacheless systems, coprocessors are connected to the CPU-memory bus interface. They 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 five-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.

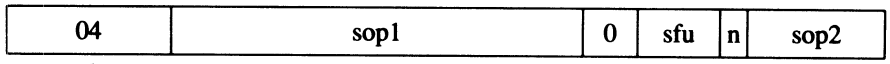
Some coprocessors are capable of supporting double-word load and store operations. These operations are implemented on all systems that support such coprocessors, even though they may require additional cycles for some machines. Coprocessor load and store instructions to I/O address spaces are undefined.

The CPU must also provide the current privilege level to coprocessors. Privilege levels could be broadcast each time they change or could be transmitted with each coprocessor operation. Use of the privilege level by the coprocessor is specific to each of the units. The operation paragraph 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) instruction, that is defined for coprocessors with unit identifiers four through seven. Coprocessors with unit identifiers zero through three have a mechanism to identify themselves that is individually architected.

SPECIAL OPERATION ZERO

SPOP0

Format: SPOP0,sfu,sop,n



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.

If hardware is not present to perform the operation, an Assist emulation trap occurs.

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).

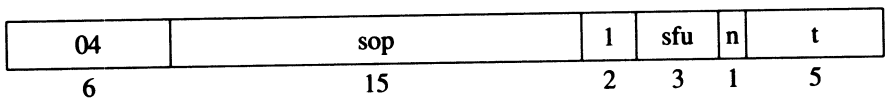
Operation:

```
sfu_operation(0,cat(sop1,sfu,n,sop2),IAOQ_Front{30..31});
if (n && sfu_condition(0,cat(sop1,sfu,n,sop2),IAOQ_Front{30..31}))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

SPECIAL OPERATION ONE

Format: SPOP1,sfu,sop,n t



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.

If hardware is not present to perform the operation, an assist emulation trap occurs.

Operation:
```
GR[t] ← sfu_operation(1,cat(sop,sfu,n),IAOQ_Front{30..31});
if (n && sfu_condition(1,cat(sop,sfu,n),IAOQ_Front{30..31}))
   PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

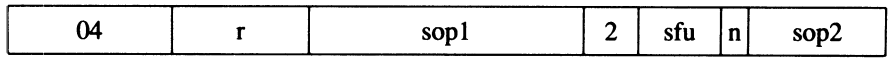
Notes: The SPECIAL OPERATION ONE instruction is used to implement the IDENTIFY SFU pseudo operation. This operation returns a 32-bit identification number from the special function unit *sfu* to general register *t*. The value returned is implementation dependent and is useful for configuration, diagnostics, and error recovery. The state of the SFU is undefined after this instruction.

Each implementation must chose an identification number that identifies the version of the SFU. The values zero and all ones are reserved. Emulation trap handlers return zero when executing this instruction. An 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
```

SPECIAL OPERATION TWO — SPOP2

Format: SPOP2,sfu,sop,n r



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.

If hardware is not present to perform the operation, an assist emulation trap occurs.

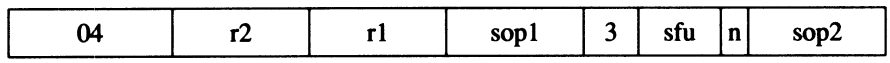
The *sop* field in the assembly language format is the concatenation of the *sop1* and *sop2* fields in the machine instruction, sop = cat(sop1,sop2).

Operation:
```
sfu_operation(2,cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r]);
if (n && sfu_condition(2,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



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.

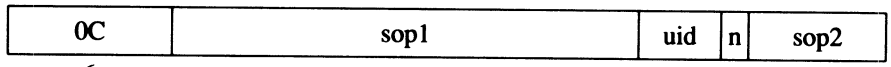
If hardware is not present to perform the operation, an assist emulation trap occurs.

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_operation(3,cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r1],GR[r2]);
if (n && sfu_condition(3,cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r1],GR[r2]))
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap
Assist exception trap

Format: COPR,uid,sop,n



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 32-bit identification number from the coprocessor uid into coprocessor register 0. This value is implementation dependent and is useful for configuration, diagnostic, and error recovery. The state of the coprocessor is undefined after this instruction executes, except that a store coprocessor word from register 0 will return the identification number.

Each implementation must chose an identification number that identifies the version of the coprocessor. The values zero 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.

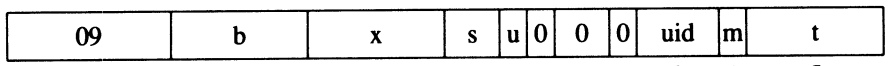
The IDENTIFY COPROCESSOR pseudo operation is coded as follows:

```
COPR,uid,0
```

COPROCESSOR LOAD WORD INDEXED

CLDWX

Format: CLDWX,uid,cmplt x(s,b),t



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 for the assembly language completer mnemonics.)

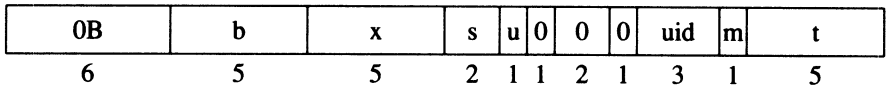
Operation:

```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
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,IAOQ_Front{30..31});
CPR[uid][t] ← mem_load(space,offset,0,31);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

COPROCESSOR LOAD DOUBLEWORD INDEXED — CLDDX

Format: CLDDX,uid,cmplt x(s,b),t



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 for the assembly language completer mnemonics.)

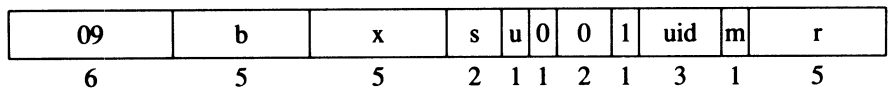
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,IAOQ_Front{30..31});
CPR[uid][t]{0..63} ← mem_load(space,offset,0,63);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

COPROCESSOR STORE WORD INDEXED — CSTWX

Format: CSTWX,uid,cmplt r,x(s,b)



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 for the assembly language completer mnemonics.)

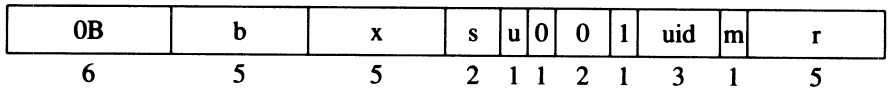
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,IAOQ_Front{30..31});
mem_store(space,offset,0,31,CPR[uid][r])
```

Exceptions:

Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit fault

Format: CSTDX,uid,cmplt r,x(s,b)



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 for the assembly language completer mnemonics.)

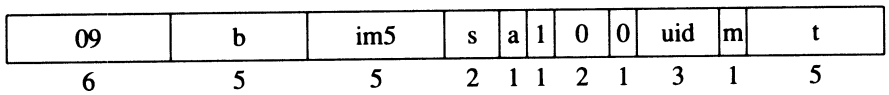
Operation:

```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
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,IAOQ_Front{30..31});
mem_store(space,offset,0,63,CPR[uid][r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

COPROCESSOR LOAD WORD SHORT — CLDWS

Format: CLDWS,uid,cmplt d(s,b),t



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 5-11 for the assembly language completer mnemonics.)

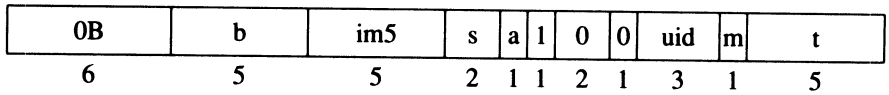
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,IAOQ_Front{30..31});
CPR[uid][t] ← mem_load(space,offset,0,31);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

COPROCESSOR LOAD DOUBLEWORD SHORT — CLDDS

Format: CLDDS,uid,cmplt i(s,b),t



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 5-9 for the assembly language completer mnemonics.)

Operation:

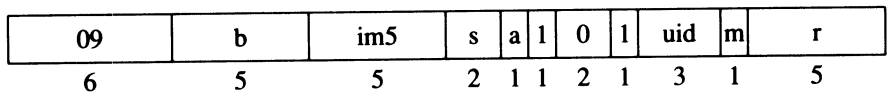
```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
switch (cmplt)
    {
    case ',MB':     offset ← GR[b] + low_sign_ext(im5,5);     /*a=1, m=1*/
                    GR[b] ← GR[b] + low_sign_ext(im5,5);
                    break;
    case ',MA':     offset ← GR[b];                           /*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,IAOQ_Front{30..31});
CPR[uid][t] ← mem_load(space,offset,0,63);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

COPROCESSOR STORE WORD SHORT

CSTWS

Format: CSTWS,uid,cmplt r,d(s,b)



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 5-9 for the assembly language completer mnemonics.)

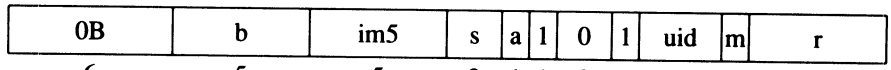
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,IAOQ_Front{30..31});
mem_store(space,offset,0,31,CPR[uid][r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

COPROCESSOR STORE DOUBLEWORD SHORT — CSTDS

Format: CSTDS,uid,cmplt r,d(s,b)



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 5-9 for the assembly language completer mnemonics.)

Operation:

```
if (s == 0)
    space ← SR[GR[b]{0..1} + 4];
else
    space ← SR[s];
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,IAOQ_Front{30..31});
mem_store(space,offset,0,31,CPR[uid][r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

6

Floating-point Coprocessor

Introduction

The HP Precision Architecture floating-point coprocessor is an assist processor (coprocessor unit 0) that executes floating-point arithmetic instructions and fully conforms to the requirements and recommendations of the IEEE standard. This coprocessor defines a set of operations that are performed on a register file which is independent of the CPU's general registers. The registers in the file are accessed through coprocessor load and store instructions. The COPROCESSOR OPERATION instruction is used to perform arithmetic and data movement functions.

The IEEE Floating-point Standard

The term "IEEE standard" or simply "the standard", when used in this chapter, refers to the *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*. As a matter of policy, Hewlett-Packard is committed to adherence to this standard.

The IEEE standard defines four floating-point data formats:

Format	Data Length	Implementation
Single-precision	32-bits	required
Single-precision extended	≥ 43-bits	recommended
Double-precision	64-bits	recommended
Double-precision extended	≥ 79-bits	recommended

HP Precision Architecture uses the double-precision format for the single-precision extended format. A quad-precision format with 128 bits of precision is defined as the double-precision extended format.

The exceptions listed below are defined by the IEEE standard and can be enabled/disabled separately or in groups:

- Invalid operation
- Division-by-zero
- Overflow
- Underflow
- Inexact

The standard requires that status information on disabled traps be maintained to indicate when exceptions have occurred. The exception trap mechanism may be used to avoid handling operations and exceptional conditions in hardware. See *Exceptions* for detailed information.

The standard does not require that all floating-point operations be performed in hardware, and it does not specify the instruction-set level presentation of the hardware. When hardware offers only a marginal performance advantage over software, the operation may be performed in software.

Definition of Terms

Listed below are the terms that have specific meaning for floating-point operations.

bias

A constant added to the exponent to center its range. The bias constant is +127 for single-precision, +1023 for double-precision, and +16383 for quad-precision.

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.

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.

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.

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.

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 $-\infty$.

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.

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.

Coprocessor Registers

The coprocessor contains sixteen 64-bit floating-point registers (Table 6-1), all of which may be read or written by instructions executing at any privilege level. Double-word load/store operations access the entire 64-bit register; 32-bit load/stores access only the most-significant portion of a register. The value of the unused portion of a register is undefined.

Table 6-1. Coprocessor 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	
16-31	Reserved	

Registers 0 - 3 are used for status and exceptions, 4 - 15 are data registers. Register specifiers 16 - 31 are reserved.

Registers 0 - 3 are partitioned into eight 32-bit registers to provide a status register and seven exception registers. The status register is accessed using single-word and double-word load and store instructions.

The exception registers are accessed using double-word load and store instructions. Single-word load/stores of exception registers are undefined. Exception registers record instructions (perhaps pipelined) that were started then abandoned due to exceptional conditions.

Register 0, when used as the source for a store or the target for a load instruction, accesses the status register. Register *specifier* 0, when used as a source by non-load/store operations, encodes a floating-point zero. The use of register specifier 0 as a destination or the use of register specifiers 1, 2 or 3 is an undefined operation for non-load/store operations.

Data registers (registers 4 - 15) are used as operands. Each data register can hold a number in single-word or double-word format. Adjacent even-odd register pairs can hold quad-word format numbers. The quad formats must specify even register numbers, the use of odd-numbered specifiers is an undefined operation.

Status Register

The Status register (Figure 6-1) controls arithmetic rounding mode and the enabling of user-level traps and it indicates exceptions that may have occurred without being trapped.

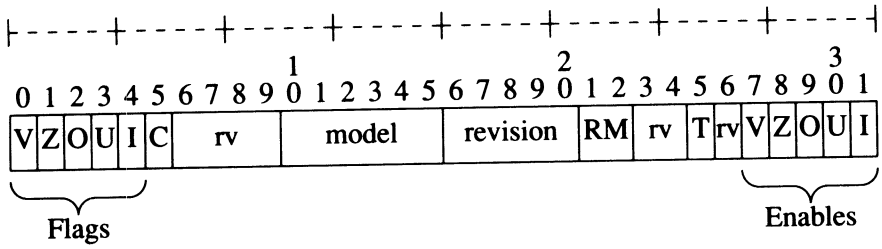


Figure 6-1. Status Register.

The fields labeled *rv* are reserved bits.

The model field contains the implementation-dependent model number. Model number zero (0) is reserved for the software emulation routines. The revision field contains the implementation-dependent version number.

The RM field controls the rounding mode for all floating-point operations:

Rounding mode	Description
0	Round to nearest
1	Round toward zero
2	Round toward $+\infty$
3	Round toward $-\infty$

NOTE

In the parts of the IEEE standard quoted in this chapter, references to other sections of the standard are enclosed in parentheses. References to sections of this document (the one you are reading) are enclosed in square brackets.

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*]. Except for binary-decimal conversion (whose weaker conditions are specified in 5.6 [not included]), every operation specified in Section 5 [see *Floating-Point Instruction Set*] shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then round that result according to one of the modes in this section.
>
> The rounding modes affect all arithmetic operations except comparison and remainder. The rounding modes may affect the signs of zero sums (6.3) [see *Sign Bit*], and do affect the thresholds beyond which overflow (7.3) [see *Overflow Exception*] and underflow (7.4) [see *Underflow Exception*] 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 $2^{E_{max}}(2-2^{-p})$ shall round to ∞ with no change in sign; here E_{max} and p are determined by the destination format (§3) [see *Binary Floating-Point Formats*] unless overridden by a rounding precision mode (4.3) [not possible in HP Precision Architecture].

[§]**4.2 Directed Roundings.** An implementation shall also provide three user-selectable directed rounding modes: round toward $+\infty$, round toward $-\infty$, and round toward 0. When rounding toward $+\infty$, the result shall be the format's value (possibly $+\infty$) closest to and no less than the infinitely precise result. When rounding toward $-\infty$, 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 IEEE standard defines the five exceptions listed below. The floating-point status register uses the same names for both the exception flag bits and the exception enable bits.

Bit Name	Description
V	Invalid operation
Z	Division-by-zero
O	Overflow
U	Underflow
I	Inexact result

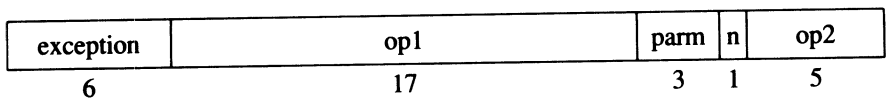
An enable bit and a flag bit is associated with each exception (see Figure 6-1). When an exception occurs and the corresponding exception enable bit is set, the exception is signaled with trap. When an exception occurs and the corresponding exception enable bit is not set, the corresponding flag bit is set and the default action is taken. The result when the default action is taken may differ from the result when an exception is signaled with a trap. The exception flags are never cleared as a side effect of non-load/store floating-point instructions, but they may be set or cleared by load instructions.

The C-bit in the floating-point status register records the conditional result of the most recent compare instruction. No other non-load/store instruction affects this bit, but it may be set or cleared by load instructions.

The T-bit in the floating-point status register is used by save and restore software to record the state of traps. It is set by the coprocessor whenever a trap is to be signaled. Generally, when the T-bit is set, the next floating-point instruction is aborted with a trap. See *Saving And Restoring State* for information on state swapping and *Exceptions* for a detailed discussion of the T-bit's operation.

Exception Registers

The exception registers contain floating-point instructions that have completed execution and are no longer in the IA queue. Figure 6-2 shows the format of the exception registers. These registers hold instructions in the same format they occur in memory. The fields specifying the coprocessor operation and floating-point unit number have been replaced with fields indicating the cause of the exception together with additional data related to the exception. The remaining fields are duplicates of the original instruction.



Exception	Description
000000	No exceptions
100000	Invalid operation
010000	Division-by-zero
001000	Overflow
000100	Underflow
000010	Inexact
000001	Unimplemented

Field	Description
n	Instruction nullify bit
op1	Instruction opcode 1
op2	Instruction opcode 2
parm	Instruction parameters

Figure 6-2. Exception Register Format.

The bits of the *exception* field are set for instructions that complete with an IEEE standard exception or an exception used to emulate certain features of the IEEE standard. If two exceptions occur simultaneously, the exception field contains the inclusive OR of the values for each exception. The only exceptions that coincide are inexact with overflow and inexact with underflow.

When no exceptions are present, the other fields of the exception register are undefined.

The *parm* field is defined for exceptions requiring additional information to handle the exception. For example, untrapped underflows (that is, underflow was detected but no trap was requested by the user) need information on the rounding and exactness of the result to properly denormalize.

Data Types

Each of the floating-point data registers may contain values in a number of formats. Formats that require less than the full width of the floating-point register are left-justified. Formats that exceed the full width of a floating-point register (> 64 bits) are packed into adjacent even-odd pairs of registers. These longer formats are assembled together into double-length registers as shown in Figure 6-3.

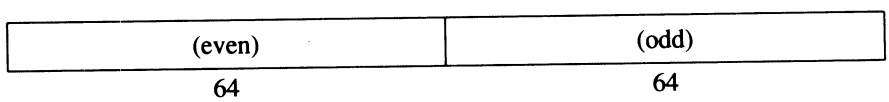


Figure 6-3. Floating-point Data Register Format.

Floating-point register fields are packed into words, double-words, or quad-words so that load and store operations do not require field-shuffling or tag bits. The unused portion of data registers is undefined.

While this allows reasonable freedom when working with shorter formats, double-word stores of single-word values must still keep the 32-bit value left-justified in the double-word memory location. State save requires this restriction.

Binary 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, $f=.b_1 b_2 \cdots b_{p-1}$.

The range of the unbiased exponent E includes every integer between two values E_{min} and E_{max} inclusive, and also two other reserved values: $E_{min}-1$ to encode ± 0 and denormalized numbers, and $E_{max}+1$ to encode $\pm\infty$ and NaNs. Each representable nonzero numerical value has just one encoding.

The value of a number, v, is determined by the following:

1. If $E = E_{max}+1$ and $f \neq 0$, then v is NaN, regardless of s.
2. If $E = E_{max}+1$ and $f = 0$, then $v = (-1)^s \infty$.
3. If $E_{min} \leq E \leq E_{max}$, then $v = (-1)^s 2^E (1.f)$.
4. If $E = E_{min}-1$ and $f \neq 0$, then $v = (-1)^s 2^{E_{min}} (0.f)$.
5. if $E = E_{min}-1$ and $f = 0$, then $v = (-1)^s 0$.

If v is NaN, the most-significant bit of f determines whether the value is a signaling or quiet NaN. v is a signaling NaN if the most-significant bit of f is a one; otherwise v is a quiet NaN.

The format parameters in the preceding description have the values listed in Table 6-2.

Table 6-2. 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
exponent width in bits	8	11	15
format width in bits	32	64	128

The bit positioning for the binary floating-point formats are shown in Figure 6-4.

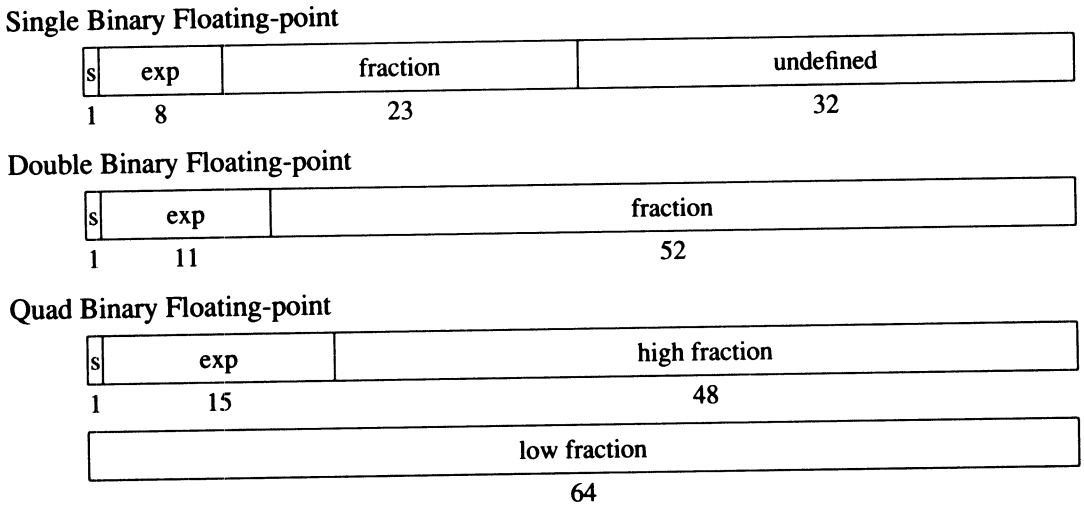


Figure 6-4. Binary Floating-point Formats.

Binary Fixed-point

Binary fixed-point values are held in two's complement format as shown in Figure 6-5.

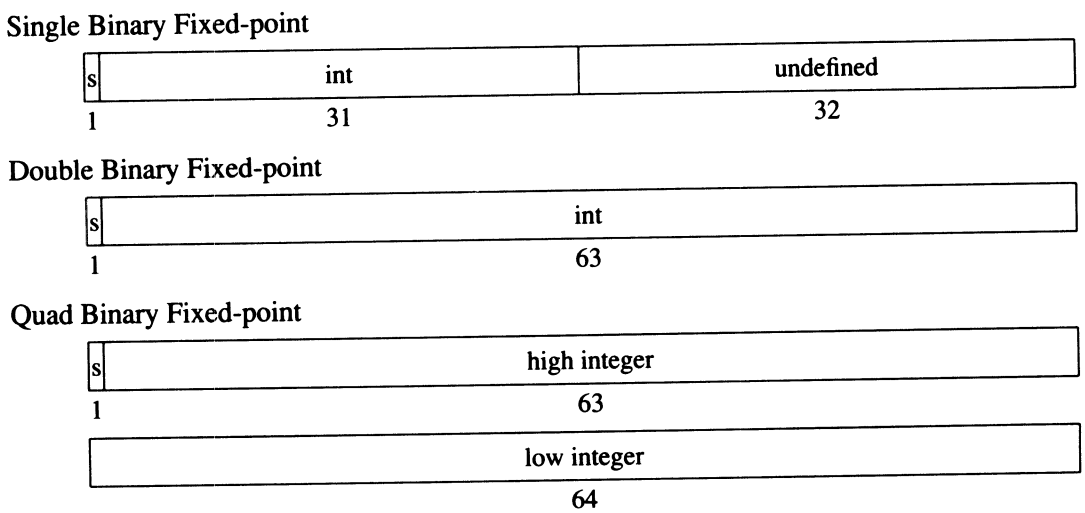


Figure 6-5. Binary Fixed-point Formats.

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 on ∞ is always exact and therefore shall signal no exceptions, except for the invalid operation specified for ∞ in 7.1 [see *Invalid Exception*]. The exceptions that do pertain to ∞ are signaled only when:
>
> (1) ∞ is created from finite operands by overflow (7.3) [see *Overflow Exception*] or division by zero (7.2) [see *Divide by Zero Exception*], with the corresponding trap disabled, or
>
> (2) ∞ is an invalid operand (7.1) [see *Invalid 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.

Signaling NaNs cause an invalid exception for all arithmetic instructions except conversions to integer formats. Load, store, copy and abs are not arithmetic and do not signal an invalid operation exception. Every arithmetic instruction involving a signaling NaN, if no trap occurs and if a floating-point result is delivered, delivers the NaN converted to a quiet NaN. If both operands are signaling NaNs then the *rl* register contents is delivered. The creation of a quiet NaN from a signaling NaN cannot change any portion of the fraction except the leading two bits (clears the first, and optionally sets the second).

An arithmetic instruction involving one input quiet NaN, that delivers a floating-point result, does not signal an exception and delivers as its result the input quiet NaN. Similar instructions involving two input quiet NaNs deliver the *rl* register contents. The creation of a quiet NaN (neither input is a NaN) sets the bit to the right of the most-significant fraction bit and clears the remaining fraction bits.

Untrapped conversions of a NaN to a smaller floating-point format preserves the most-significant portion of the fraction that fits in the destination. Untrapped conversions of a NaN to a larger floating-point format augments the fraction with zeros. Conversions to integer formats involving an input NaN must preserve the source registers, deliver no results, and signal an unimplemented 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 $-\infty$, 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.

Exceptions

An exception is an unexpected or unusual event that occurred during the execution of a floating-point instruction. Exceptions can occur when using numbers that are not valid operands for a particular operation (for example, division-by-zero). They may also occur when using reserved operations, reserved operands or implementation-restricted operations. For example, the square root instruction completes with an exception when an operand less than zero is used or when hardware does not implement the function. Besides the IEEE-defined exceptions (invalid operation, division-by-zero, overflow, underflow, and inexact), there are two additional exceptions: unimplemented and reserved-op. These two exceptions provide a mechanism to fully implement the IEEE standard with a combination of software and hardware.

When an exception is detected, the associated floating-point instruction is completed in an exception-dependent way. Exceptions are signaled in one of three ways:

1. A non-trapping exception is signaled by setting the associated status register flag. A default result is produced and normal instruction sequencing continues.
2. An immediate-trapping exception is signaled by causing an assist exception trap when the instruction is issued (i.e. pointed to by the IA queue). This exception does not produce results. The interruption instruction register, IIR, contains a copy of the instruction and its contents are interpreted by the exception trap handler. The reserved-op exception is the only immediate-trapping exception.
3. An exception is signaled by causing an assist exception trap some time after the instruction is issued (i.e. no longer pointed to by the IA queue), a delayed-trap exception. Except for the reserved-op exception, exceptions that are signaled with a trap are always delayed-trap exceptions.

Assist exception traps can only be taken when a floating-point coprocessor instruction is executing. The unit field of the IIR register will contain a zero value, indicating that the floating-point coprocessor is the cause of the exception trap. An IEEE-defined exception (invalid, division-by-zero, overflow, underflow, or inexact) must be signaled by a trap when the associated trap-enable bit is set. If the trap-enable bit is clear, then a non-trapping exception is signaled, the value specified by the IEEE standard is stored, and the associated flag bit is set.

Unimplemented and reserved-op exceptions are always signaled with a trap. There are no corresponding enable or flag bits. These exceptions are normally invisible to user-level code. They are used to support

a minimal hardware configuration that does not handle the full set of operations, denormalized numbers, or NaN values without software assistance.

PROGRAMMING NOTE

Reserved-op exceptions may occur with load and store instructions. Trapping is not delayed for reserved-op exceptions because the address of the memory operand is available only when the instruction is executed. When the trap occurs, the contents of the IIR is interpreted, the instruction pointed to by the front element of the IA queue is nullified, and control is returned to the trapping process. For all other exceptions the exception registers are processed and the instruction pointed to by the front element of the IA queue is retried.

Results for Non-trapped and Trapped Exceptions

IEEE-defined exceptions that are not signaled with a trap produce the results shown in Table 6-3.

Table 6-3. Non-trapped Exception Results.

Exception type	Non-trapped result
Invalid operation	quiet NaN
Division-by-zero	properly signed ∞
Overflow	rounded result
Underflow	rounded result
Inexact	rounded result

For exceptions signaled with a trap, the status of the source and target registers depend on the exception. For certain exceptions (overflow and underflow), a value related to the true result is placed in the floating-point result register. Others require that the source registers be preserved. Table 6-4 specifies these requirements.

Table 6-4. Trapped Exception Results.

Exception type	Trapped result
Reserved op	original operand values
Invalid operation	original operand values
Division-by-zero	original operand values
Overflow	rounded bias-adjusted result
Underflow	rounded bias-adjusted result
Inexact	rounded result
Unimplemented	original operand values

Exception Register Operation

When a delayed trapping exception is signaled with a trap, the exception type number of the trap is stored in exception register 1 along with the instruction that caused the exception trap. The field specifying a coprocessor operation (most-significant 6 bits) is replaced with the exception type number and the floating-point unit number is replaced with values specific to the exception. Since the reserved-op exception traps immediately, the IIR records the instruction. Neither the exception register nor the T-bit in the floating-point status register are set.

The instruction contained in exception register 1 is called the *excepting instruction* since it completed with an exception and caused the trap. The instruction being executed when the trap was taken is called the *trapped instruction*. The *current instruction* always refers to the instruction pointed to by the front position of the IA queue. It may be an instruction for the processor or may be a floating-point coprocessor instruction.

Once an exception trap has been processed, the exception registers must be cleared before non-load/store instructions can be executed. Execution of non-load/store floating-point instructions with non-zero exception registers, is undefined.

Exception Traps

When a delayed trapping exception that must be signaled by a trap is detected, the coprocessor:

1. completes all instructions that are executing,
2. sets the exception registers to their architected values,
3. sets the T-bit in the floating-point status register, and
4. if the current instruction is a floating-point instruction, aborts it with an assist exception trap. Otherwise, the coprocessor aborts the next floating-point instruction with an assist exception trap.

Since the coprocessor continues to trap if its T-bit floating-point status register is set, the trap handler must first clear it by executing a double-word store of the status register (see *Saving and Restoring State*).

Pipelined instructions that complete with an exception are recorded in exception registers 2 through 7. These registers form a queue of exceptions from the oldest (register 2) to the newest (register 7). The queue limits the depth of any pipeline to seven instructions. If an instruction completes without a trapping exception, no record of that instruction appears in the exception registers. A zero exception-type field indicates *no-exception*, and leaves the remainder of the exception register undefined. The exception queue need not be packed. Pipelined instructions that complete without a trapping exception can remain with a zero exception-type field. In summary, exception register 1 contains the instruction causing the trap and exception registers 2 through 7 contain newer instructions that also completed with an exception.

Trapping exceptions may need to abort the current instruction whenever that instruction has data dependent interlocks resulting in the detection of an exception. For example, an access to the status register forces the completion of an earlier pipelined add instruction; the add completes with an overflow. In this case, the current instruction (the status register access) is aborted with a trap. After the trap is processed, the status register access is retried.

NOTE

In the following subsections, quotes are taken from the IEEE standard. The unimplemented and reserved-op exceptions may still occur even when the standard indicates that no exception occurs.

Invalid Operation Exception

From the standard:

> [§]**7.1 Invalid operation.** The invalid operation exception is signaled if an operand is invalid for the operation to be performed. The result, when the exception occurs without a trap, shall be a quiet NaN (6.2) [see *Operations with NaNs*], provided the destination has a floating-point format. The invalid operations are:
>
> (1) Any operation on a signaling NaN (6.2) [see *Operations with NaNs*];
>
> (2) Addition or subtraction: magnitude subtraction of infinities like $(+\infty) + (-\infty)$;
>
> (3) Multiplication: $0 \times \infty$;
>
> (4) Division: 0/0 or ∞/∞;
>
> (5) Remainder: x REM y, where y is zero or x is infinite;
>
> (6) Square root if the operand is less than zero;
>
> (7) Conversion of a binary floating-point number to an integer format or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled; and
>
> (8) Comparison of predicates involving "<" or ">" without "?", when the operands are unordered (5.7, Table 4) [see *Comparisons*].

Point seven concerning conversions to integer or decimal formats does not apply to this architecture. Integer conversion exceptions are signaled using an unimplemented exception. Decimal formats are not a part of the architecture. Software may assert an invalid exception for other operations that are invalid for the given source operands, such as $\ln(-1)$ or $\cos^{-1}(2)$.

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*].
>
> Software may assert this exception for other operations that produce a signed infinity, such as $\ln(0)$ or $csc(0)$ or 0^{-1}.

Overflow Exception

From the standard:

> [§]**7.3 Overflow.** The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (§4) [see *Rounding Modes*], were the exponent range unbounded. The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate result as follows:
>
> (1) Round to nearest carries all overflows to ∞ with the sign of the intermediate result.
>
> (2) Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result.
>
> (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 +∞.
>
> (5) Trapped overflows on all operations except conversion shall deliver to the trap handler the result obtained by dividing the infinitely precise result by 2^a and then rounding. The bias adjust a is 192 in the single, 1536 in the double, and $3 \times 2^{n-2}$ in the extended format, where n is the number of bits in the exponent field.

For quad format, n is 15 and results in a bias of 24576. The following defines the actions taken by the floating-point coprocessor on detecting overflow in situations left undefined by the IEEE standard.

Trapped overflow on conversion from a binary floating-point format to another binary floating-point format provides the result in the destination format, with the exponent bias adjusted. If the result lies too far outside the range for the bias to be adjusted, the source register must be preserved and an unimplemented exception signaled. The architectural organization of the register file does not allow meeting IEEE standards for this condition. Software can report the proper result.

Untrapped overflow on conversion from a binary floating-point format to another binary floating-point returns the result determined by the rounding mode and the sign of the source operand.

The overflow exception is not signaled for integer results. Integer overflows are signaled with an unimplemented exception.

Implementations are allowed to signal an overflow trap with an overflow exception when the enable bit is clear (when a default result was to be delivered). As usual, the rounded bias-adjusted result is delivered to the trap handler. Software determines the correct result transparent to the user. For example, an implementation is able to detect overflow and provide rounded bias-adjusted results but cannot easily generate infinities and largest finite numbers. For overflows, such an implementation can ignore the overflow trap-enable bit and proceed as if it were set (by trapping).

Underflow Exception

From the standard:

> [§]**7.4 Underflow.** Two correlated events contribute to underflow. One is the creation of a tiny non-zero 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 after rounding (when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E\,\mathrm{min}}$).

Loss of accuracy is detected on *inexact result* (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).

Again from the standard:

> . . . When an underflow trap . . . is not enabled (the default case) underflow shall be signaled (via the underflow flag) only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or $\pm 2^{E\,\mathrm{min}}$. When an underflow trap has been implemented [always true in this architecture] and is enabled, underflow shall be signaled when tininess is detected regardless of loss of accuracy. Trapped underflows on all operations except conversion shall deliver to the trap handler the result obtained by multiplying the infinitely precise result by 2^a and then rounding. The bias adjust a is 192 in the single, 1536 in the double, and $3 \times 2^{n-2}$ in the extended format, where n is the number of bits in the exponent field.

For quad format, n is 15 and the bias of 24576. The following defines the actions taken by the floating-point coprocessor on detecting underflow in situations left undefined by the IEEE standard.

Trapped underflow on conversion from a binary floating-point format to another binary floating-point format provides the result in the destination format with the exponent bias adjusted. If the result lies too far outside the range for the bias to be adjusted, the source register must be preserved and an unimplemented exception signaled. The architectural organization of the register file precludes meeting IEEE standards for this condition. Software can report the proper result.

Untrapped underflow on conversion from a binary floating-point format to another binary floating-point format returns the result determined by the rounding mode and the sign of the source operand.

Conversion to an integer format cannot underflow. The result when the magnitude of the source operand is less then one is 0, "+1", or "−1" depending on the rounding mode and the sign of the source operand.

Implementations are allowed to signal an underflow exception with an underflow trap when the enable bit is clear (a default result is to be delivered). An additional parameter field must be delivered in the exception register to assist the software. As usual, the rounded bias-adjusted result is delivered to the trap handler and software determines the correct denormalized result transparent to the user. For example, an implementation may be able to detect underflow and provide rounded bias-adjusted results but can not easily generate denormalized numbers. Then such an implementation can ignore the underflow trap-enable bit and proceed as if it were set.

When the coprocessor signals an underflow exception with an underflow trap and the enable bit is clear, two additional bits in the exception register's *parm* field must be provided (see Figure 6-6). The first is the inexact (I) bit which is set if the rounded bias-adjusted result is not the infinitely precise result, also biased. The second parameter bit, the round away (RA) bit, is set whenever the result is rounded away from zero. The trap handler uses this information to denormalize the result and prevent errors caused by rounding twice.

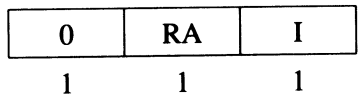


Figure 6-6. Exception Register Parm Field.

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].

Inexact exceptions are also signaled on conversion to integer formats where the result is not exact.

Unimplemented Exception

If an instruction that the hardware does not perform is specified, then an unimplemented exception is signaled. An unimplemented exception always causes a trap for which there are no corresponding enable or flag bits. This trap cannot be disabled. Reserved non-load/store instructions may optionally be reported using an unimplemented exception.

The instruction may be emulated in software, possibly using implemented floating-point unit instructions to accomplish the emulation. Implementations may complete pipelined instructions when an earlier instruction completes with an exception by marking the waiting instructions as *unimplemented* in the exception registers. The exception handler will simply emulate those instructions.

An unimplemented exception is also signaled when:

1. A conversion from a floating-point format to a floating-point format lies too far outside the range for the exponent to be adjusted (gross over/underflow).
2. A conversion from a floating-point format to an integer format overflows.
3. A conversion to an integer format involves a NaN.

Reserved-op Exception

If a reserved operation or operand is specified, then a reserved-op exception is signaled. A reserved-op exception always causes a trap for which there are no corresponding enable or flag bits. The trap cannot be disabled. Reserved non-load/store operations or operands may also be signaled with an unimplemented exception.

Trap Handlers

The IEEE standard strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions. The trap handler computes or specifies a substitute result to be placed in the destination register of the operation. The trap handler determines what operation was being performed and what exceptions occurred during the operation. The destination's format is found by examining a

copy of the operation exception register returned by the floating-point coprocessor. 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 invalid operation and divide-by-zero exceptions, the trap handler has access to the operand values by examining the source registers of the instruction.

The IEEE standard recommends that, if enabled, the overflow and underflow traps take precedence over a simultaneous inexact trap.

Saving and Restoring State

Sixteen double-word coprocessor loads/stores are sufficient to save or restore the coprocessor's state. A pending trap is canceled by executing a double-word store of coprocessor register zero which forces the completion of pipelined instructions, suppresses any trap that might otherwise occur and completes the store. If a trap is canceled, then the value written to memory has the T-bit of the floating-point status register set; otherwise it is cleared. Canceling a pending trap also clears the T-bit in the floating-point status register. This special treatment of a double-word store allows the save routine to be nested, does not require the assistance of a trap handler, and need not have the IA queue enabled. Double-word stores of registers 1 through 3 with a pending trap are undefined.

The restore sequence must also have a way to re-arm a canceled trap. A double-word load of coprocessor register zero which sets the T-bit re-arms a trap. The next floating-point instruction will cause a trap (apart from the double-word register zero store) and the exception registers remain unchanged. Single-word loads of register zero that set the T-bit are undefined.

Canceling and re-arming traps allows the following state save and restore sequence.

```
; 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    FPR14,8(SaveAreaPtr)    ;save data register
        FSTDS       FPR15,0(SaveAreaPtr)    ;save last data register
```

```
; enter with SaveAreaPtr pointing at the last double-word of the save area.
RSTFPU
        FLDDS       0(SaveAreaPtr),FPR15    ;restore data register
        FLDDS,MB    -8(SaveAreaPtr),FPR14   ;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
```

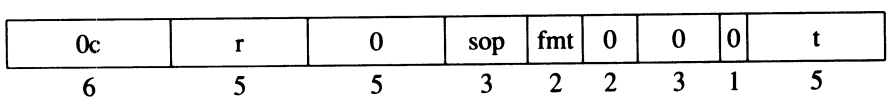
The only required ordering in this sequence saves register zero first and restores it last.

Instruction Format

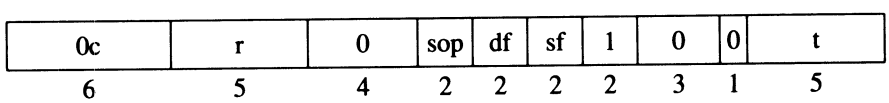
Floating-point instructions use coprocessor protocols and the 5-bit register specifiers map directly to the associated floating-point register. For quad format operations the low bit of the register specifier must be zero. Non-zero values are undefined.

The basic operations are two- and three-register operations. Bits 21 and 22 of the coprocessor instruction indicate the floating-point class for the operation to be performed while bits 19 and 20 indicate the format of the input operands. For operation classes 0, 2, and 3, bits 16-18 specify the operation; class 1 instructions (conversions) use bits 15 and 16 to specify the operation and bits 17 and 18 to specify the destination format. All class 0, 2, and 3 instructions use the same format for sources and destinations; class 1 instructions specify both source and destination formats. In Figure 6-7, *fmt*, *df*, and *sf* correspond to the format, destination format, and source format respectively.

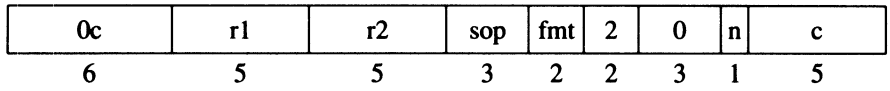
Floating-point operation class zero: 1 source, 1 destination



Floating-point operation class one: 1 source, 1 destination



Floating-point operation class two: 2 sources, no destination



Floating-point operation class three: 2 sources, 1 destination

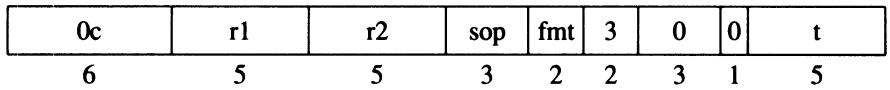


Figure 6-7. Floating-point Instruction Format.

Floating-Point Instruction Set

The floating-point instructions are part of the HP Precision Architecture standard instruction set. The assist emulation trap may be used on some systems to implement these instructions using software. If no floating-point hardware is present, the assist emulation trap is taken on each instruction that refers to the floating-point coprocessor. The instructions are emulated by software routines.

Such implementations incur many cycles of overhead upon execution of these instructions as well as possible increased interrupt latency. This overhead is likely to be incurred only on systems that have low performance requirements for floating-point operations such as device controllers or personal/portable computers.

To minimize interrupt latency while retaining modest overhead, load and store coprocessor instructions may be emulated with interruptions turned off while the longer floating-point operation instructions may be emulated after restoring sufficient CPU state to execute with interruptions on.

The floating-point registers for a process may be maintained by the emulation code in the region that would otherwise be used by the operating system for saving and restoring the registers. They can be referenced using absolute addresses by code running at the most privileged level.

Operand Formats

Each floating-point instruction can be applied to a number of operand formats. The operand format for an instruction is specified by a 2-bit field and signaled with the appropriate instruction completer. Table 6-5 indicates the operand format codes and their instruction completers.

Table 6-5. Floating-point Operand Format Completers.

fmt,df,sf	Description	Code
	single-precision format (32 bits)	0
SGL	single-precision format (32 bits)	0
DBL	double-precision format (64 bits)	1
QUAD	quad-precision format (128 bits)	3

In the above table, the field column "fmt, df, sf" is in assembly language format and the column labeled "code" is in machine language format.

Floating-point Operations

All of the floating-point operations, except for conversions to fixed-point formats, produce floating-point results. Source and destination formats are the same. Conversions specify the destination format independent from the source format. Table 6-6 defines the floating-point operations, their mnemonics, class, and opcodes.

Table 6-6. Floating-point Operations.

Opcode	Class	Mnemonic	Operation
0-1	0		undefined
2		FCPY	Copy
3		FABS	Absolute value
4		FSQRT	Square root
5		FRND	Round to integer
6-7			reserved
0	1	FCNVFF	Convert from floating-point to floating-point
1		FCNVXF	Convert from fixed-point to floating-point
2		FCNVFX	Convert from floating-point to fixed-point
3		FCNVFXT	Convert from floating-point to fixed-point with explicit round to zero rounding.
0	2	FCMP	Arithmetic compare
1		FTEST	Test condition code
2-7			reserved
0	3	FADD	Add
1		FSUB	Subtract
2		FMPY	Multiply
3		FDIV	Divide
4		FREM	Remainder
5-7			reserved

Conversion between all formats are allowed except the identity conversions single-to-single, double-to-double, and quad-to-quad floating-point.

Table 6-7 is derived from Table 4 in the IEEE standard. It describes the twenty-six predicates named in the standard. An additional six predicates are listed, which round out the set of possible predicates based on the conditions tested by a comparison. Invalid exceptions are only given on comparisons that include the characters "<" or ">," but not "?".

The "!" predicate is used to indicate the logical NOT of the condition specified. The "?" predicate is used to indicate an *unordered* condition. The term unordered is used when at least one of the two comparison operands is a NaN.

The nomenclature for the predicates leaves some types of comparisons unnamed, such as "test for equality and trap on unordered". When the "?" character is not present in the predicate and one of the two operands is a NaN, the tests for equality do not raise the invalid exception. The notations, "=T" and "!=T", are used in the above case.

Table 6-7. Floating-point Compare Conditions.

cond	Relations				c
	>	<	=	unordered	
false?	F	F	F	F	0
false	F	F	F	F *	1
?	F	F	F	T	2
!<=>	F	F	F	T *	3
=	F	F	T	F	4
=T	F	F	T	F *	5
?=	F	F	T	T	6
!<>	F	F	T	T *	7
!?>=	F	T	F	F	8
<	F	T	F	F *	9
?<	F	T	F	T	10
!>=	F	T	F	T *	11
!?>	F	T	T	F	12
<=	F	T	T	F *	13
?<=	F	T	T	T	14
!>	F	T	T	T *	15

cond	Relations				c
	>	<	=	unordered	
!?<=	T	F	F	F	16
>	T	F	F	F *	17
?>	T	F	F	T	18
!<=	T	F	F	T *	19
!?<	T	F	T	F	20
>=	T	F	T	F *	21
?>=	T	F	T	T	22
!<	T	F	T	T *	23
!?=	T	T	F	F	24
<>	T	T	F	F *	25
!=	T	T	F	T	26
!=T	T	T	F	T *	27
!?	T	T	T	F	28
<=>	T	T	T	F *	29
true?	T	T	T	T	30
true	T	T	T	T *	31

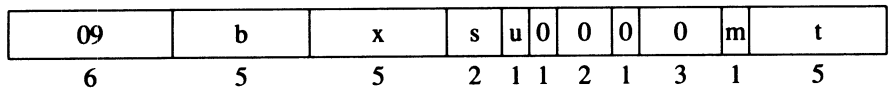
In the above table, an asterisk indicates an invalid operation exception if unordered. The column labeled cond is in assembly language format and the column labeled c is in machine language format.

Instruction Set Description

In the following pages, the notation, FPR, refers to floating-point coprocessor registers 0 through 15. Refer to the instruction notation section of chapter 5 for the explanation of the operation section. The mem_load and mem_store descriptions are located in the memory reference instruction section.

FLOATING-POINT LOAD WORD INDEXED — FLDWX

Format: FLDWX,cmplt x(s,b),t



Purpose: To load a word into a floating-point coprocessor register.

Description: The aligned word at the effective address is loaded into register *t* of the floating-point coprocessor. 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.

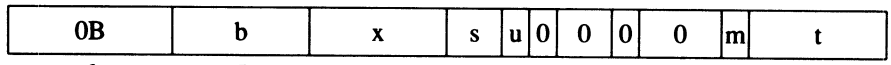
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

FLOATING-POINT LOAD DOUBLEWORD INDEXED — FLDDX

Format: FLDDX,cmplt x(s,b),t



Purpose: To load a double-word into a floating-point coprocessor register.

Description: The aligned double word at the effective address is loaded into register *t* of the floating-point coprocessor. 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.

Operation:

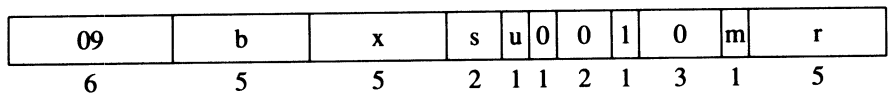
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',S':     offset ← GR[b] + lshift(GR[x],3);          /*u=1, m=0*/
                  break;
   case ',M':     offset ← GR[b];                            /*u=0, m=1*/
                  GR[b] ← GR[b] + GR[x];
                  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);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

FLOATING-POINT STORE WORD INDEXED

FSTWX

Format: FSTWX,cmplt r,x(s,b)



Purpose: To store a word from a floating-point coprocessor register.

Description: Register *r* of the floating-point coprocessor is stored into 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.

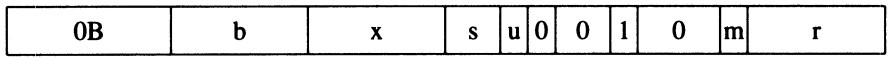
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,FPR[r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

FLOATING-POINT STORE DOUBLEWORD INDEXED — FSTDX

Format: FSTDX,cmplt r,x(s,b)



Purpose: To store a double-word from a floating-point coprocessor register.

Description: Register *r* of the floating-point coprocessor is stored into 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 3. The completer, encoded in the *u* and *m* fields of the instruction, also specifies base register modification.

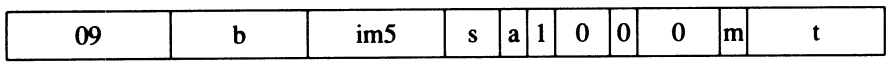
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,FPR[r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

FLOATING-POINT LOAD WORD SHORT — FLDWS

Format: FLDWS,cmplt d(s,b),t



Purpose: To load a word into a floating-point coprocessor register.

Description: The aligned word is loaded, from the effective address, into register *t* of the floating-point coprocessor. 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.

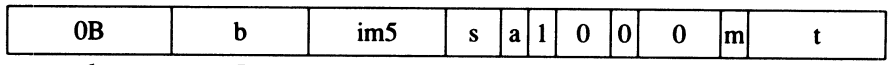
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

FLOATING-POINT LOAD DOUBLEWORD SHORT — FLDDS

Format: FLDDS,cmplt d(s,b),t



0B	b	im5	s	a	1	0	0	0	m	t
6	5	5	2	1	1	2	1	3	1	5

Purpose: To load a double-word into a floating-point coprocessor register.

Description: The aligned double word is loaded, from the effective address, into register *t* of the floating-point coprocessor. 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.

Operation:

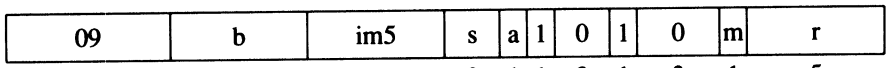
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);      /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                            /*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);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap

FLOATING-POINT STORE WORD SHORT

FSTWS

Format: FSTWS,cmplt r,d(s,b)



Purpose: To store a word from a floating-point coprocessor register.

Description: Register *r* of the floating-point coprocessor is stored into 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.

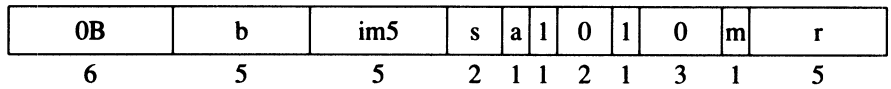
Operation:

```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
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,FPR[r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

FLOATING-POINT STORE DOUBLEWORD SHORT — FSTDS

Format: FSTDS,cmplt r,d(s,b)



Purpose: To store a double-word from a floating-point coprocessor register.

Description: Register *r*, of the floating-point coprocessor, is stored into 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.

Operation:

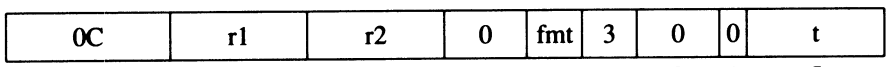
```
if (s == 0)
   space ← SR[GR[b]{0..1} + 4];
else
   space ← SR[s];
switch (cmplt)
   {
   case ',MB':   offset ← GR[b] + low_sign_ext(im5,5);        /*a=1, m=1*/
                 GR[b] ← GR[b] + low_sign_ext(im5,5);
                 break;
   case ',MA':   offset ← GR[b];                              /*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,FPR[r]);
```

Exceptions:
Assist emulation trap
Assist exception trap
Data TLB miss fault/data page fault
Data memory protection trap/Unaligned data reference trap
Page reference trap
Data memory break trap
TLB dirty bit trap

FLOATING-POINT ADD

FADD

Format: FADD,fmt r1,r2,t



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.

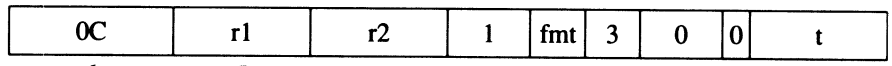
Description: The floating-point registers or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically added. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register or register pair specified by *t*.

Operation: FPR[t] ← FPR[r1] + FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT SUBTRACT — FSUB

Format: FSUB,fmt r1,r2,t



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 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 floating-point register or register pair specified by *t*.

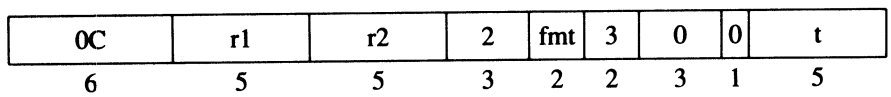
Operation: FPR[t] ← FPR[r1] – FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT MULTIPLY

FMPY

Format: FMPY,fmt r1,r2,t



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.

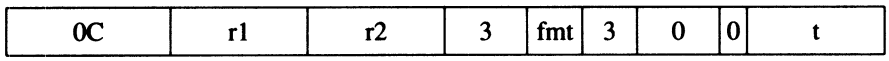
Description: The floating-point registers or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register or register pair specified by *t*.

Operation: FPR[t] ← FPR[r1] * FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT DIVIDE — FDIV

Format: FDIV,fmt r1,r2,t



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.

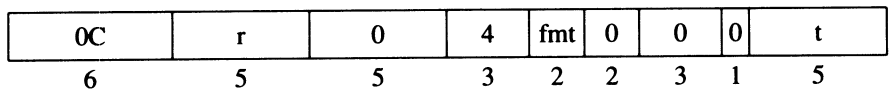
Description: The floating-point registers or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically divided. The result is calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register or register pair specified by *t*.

Operation: FPR[t] ← FPR[r1] / FPR[r2];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT SQUARE ROOT — FSQRT

Format: FSQRT,fmt r,t



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.

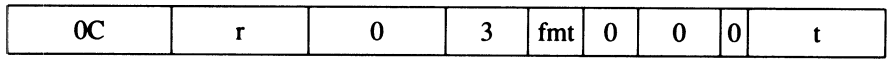
Description: The floating-point register or register pair specified by *r* is interpreted in the specified format and the positive arithmetic square root is taken. The result is calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. If the value of *r* corresponds to –0, the result will be –0. The result is placed in the floating-point register or register pair specified by *t*.

Operation: FPR[t] ← square_root(FPR[r]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT ABSOLUTE VALUE — FABS

Format: FABS,fmt r,t



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 floating-point register or register pair specified by *r* is moved to the floating-point register or register pair specified by *t* with the sign bit cleared. This instruction is non-arithmetic and does not cause an invalid operation exception when a NaN is copied.

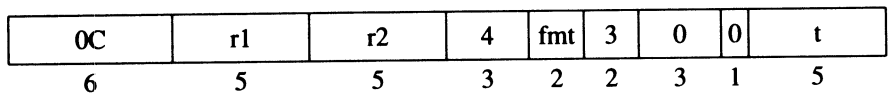
Operation:
```
FPR[t]{all_bits_except_sign} ← FPR[r]{all_bits_except_sign};
FPR[t]{sign_bit} ← 0;
```

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT REMAINDER

FREM

Format: FREM,fmt r1,r2,t



0C	r1	r2	4	fmt	3	0	0	t
6	5	5	3	2	2	3	1	5

Purpose: To perform a floating-point division, returning the floating-point remainder.

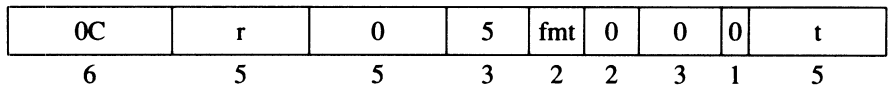
Description: The floating-point registers or register pairs specified by *r1* and *r2* are interpreted in the specified format and arithmetically *remaindered.* The result is placed in the floating-point register or register pair specified by *t* and is defined, regardless of the rounding mode, by the mathematical relation: $r = FPR[r1] - (FPR[r2] * n)$ where n is the integer nearest the exact value $\frac{FPR[r1]}{FPR[r2]}$; if $\left| n - \frac{FPR[r1]}{FPR[r2]} \right| = \frac{1}{2}$, then n is the nearest even integer. The remainder is always exact. If r is zero, its sign is that of $FPR[r1]$.

Operation: FPR[t] ← floating_point_remainder(FPR[r1], FPR[r2]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT ROUND TO INTEGER — FRND

Format: FRND,fmt r,t



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 floating-point register or 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 floating-point register or register pair specified by *t*. An inexact exception is signaled when the result and source are not the same.

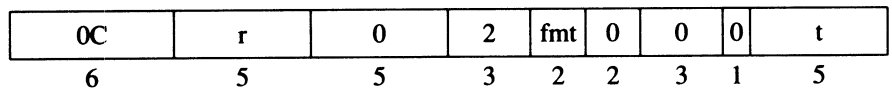
Operation: FPR[t] ← floating_point_round(FPR[r]);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT COPY

FCPY

Format: FCPY,fmt r,t



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 floating-point register or register pair specified by *r* is interpreted in the specified format and is copied into the floating-point register or register pair specified by *t*. This operation is non-arithmetic and does not cause an invalid operation exception when a NaN is copied.

Operation: FPR[t] ← FPR[r];

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT CONVERT FROM FLOATING-POINT TO FLOATING-POINT

FCNVFF

Format: FCNVFF,sf,df r,t

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 floating-point register or 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 floating-point register or register pair specified by *t*.

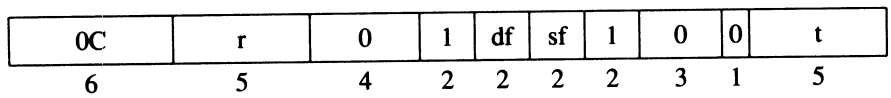
Rounding occurs according to the currently specified rounding mode.

Operation: FPR[t] ← convert_float_to_float(FPR[r],sf,df);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT CONVERT FROM FIXED-POINT TO FLOATING-POINT

Format: FCNVXF,sf,df r,t



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 floating-point register or register pair specified by *r* is interpreted as a fixed-point value in the specified source format, *sf*, and arithmetically converted to the specified destination format, *df*, as a floating-point value. The result is placed in the floating-point register or register pair specified by *t*.

Rounding occurs according to the currently specified rounding mode.

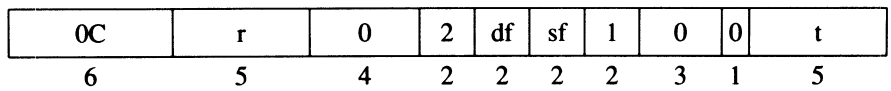
Operation: FPR[t] ← convert_fixed_to_float(FPR[r],sf,df);

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT

FCNVFX

Format: FCNVFX,sf,df r,t



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 floating-point register or register pair specified by *r* is interpreted as a floating-point number in the specified source format, *sf*, and arithmetically converted to a fixed-point number in the specified destination format, *df*. The result is placed in the floating-point register or register pair specified by *t*.

Rounding occurs according to the currently specified rounding mode.

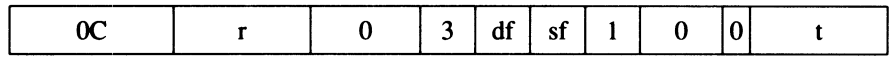
Operation: FPR[t] ← convert_float_to_fixed(FPR[r],sf,df);

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



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 floating-point register or register pair specified by *r* is interpreted as a floating-point number in the specified source format, *sf*, and arithmetically converted to a fixed-point number in the specified destination format, *df*. The result is placed in the floating-point register or register pair specified by *t*.

The current rounding mode is ignored and the result is rounded toward zero.

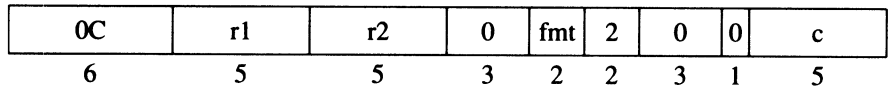
Operation:

```
temp_mode ← FPR[0]{rounding_mode_bits};
current_rounding_mode ← round_zero;
FPR[t] ← convert_float_to_fixed(FPR[r],sf,df);
FPR[0]{rounding_mode_bits} ← temp_mode;
```

Exceptions: Assist emulation trap
Assist exception trap

FLOATING-POINT COMPARE — FCMP

Format: FCMP,fmt,cond r1,r2



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 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. If the result is true, the C-bit in the floating-point status register is set. If one of the values is a NaN, and the low-order bit of the condition is set, an invalid operation exception is signaled. When an invalid exception is signaled with a trap, the state of the C-bit is not defined and may contain any value. A signaling NaN, as usual, signals an invalid exception. The delivered C-bit for untrapped invalid exceptions is the AND of the unordered relation (true) and bit 3 of the condition 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.

Operation:

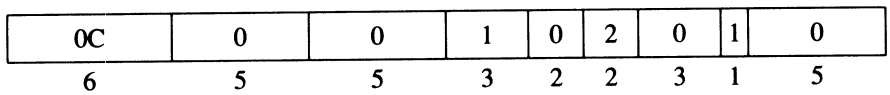
```
if (NaN(FPR[r1]) || NaN(FPR[r2]))
    if (c{4})
        invalid_operation_exception;
    else
        {
        greater_than ← false; less_than ← false;
        equal_to ← false; unordered ← true;
        }
else
    {
    greater_than ← FPR[r1] > FPR[r2]; less_than ← FPR[r1] < FPR[r2];
    equal_to ← FPR[r1] = FPR[r2]; unordered ← false;
    }
FPR[0]{C-bit} ← (((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



0C	0	0	1	0	2	0	1	0
6	5	5	3	2	2	3	1	5

Purpose: To test the result of an earlier comparison.

Description: The C-bit in the floating-point status register is tested. If set, then the following instruction is nullified. No exception traps are possible for this instruction.

Operation:
```
if (FPR[0]{C-bit})
    PSW[N] ← 1;
```

Exceptions: Assist emulation trap

Notes: FTEST is the only instruction that is defined with the nullify bit set. No reserved operation can be defined to use the nullify bit. Implementations may decode the FTEST instruction using only the nullify bit.

A

Glossary

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 memory 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

HP Precision Architecture is a byte-addressable system. The byte address can be split into two parts: the high-order bits which are the space identification and the 32 low-order bits that give the space offset. Doublewords, words, and halfwords are always stored on aligned addresses. An address may be either virtual or absolute.

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. Aliasing is permitted by the HP Precision Architecture only when a virtual and an absolute address which access the same physical memory location, are equivalently-mapped. A virtual and an absolute address are equivalently-mapped when the virtual address has a space identifier equal to zero and a virtual offset equal to its absolute address.

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 Precision system that performs arithmetic and logic operations on its inputs and generates an output and status.

Assist Processor

Processors which may be added to the basic HP Precision Architecture 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).

Attached Processor

A type of assist processor which interfaces at the level of main memory, generates its own bus addresses, and typically has its own registers and local storage. Examples are array processors and I/O processors.

B-bit (Taken Branch in Previous Cycle)

A bit in the PSW that is set if the previous instruction was a taken branch.

Base Register (Base Address Register)

A register that holds the numeric value that is used as a base value in the calculation of virtual 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.

Byte

A group of eight contiguous bits which is the smallest addressable unit on a HP Precision Architecture system.

C-bit (Code Address Translation Enable)

The C-bit in the PSW 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(s) 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 Miss

A cache miss occurs when the cache does not contain a copy of the physical memory address being requested by the virtual address. The cache is updated with the correct 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) of the result of the previous arithmetic operation.

Central Processing Unit

The part of a HP Precision Architecture system that fetches and executes instructions. The central processor conceptually contains an Execution Unit and a Control Unit.

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.

Compatibility

The ability for software developed for one machine type to execute on another machine type. The HP Precision Architecture 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 an entity or a relationship between entities used in determining whether an instruction is to branch, nullify, or trap.

Control Register

A register which contains system state information used for memory access protection, interruption control, and processor state control. A HP Precision Architecture system 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)

Control register 10, 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 means that the corresponding coprocessor is present and operational. If a bit is zero, an attempt to reference the corresponding coprocessor causes an assist emulation trap.

Current Instruction

The instruction whose address is in the front entry of the instruction address queues (IASQ and IAOQ).

D-bit (Data Address Translation Enable)

A bit in the PSW that controls data address translation. If the D-bit is reset (D-bit=0), the TLB is turned off and only physical or absolute addressing is used.

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.

Displacement

The amount that is added to a base register to form an offset in the virtual address computation.

DTLB

A separate TLB which does address translation only for data.

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.

Effective Memory Address

The address derived by applying specific address building rules by the current instruction that is used to identify the current operand.

Equivalent Map

A condition of a virtual address with a space identifier equal to zero and a virtual offset equal to its absolute address.

External Interrupt Enable Mask Register (EIEM)

Control Register 15 which is a 32-bit register containing one bit for each external interrupt class. When set to 0, the bit masks pending external interrupts assigned to that class.

External Interrupt Request Register (EIR)

Control Register 23 which is a 32-bit register containing one bit for each external interrupt. When set to 1, each bit designates that an interruption is pending for the external interrupt designated by that bit position.

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.

Following Instruction

The instruction whose address is in the back entry 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.

General Register

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 and store data to memory and hold operands and results from the ALU. RISC architectures are very register-intensive.

H-bit (Higher Privilege Transfer Trap Enable)

The bit in the PSW that enables an interruption whenever the next instruction will execute at a higher privilege level.

Hash Algorithm

A transformation on a set of bits used to produce a uniformly distributed many-to-one function.

Hash Table

A table used in software virtual address translation to maintain correspondence between hash values and entries in the PDIR.

HPMC (High-Priority Machine Check)

An interruption which occurs when an unrecoverable hardware fault has been detected.

I-bit (External, Power Failure, and LPMC Interruption Enable)

A bit in the PSW used as an enable for external interrupts, power failure interrupts, and low-priority machine check interrupts.

IAOQ (Instruction Address Offset Queue)

A two-entry queue of 32-bit registers that is used to hold the Instruction Address Offset (IA OFFSET). The first entry is the IAOQ-Front and holds the IA Offset of the current instruction. The other entry is the 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-entry queue of 16- or 32-bit registers that is used to hold the Instruction Address Space (IA SPACE). The first entry is the IASQ-Front and holds the IA Space of the current instruction. The other entry is the IASQ-Back and holds the IA Space of the following instruction.

IIAOQ (Interruption Instruction Address Offset Queue)

A two-entry 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-entry queue of 16- or 32-bit registers that is used to save the Instruction Address Space for use in processing interruptions.

IIR (Interruption Instruction Register)

Control Register 19 which is used by the hardware to store the instruction that causes the interruption or the instruction that was in progress at the time the interruption occurred.

Instruction Cache (I-cache)

The instruction 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.

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.

Interrupt Parameter Registers (IPR)

Three registers (IIR, ISR, and IOR - Control Registers 19, 20, and 21) set by the hardware, when an interruption occurs, to pass the interrupted instruction and it's virtual address to an interruption handler.

Interruption

An event that changes the instruction stream to handle exceptional conditions including traps, checks, faults, and interrupts.

Interruption Vector Address (IVA)

Control Register 14 which 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 compared to 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.

IOR (Interruption Offset Register)

Control Register 21 which receives a copy of the IA offset at the time of the interruption.

IPSW (Interruption Processor Status Word)

Control Register 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.

ISR (Interruption Space Register)

Control Register 20 which receives a copy of the IA Space at the time of the interruption.

ITLB

A separate TLB which does address translation only for instructions.

L-bit (Lower Privilege Transfer Trap Enable)

A bit in the PSW that is used for enabling lower-privilege transfer traps. This bit, when set, causes a trap when the instruction which is about to execute is at a lower privilege level than the instruction which executed immediately prior to it.

Long Pointer

A virtual pointer which is made up of a space identifier and a 32-bit byte offset within the virtual space.

M-bit (High Priority Machine Check Disable)

The bit in the PSW that disables the recognition of a HPMC.

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 addresses in the range X'00000000 through X'EFFFFFFF.

Memory-mapped I/O

Control of input and output through load and store instructions to particular virtual or physical addresses.

Multiprocessor

A computer with multiple processors provide fault tolerance.

NaN

A term used in describing floating-point operation which refers to any value which is "not a number".

Nullify

This term is associated with the execution of instructions. To "nullify" an instruction is equivalent to skipping the execution of that instruction.

P-bit (Protection ID Validation Enable)

A bit in the PSW that is used as a protection identifier validation enable bit. If the P-bit is set, the protection IDs 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 2048 bytes (2K bytes).

PDIR (Physical Page Directory)

A table which is used with the Hash Table to perform virtual address translations. The PDIR stores virtual address translations that either the TLB miss software or hardware will load into the TLB.

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.

Privilege Level

HP Precision Architecture's access control mechanisms are based on 4 privilege levels numbered from 0 to 3, with 0 the most-privileged. The current privilege level is maintained in the front Instruction Address (IA) queue element.

Protection Identifiers

Control Registers 8, 9, 12, and 13 contain 16-bit strings which designate up to four groups of pages accessible to the currently executing process.

PSW (Processor Status Word)

A 32-bit register which contains information about the processor state.

System Mask

The R, Q, P, D, and I bits of the PSW are known as the System Mask. They may be set, reset, and read by the Set System Mask (SSM), Reset System Mask (RSM) and Move to System Mask (MSM) instructions or through interruptions
and the RFI instruction.

Q-bit (Interruption State Collection Enable)

The bit in the PSW that, when set on, enables collection of the machine state at the instant of interruption (IPSW, IIR, ISR, IOR).

R-bit (Recovery Counter Enable)

A bit in the PSW that enables a recovery counter trap.

Recovery Counter

Control Register 0 acts as the Recovery Counter and counts down by 1 during execution of each non-nullified instruction.

SAR (Shift Amount Register)

This is Control Register 11 and is used by the variable shift, extract, deposit, and branch on bit instructions. It specifies the number of bits or the ending bit position that a quantity is to be shifted, extracted or deposited.

SFU (Special Function Unit)

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.

Short Pointer

A 32-bit pointer used in virtual addressing. The two high-order bits point to one of four virtual address spaces and the remaining 30 bits point to the virtual byte address.

Space Identifier

A 16-bit or 32-bit pointer which occupies the upper portion of a virtual address and specifies the virtual space portion of the virtual address.

Space Register

A register used to specify the space identifier for virtual addressing.

Static Displacement

If the displacement is a fixed value that is known at compile time, it is called static.

T-bit (Taken Branch Trap Enable)

A bit in the PSW that enables the taken branch trap.

Taken Branch

A conditional branch is considered to be "taken" if the specified condition is met.

TLB (Translation Lookaside Buffer)

A hardware table 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 21-bit absolute page number (contained in the entry) is concatenated with the 11-bit page offset from the original virtual address to form a 32-bit address. If the entry does not exist, software updates the TLB via a fault or, in other implementations, with special hardware.

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.

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 Disable (WD) Bit

The low-order bit of each of the four protection identifiers (PIs) that, when set to 1, disables writing for all privilege levels for the pages so protected.

X-bit (Data Memory Break Disable)

A bit in the PSW that disables the data memory break trap if set. A data memory break trap happens if a write is performed to a page whose TLB B-bit is off.

B

Instruction Index

ADD (ADD) 5-82
ADD AND BRANCH IF FALSE (ADDBF) 5-75
ADD AND BRANCH IF TRUE (ADDBT) 5-74
ADD AND TRAP ON OVERFLOW (ADDO) 5-84
ADD IMMEDIATE AND BRANCH IF FALSE (ADDIBF) 5-77
ADD IMMEDIATE AND BRANCH IF TRUE (ADDIBT) 5-76
ADD IMMEDIATE LEFT (ADDIL) 5-55
ADD LOGICAL (ADDL) 5-83
ADD TO IMMEDIATE (ADDI) 5-114
ADD TO IMMEDIATE AND TRAP ON CONDITION (ADDIT) 5-116
ADD TO IMMEDIATE AND TRAP ON CONDITION OR OVERFLOW (ADDITO) 5-117
ADD TO IMMEDIATE AND TRAP ON OVERFLOW (ADDIO) 5-115
ADD WITH CARRY (ADDC) 5-85
ADD WITH CARRY AND TRAP ON OVERFLOW (ADDCO) 5-86
AND (AND) 5-106
AND COMPLEMENT (ANDCM) 5-107
BRANCH AND LINK (BL) 5-60
BRANCH AND LINK EXTERNAL (BLE) 5-66
BRANCH AND LINK REGISTER (BLR) 5-63
BRANCH EXTERNAL (BE) 5-65
BRANCH ON BIT (BB) 5-79
BRANCH ON VARIABLE BIT (BVB) 5-78
BRANCH VECTORED (BV) 5-64
BREAK (BREAK) 5-139
COMPARE AND BRANCH IF FALSE (COMBF) 5-71
COMPARE AND BRANCH IF TRUE (COMBT) 5-70
COMPARE AND CLEAR (COMCLR) 5-103
COMPARE IMMEDIATE AND BRANCH IF FALSE (COMIBF) 5-73
COMPARE IMMEDIATE AND BRANCH IF TRUE (COMIBT) 5-72
COMPARE IMMEDIATE AND CLEAR (COMICLR) 5-120
COPROCESSOR LOAD DOUBLEWORD INDEXED (CLDDX) 5-181
COPROCESSOR LOAD DOUBLEWORD SHORT (CLDDS) 5-185
COPROCESSOR LOAD WORD INDEXED (CLDWX) 5-180
COPROCESSOR LOAD WORD SHORT (CLDWS) 5-184
COPROCESSOR OPERATION (COPR) 5-179
COPROCESSOR STORE DOUBLEWORD INDEXED (CSTDX) 5-183
COPROCESSOR STORE DOUBLEWORD SHORT (CSTDS) 5-187
COPROCESSOR STORE WORD INDEXED (CSTWX) 5-182
COPROCESSOR STORE WORD SHORT (CSTWS) 5-186
DECIMAL CORRECT (DCOR) 5-111
DEPOSIT (DEP) 5-129

DEPOSIT IMMEDIATE (DEPI) 5-131
DIAGNOSE (DIAG) 5-171
DIVIDE STEP (DS) 5-102
EXCLUSIVE OR (XOR) 5-105
EXTRACT SIGNED (EXTRS) 5-127
EXTRACT UNSIGNED (EXTRU) 5-126
FLOATING-POINT ABSOLUTE VALUE (FABS) 6-35
FLOATING-POINT ADD (FADD) 6-30
FLOATING-POINT COMPARE (FCMP) 6-43
FLOATING-POINT CONVERT FROM FIXED-POINT TO FLOATING-POINT (FCNVXF) 6-40
FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT (FCNVFX) 6-41
FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT AND TRUNCATE (FCNVFXT) 6-42
FLOATING-POINT CONVERT FROM FLOATING-POINT TO FLOATING-POINT (FCNVFF) 6-39
FLOATING-POINT COPY (FCPY) 6-38
FLOATING-POINT DIVIDE (FDIV) 6-33
FLOATING-POINT LOAD DOUBLEWORD INDEXED (FLDDX) 6-23
FLOATING-POINT LOAD DOUBLEWORD SHORT (FLDDS) 6-27
FLOATING-POINT LOAD WORD INDEXED (FLDWX) 6-22
FLOATING-POINT LOAD WORD SHORT (FLDWS) 6-26
FLOATING-POINT MULTIPLY (FMPY) 6-32
FLOATING-POINT REMAINDER (FREM) 6-36
FLOATING-POINT ROUND TO INTEGER (FRND) 6-37
FLOATING-POINT SQUARE ROOT (FSQRT) 6-34
FLOATING-POINT STORE DOUBLEWORD INDEXED (FSTDX) 6-25
FLOATING-POINT STORE DOUBLEWORD SHORT (FSTDS) 6-29
FLOATING-POINT STORE WORD INDEXED (FSTWX) 6-24
FLOATING-POINT STORE WORD SHORT (FSTWS) 6-28
FLOATING-POINT SUBTRACT (FSUB) 6-31
FLOATING-POINT TEST (FTEST) 6-44
FLUSH DATA CACHE (FDC) 5-167
FLUSH DATA CACHE ENTRY (FDCE) 5-169
FLUSH INSTRUCTION CACHE (FIC) 5-168
FLUSH INSTRUCTION CACHE ENTRY (FICE) 5-170
GATEWAY (GATE) 5-61
INCLUSIVE OR (OR) 5-104
INSERT DATA TLB ADDRESS (IDTLBA) 5-162
INSERT DATA TLB PROTECTION (IDTLBP) 5-164
INSERT INSTRUCTION TLB ADDRESS (IITLBA) 5-163
INSERT INSTRUCTION TLB PROTECTION (IITLBP) 5-165
INTERMEDIATE DECIMAL CORRECT (IDCOR) 5-113
LOAD AND CLEAR WORD INDEXED (LDCWX) 5-38
LOAD AND CLEAR WORD SHORT (LDCWS) 5-44
LOAD BYTE (LDB) 5-28
LOAD BYTE INDEXED (LDBX) 5-36
LOAD BYTE SHORT (LDBS) 5-42
LOAD HALFWORD (LDH) 5-27
LOAD HALFWORD INDEXED (LDHX) 5-35
LOAD HALFWORD SHORT (LDHS) 5-41

LOAD HASH ADDRESS (LHA) 5-157
LOAD IMMEDIATE LEFT (LDIL) 5-54
LOAD OFFSET (LDO) 5-53
LOAD PHYSICAL ADDRESS (LPA) 5-155
LOAD SPACE IDENTIFIER (LDSID) 5-144
LOAD WORD (LDW) 5-26
LOAD WORD ABSOLUTE INDEXED (LDWAX) 5-37
LOAD WORD ABSOLUTE SHORT (LDWAS) 5-43
LOAD WORD AND MODIFY (LDWM) 5-32
LOAD WORD INDEXED (LDWX) 5-34
LOAD WORD SHORT (LDWS) 5-40
MOVE AND BRANCH (MOVB) 5-67
MOVE FROM CONTROL REGISTER (MFCTL) 5-149
MOVE FROM SPACE REGISTER (MFSP) 5-148
MOVE IMMEDIATE AND BRANCH (MOVIB) 5-68
MOVE TO CONTROL REGISTER (MTCTL) 5-146
MOVE TO SPACE REGISTER (MTSP) 5-145
MOVE TO SYSTEM MASK (MTSM) 5-143
PROBE READ ACCESS (PROBER) 5-151
PROBE READ ACCESS IMMEDIATE (PROBERI) 5-152
PROBE WRITE ACCESS (PROBEW) 5-153
PROBE WRITE ACCESS IMMEDIATE (PROBEWI) 5-154
PURGE DATA CACHE (PDC) 5-166
PURGE DATA TLB (PDTLB) 5-158
PURGE DATA TLB ENTRY (PDTLBE) 5-160
PURGE INSTRUCTION TLB (PITLB) 5-159
PURGE INSTRUCTION TLB ENTRY (PITLBE) 5-161
RESET SYSTEM MASK (RSM) 5-142
RETURN FROM INTERRUPTION (RFI) 5-140
SET SYSTEM MASK (SSM) 5-141
SHIFT DOUBLE (SHD) 5-123
SHIFT ONE AND ADD (SH1ADD) 5-87
SHIFT ONE AND ADD LOGICAL (SH1ADDL) 5-88
SHIFT ONE, ADD AND TRAP ON OVERFLOW (SH1ADDO) 5-89
SHIFT THREE AND ADD (SH3ADD) 5-93
SHIFT THREE AND ADD LOGICAL (SH3ADDL) 5-94
SHIFT THREE, ADD AND TRAP ON OVERFLOW (SH3ADDO) 5-95
SHIFT TWO AND ADD (SH2ADD) 5-90
SHIFT TWO AND ADD LOGICAL (SH2ADDL) 5-91
SHIFT TWO, ADD AND TRAP ON OVERFLOW (SH2ADDO) 5-92
SPECIAL OPERATION ONE (SPOP1) 5-176
SPECIAL OPERATION THREE (SPOP3) 5-178
SPECIAL OPERATION TWO (SPOP2) 5-177
SPECIAL OPERATION ZERO (SPOP0) 5-175
STORE BYTE (STB) 5-31
STORE BYTE SHORT (STBS) 5-48
STORE BYTES SHORT (STBYS) 5-50
STORE HALFWORD (STH) 5-30

STORE HALFWORD SHORT (STHS) 5-47
STORE WORD (STW) 5-29
STORE WORD ABSOLUTE SHORT (STWAS) 5-49
STORE WORD AND MODIFY (STWM) 5-33
STORE WORD SHORT (STWS) 5-46
SUBTRACT (SUB) 5-96
SUBTRACT AND TRAP ON CONDITION (SUBT) 5-100
SUBTRACT AND TRAP ON CONDITION OR OVERFLOW (SUBTO) 5-101
SUBTRACT AND TRAP ON OVERFLOW (SUBO) 5-97
SUBTRACT FROM IMMEDIATE (SUBI) 5-118
SUBTRACT FROM IMMEDIATE AND TRAP ON OVERFLOW (SUBIO) 5-119
SUBTRACT WITH BORROW (SUBB) 5-98
SUBTRACT WITH BORROW AND TRAP ON OVERFLOW (SUBBO) 5-99
SYNCHRONIZE CACHES (SYNC) 5-150
UNIT ADD COMPLEMENT (UADDCM) 5-109
UNIT ADD COMPLEMENT AND TRAP ON CONDITION (UADDCMT) 5-110
UNIT XOR (UXOR) 5-108
VARIABLE DEPOSIT (VDEP) 5-128
VARIABLE DEPOSIT IMMEDIATE (VDEPI) 5-130
VARIABLE EXTRACT SIGNED (VEXTRS) 5-125
VARIABLE EXTRACT UNSIGNED (VEXTRU) 5-124
VARIABLE SHIFT DOUBLE (VSHD) 5-121
ZERO AND DEPOSIT (ZDEP) 5-134
ZERO AND DEPOSIT IMMEDIATE (ZDEPI) 5-136
ZERO AND VARIABLE DEPOSIT (ZVDEP) 5-132
ZERO AND VARIABLE DEPOSIT IMMEDIATE (ZVDEPI) 5-135

C

Instruction Formats

The HP Precision Architecture instruction formats are shown below. The most general form of each format is given. Individual instructions in each class may have zeroes 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	0	ext4	m	t
6	5	5	2	1	1	2	4	1	5

3. Short Displacement Loads

op	b	im5	s	a	1	0	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	0	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	ext7	t
6	5	5	3	1	7	5

7. Arithmetic Immediate

op	r	t	c	f	e	im11
6	5	5	3	1	1	11

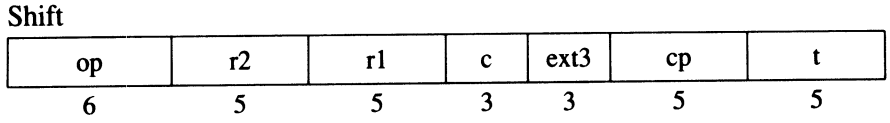
8. Extract

op	r	t	c	ext3	p	clen
6	5	5	3	3	5	5

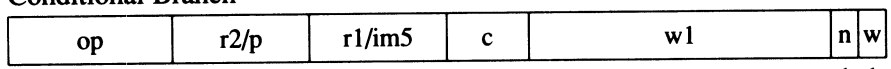
9. Deposit

op	t	r/im5	c	ext3	cp	clen
6	5	5	3	3	5	5

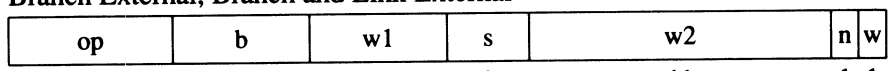
10. Shift



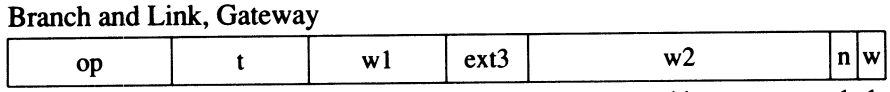
11. Conditional Branch



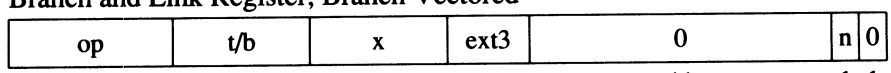
12. Branch External, Branch and Link External



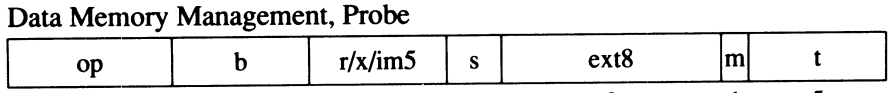
13. Branch and Link, Gateway



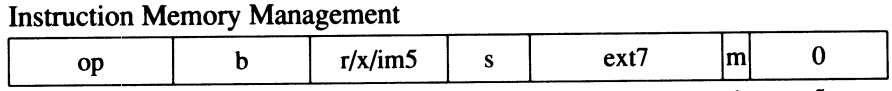
14. Branch and Link Register, Branch Vectored



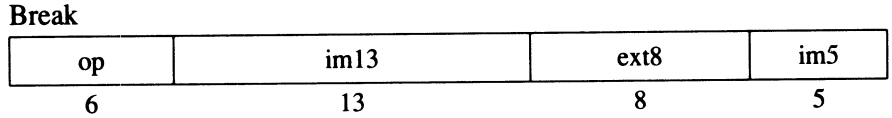
15. Data Memory Management, Probe



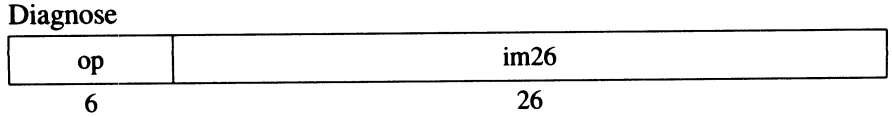
16. Instruction Memory Management



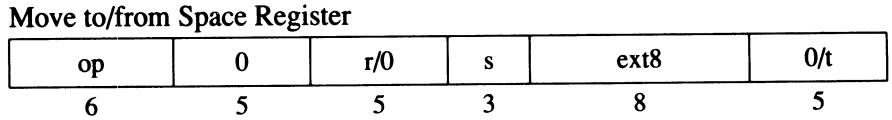
17. Break



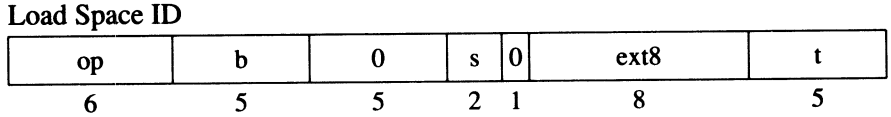
18. Diagnose



19. Move to/from Space Register



20. Load Space ID



21. Move to Control Register

op	t	r	0	ext8	0
6	5	5	3	8	5

22. Move from Control Register

op	r	0	0	ext8	t
6	5	5	3	8	5

23. System Control

op	0/b	0/r/im5	0	ext8	0/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	0	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

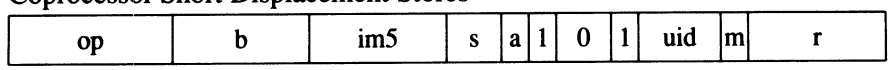
30. Coprocessor Indexed Stores

op	b	x	s	u	0	0	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

31. Coprocessor Short Displacement Loads

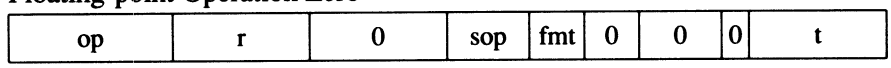
op	b	im5	s	a	1	0	0	uid	m	t
6	5	5	2	1	1	2	1	3	1	5

32. Coprocessor Short Displacement Stores



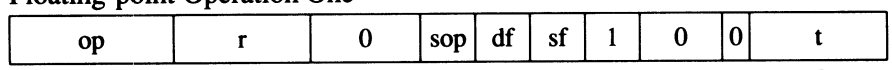
op	b	im5	s	a	1	0	1	uid	m	r
6	5	5	2	1	1	2	1	3	1	5

33. Floating-point Operation Zero



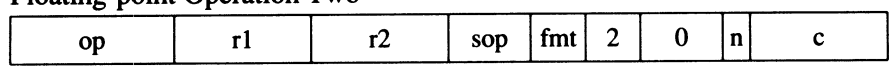
op	r	0	sop	fmt	0	0	0	t
6	5	5	3	2	2	3	1	5

34. Floating-point Operation One



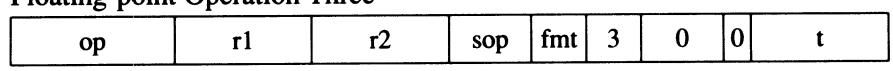
op	r	0	sop	df	sf	1	0	0	t
6	5	4	2	2	2	2	3	1	5

35. Floating-point Operation Two



op	r1	r2	sop	fmt	2	0	n	c
6	5	5	3	2	2	3	1	5

36. Floating-point Operation Three



op	r1	r2	sop	fmt	3	0	0	t
6	5	5	3	2	2	3	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
clen	31 - extract/deposit length
cp	31 - bit position
df	floating-point destination format
e or ext	operation code extension
f	condition negation bit
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
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	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-0. Instructions are shown in uppercase. Instruction classes are capitalized.

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 (Table D-3)	LDH	COMIBT	BB
2	Arith/log (Table D-4)	LDW	COMBF	MOVB
3	Index_Mem (Table D-5)	LDWM	COMIBF	MOVIB
4	SPOPn	—	COMICLR	Extract (Table D-7)
5	DIAG	—	Subi (Table D-6)	Deposit (Table D-7)
6	—	—	—	—
7	—	—	—	—
8	LDIL	STB	ADDBT	BE
9	Copr_w (Table D-9)	STH	ADDIBT	BLE
A	ADDIL	STW	ADDBF	Branch (Table D-8)
B	Copr_dw (Table D-9)	STWM	ADDIBF	—
C	COPR	—	Addit (Table D-6)	—
D	LDO	—	Addi (Table D-6)	—
E	—	—	—	—
F	—	—	—	—

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. In the following discussions of opcode extensions, the name shown in parentheses in each heading is the major opcode class name.

System Control Instructions (System-op)

Figure D-0 shows the format of the major opcode instructions. The opcode extensions for the system control instructions (major opcode 00) are listed in Table D-2. Bits 19:21 encode the source of the operation and bits 24:26 encode the destination.

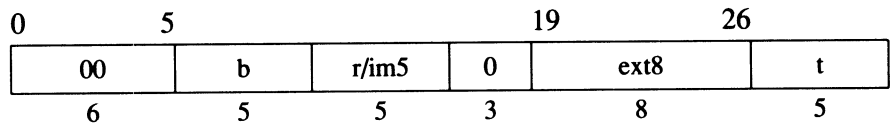


Figure D-1. Format for System Control Instructions.

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
BREAK	00	000	00	000	00
RFI	00	011	00	000	60
SSM	00	011	01	011	6B
RSM	00	011	10	011	73
MTSM	00	110	00	011	C3
LDSID	00	100	00	101	85
MTSP	00	110	00	001	C1
MTCTL	00	110	00	010	C2
MFSP	00	001	00	101	25
MFCTL	00	010	00	101	45
SYNC	00	001	00	000	20

Memory Management Instructions (Mem_Mgmt)

Figure D-2 shows the format of the memory management instructions. The opcode extensions (bits 18:26) for memory management instructions (major opcode 01) are listed in Table D-3. This group includes instructions that access the translation lookaside buffers and the caches.

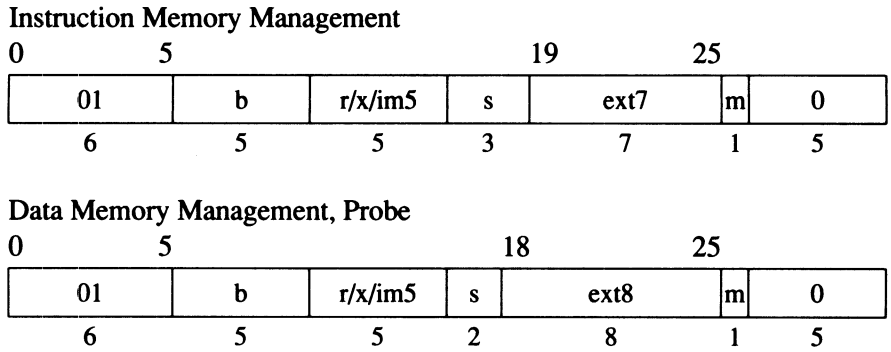


Figure D-2. Format for Memory Management Instructions.

Table D-3. Memory Management Instructions.

Instruction	Opcode	Extension						
	hex	binary						hex
	bits 0:5	bit 18	bit 19	bits 20:21	bits 22:24	bit 25	bit 26	bits 18:25
IITLBP	01	-	0	00	000	0	0	00
IITLBA	01	-	0	00	000	1	0	01
PITLB	01	-	0	00	100	0	m	08
PITLBE	01	-	0	00	100	1	m	09
FIC	01	-	0	00	101	0	m	0A
FICE	01	-	0	00	101	1	m	0B
IDTLBA	01	0	1	00	000	1	0	41
IDTLBP	01	0	1	00	000	0	0	40
PROBER	01	0	1	00	011	0	0	46
PROBEW	01	0	1	00	011	1	0	47
PROBERI	01	1	1	00	011	0	0	C6
PROBEWI	01	1	1	00	011	1	0	C7
PDTLB	01	0	1	00	100	0	m	48
PDTLBE	01	0	1	00	100	1	m	49
FDC	01	0	1	00	101	0	m	4A
FDCE	01	0	1	00	101	1	m	4B
LHA	01	0	1	00	110	0	m	4C
LPA	01	0	1	00	110	1	m	4D
PDC	01	0	1	00	111	0	m	4E

Bits	Description
18	- indicates this bit is part of another field. 0 indicates a non-immediate value. 1 indicates an immediate value.
19	0 indicates an instruction memory management. 1 indicates a data memory management.
22:24	000 encodes insert instructions. 011 encodes probe instructions. 100 or 111 encodes purge instructions. 101 encodes flush instructions. 110 encodes load instructions.
22	1 encodes a modify (bit 26) enable.
23	1 indicates a stored result.
24	1 encodes nonprivileged instructions.
26	*m* indicates 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-4. Bit 26 is always zero, so rows with bit 26 equal to one are not included in the table.

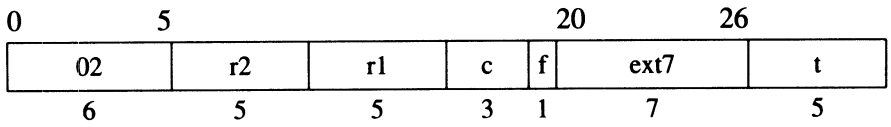


Figure D-3. Format for Arithmetic/Logical Instructions.

Table D-4. Arithmetic/Logical Instructions.

Instruction	Opcode	Extension		
	hex	binary		hex
	bits 0:5	bits 20:21	bits 22:26	bits 20:26
ANDCM	02	00	00000	00
SUB	02	01	00000	20
SUBO	02	11	00000	60
DS	02	01	00010	22
COMCLR	02	10	00100	44
SUBT	02	01	01100	26
SUBTO	02	11	01100	66
SUBB	02	01	01000	28
SUBBO	02	11	01000	68
UADDCM	02	10	01100	4C
UADDCMT	02	10	01110	4E
AND	02	00	10000	10
ADD	02	01	10000	30
ADDL	02	10	10000	50
ADDO	02	11	10000	70
OR	02	00	10010	12
SH1ADD	02	01	10010	32
SH1ADDL	02	10	10010	52
SH1ADDO	02	11	10010	72
XOR	02	00	10100	14
SH2ADD	02	01	10100	34
SH2ADDL	02	10	10100	54
SH2ADDO	02	11	10100	74
SH3ADD	02	01	10110	36
SH3ADDL	02	10	10110	56
SH3ADDO	02	11	10110	76
ADDC	02	01	11000	38
ADDCO	02	11	11000	78
UXOR	02	00	11100	1C
DCOR	02	10	11100	5C
IDCOR	02	10	11110	5E

Bits	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 condition.

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-5. The short displacement forms are distinguished from the indexed instructions by bit 19 (0=indexed, 1=short).

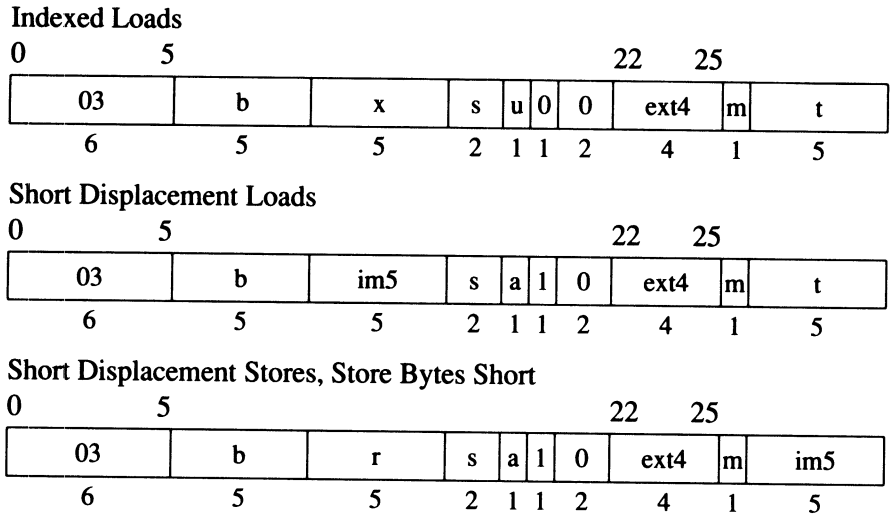


Figure D-4. Formats for Indexed and Short Displacement Load/Store Instructions.

Table D-5. 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
LDBS	03	1	00	00	00
LDBX	03	0	00	00	00
STBS	03	1	10	00	08
STBYS	03	1	11	00	0C
LDHS	03	1	00	01	01
LDHX	03	0	00	01	01
STHS	03	1	10	01	09
LDWS	03	1	00	10	02
LDWX	03	0	00	10	02
LDWAS	03	1	01	10	06
LDWAX	03	0	01	10	06
STWS	03	1	10	10	0A
STWAS	03	1	11	10	0E
LDCWS	03	1	01	11	07
LDCWX	03	0	01	11	07

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 are listed in Table D-6. The extension field, *e*, determines whether or not the instruction traps on overflow.

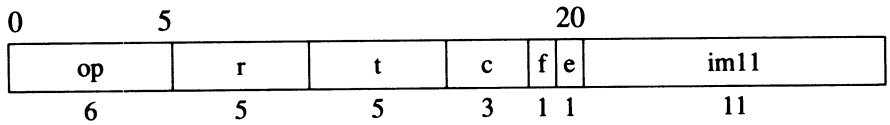


Figure D-5. Format for Arithmetic Immediate Instructions.

Table D-6. 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/Deposit/Shift Instructions (Extract and Deposit)

Figure D-6 shows the formats of the extract, deposit, and shift instructions. The opcode extensions (bits 19:21) for the extract, deposit, and shift instructions (major opcodes 34 and 35) are listed in Table D-7.

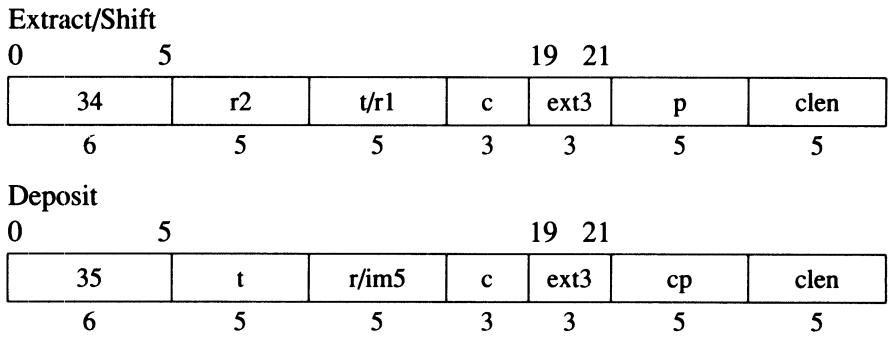


Figure D-6. Formats for Extract and Deposit Instructions.

Table D-7. Extract, Deposit, and Shift Instructions.

Instruction	Opcode	Extension	
	hex	binary	hex
	bits 0:5	bits 19:21	bits 19:21
VSHD	34	000	0
ZVDEP	35	000	0
VDEP	35	001	1
SHD	34	010	2
ZDEP	35	010	2
DEP	35	011	3
VEXTRU	34	100	4
ZVDEPI	35	100	4
VDEPI	35	101	5
VEXTRS	34	101	5
EXTRU	34	110	6
ZDEPI	35	110	6
DEPI	35	111	7
EXTRS	34	111	7

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-8.

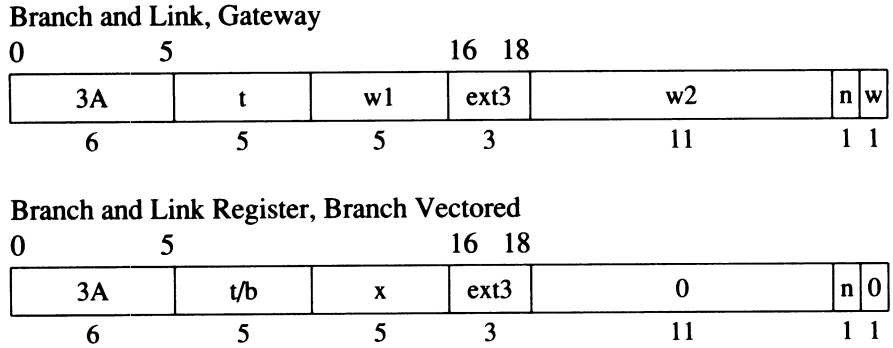


Figure D-7. Formats for Unconditional Branch Instructions.

Table D-8. Unconditional Branch Instructions.

Instruction	Opcode	Extension	
	hex	binary	hex
	bits 0:5	bits 16:18	bits 16:18
GATE	3A	001	1
BL	3A	000	0
BLR	3A	010	2
BV	3A	110	6

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-9. 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 18 (1 = short displacement) and loads from stores by bit 22 (1 = store).

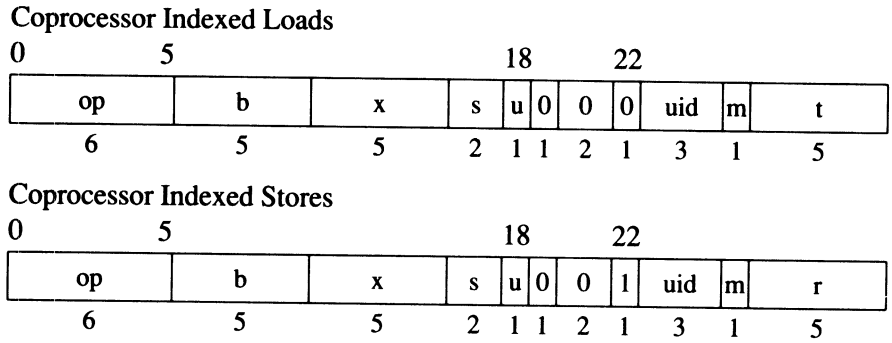


Figure D-8. Formats for Coprocessor Load/Store Instructions.

Table D-9. Coprocessor Load and Store Instructions.

Instruction	Opcode	Extension	
	hex	binary	
	bits 0:5	bit 18	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

Index

A

absolute address 2-2, 3-1
access identifier 2-11, 3-6, 3-7, A-1
access information 1-5
access rights 1-6, 2-1, 2-3, 2-8, 2-9, 2-11, 3-1, 3-6, 3-7, 4-16, 4-19, 5-62, 5-151, 5-152, 5-153, 5-154, A-1
ADD 5-5, 5-82
ADD AND BRANCH IF FALSE 5-75
ADD AND BRANCH IF TRUE 5-74
ADD AND TRAP ON OVERFLOW 5-84
add condition 5-5
ADD IMMEDIATE AND BRANCH IF FALSE 5-77
ADD IMMEDIATE AND BRANCH IF TRUE 5-76
ADD IMMEDIATE LEFT 2-6, 5-55
ADD LOGICAL 5-83
ADD TO IMMEDIATE 5-114
ADD TO IMMEDIATE AND TRAP ON CONDITION 5-116
ADD TO IMMEDIATE AND TRAP ON CONDITION OR OVERFLOW 5-117
ADD TO IMMEDIATE AND TRAP ON OVERFLOW 5-115
ADD WITH CARRY 5-85
ADD WITH CARRY AND TRAP ON OVERFLOW 5-86
ADDB 5-74, 5-75
ADDBF 5-74, 5-75
ADDBT 5-74, 5-75
ADDC 5-85
ADDCO 5-86
ADDI 5-114
ADDIB 5-76, 5-77
ADDIBF 5-76, 5-77
ADDIBT 5-76, 5-77
ADDIL 5-55
ADDIO 5-115
ADDIT 5-116
ADDITO 5-117
ADDL 5-83
ADDO 5-84
address A-1
address offset 3-1
address space 1-5, 2-2, 3-1
address translation 1-3, 1-5, 1-6, 2-3, 2-7, 3-1, 3-4, 3-6, 3-11, 4-1, A-1
addressing mode 1-2, 1-5
aliasing A-1
AND 2-14, 5-106
AND COMPLEMENT 5-107
ANDCM 5-107
architecture A-1
arithmetic/logical condition 5-2, 5-3, 5-4
ASCII 1-5, 2-4
assist emulation trap 2-12, 4-20, 5-172, 5-173, 5-175, 5-176, 5-177, 5-178, 5-179, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
assist exception trap 4-17, 5-1, 5-172, 5-173, 5-175, 5-176, 5-177, 5-178, 5-179, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
assist processor 1-1, 1-2, 1-3, 1-7, 5-172, A-2
asymmetric processor 3-14
attached module 1-6
attached processor A-2

B

B 5-60
B-bit; PSW 2-8, 4-2, 4-3, 4-7, 4-8, 4-16, 5-2, 5-61, A-2
B-bit; TLB field 3-5, 4-20, 5-16, 5-164, 5-165
base register A-2
base relative branch 4-4, 4-5, 5-56, 5-64, A-2
BB 5-79
BCD; binary coded decimal 1-5, 2-5
BE 5-65
bit ranges 5-9
bits 2-4
BL 5-60
BLE 5-66
BLR 5-63
BRANCH AND LINK 5-56, 5-60
BRANCH AND LINK EXTERNAL 2-6, 5-56, 5-58, 5-66
BRANCH AND LINK REGISTER 5-56, 5-63
BRANCH EXTERNAL 5-56, 5-65
BRANCH ON BIT 5-7, 5-79
BRANCH ON VARIABLE BIT 5-78
branch target 4-3, 4-4, 4-5, 4-7, 4-21, 5-56
BRANCH VECTORED 5-56, 5-64
BREAK 4-16, 5-139
break instruction trap 4-16, 5-139
BV 5-64
BVB 5-78

bytes 2-4, A-2

C

C-bit; PSW 2-8, 3-7, 4-1, 4-13, A-2
C/B-bits; PSW 2-9, 5-82, 5-83, 5-84, 5-85, 5-86, 5-87, 5-89, 5-90, 5-92, 5-93, 5-95, 5-96, 5-97, 5-98, 5-99, 5-100, 5-101, 5-102, 5-111, 5-113, 5-114, 5-115, 5-116, 5-117, 5-118, 5-119, A-3
cache 1-4, 1-6, 3-5, 3-14, 4-2, 5-15, A-2
cache miss A-2
CCR; coprocessor configuration register 2-12, 4-20, 5-173, A-3
check 4-9, A-3
CISC; complex instruction set computer 1-1
CLDDS 5-185
CLDDX 5-181
CLDWS 5-184
CLDWX 5-180
COMB 5-70, 5-71
COMBF 5-70, 5-71
COMBT 5-70, 5-71
COMCLR 5-103
COMIB 5-72, 5-73
COMIBF 5-72, 5-73
COMIBT 5-72, 5-73
COMICLR 5-120
COMPARE AND BRANCH IF FALSE 5-71
COMPARE AND BRANCH IF TRUE 5-70
COMPARE AND CLEAR 5-103
COMPARE IMMEDIATE AND BRANCH IF FALSE 5-73
COMPARE IMMEDIATE AND BRANCH IF TRUE 5-72
COMPARE IMMEDIATE AND CLEAR 5-120
compare/subtract condition 5-4
compatibility A-3
completer 5-2, A-3
condition A-3
condition field 5-2
conditional branch 4-3, 4-4, 4-5, 4-7, 4-21, 5-2, 5-56, 5-57
conditional trap 4-17, 5-100, 5-101, 5-110, 5-116, 5-117
control flow 1-6, 2-1, 4-1, 4-3, 4-9, 5-2
COPR 5-174, 5-179
coprocessor 1-4, 1-7, 5-172, A-3
COPROCESSOR LOAD DOUBLEWORD INDEXED 5-181
COPROCESSOR LOAD DOUBLEWORD SHORT 5-185
COPROCESSOR LOAD WORD INDEXED 5-180
COPROCESSOR LOAD WORD SHORT 5-184
COPROCESSOR OPERATION 5-179, 6-1
COPROCESSOR STORE DOUBLEWORD INDEXED 5-183
COPROCESSOR STORE DOUBLEWORD SHORT 5-187
COPROCESSOR STORE WORD INDEXED 5-182
COPROCESSOR STORE WORD SHORT 5-186
COPY 5-104
CPR; coprocessor register 2-1, 5-9, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
CPU; central processing unit 1-1, 1-3, 1-4, A-3
CR; control register 1-6, 2-10, 3-2, 3-6, 5-9, A-3
CSTDS 5-187
CSTDX 5-183
CSTWS 5-186
CSTWX 5-182
current instruction A-4
cycle time 1-1, 1-2

D

D-bit; PSW system mask 2-8, 2-9, 3-7, 5-15, 5-141, 5-142, 5-143, 5-151, 5-152, 5-153, 5-154, 5-166, 5-167, 5-168, A-4
D-bit; TLB field 3-5, 5-164, 5-165
D-cache; data cache 1-4, 2-1, 3-6, 3-14, 4-2, 5-150, A-4
data address translation 2-9, 3-2
data memory break disable 2-8
data memory break trap 2-8, 3-5, 4-20, 5-16, 5-29, 5-30, 5-31, 5-33, 5-39, 5-44, 5-46, 5-47, 5-48, 5-51, 5-166, 5-182, 5-183, 5-186, 5-187
data memory protection trap 3-6, 4-19, 5-15, 5-26, 5-27, 5-28, 5-29, 5-30, 5-31, 5-32, 5-33, 5-34, 5-35, 5-36, 5-39, 5-40, 5-41, 5-42, 5-44, 5-46, 5-47, 5-48, 5-51, 5-166, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
data page fault 4-18, 5-26, 5-27, 5-28, 5-29, 5-30, 5-31, 5-32, 5-33, 5-34, 5-35, 5-36, 5-39, 5-40, 5-41, 5-42, 5-44, 5-46, 5-47, 5-48, 5-51, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
data TLB miss fault 4-18, 5-26, 5-27, 5-28, 5-29, 5-30, 5-31, 5-32, 5-33, 5-34, 5-35, 5-36, 5-39, 5-40, 5-41, 5-42, 5-44, 5-46, 5-47, 5-48, 5-51, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
data types 1-5, 2-1, 2-4
DCOR 5-111
decimal arithmetic 1-6
DECIMAL CORRECT 5-111
delay slot 4-3, 4-5, 4-6, 4-7, 4-8, 5-61
delayed branch 2-10, 4-3, 4-7, 4-10
DEP 5-129
DEPI 5-131
DEPOSIT 5-129
DEPOSIT IMMEDIATE 5-131
DIAG 4-16, 4-17, 5-171
DIAGNOSE 5-171
dirty bit 4-20
disabling an interruption 4-11
displacement A-4
DIVIDE STEP 5-5, 5-102
double-precision 2-5
double-word floating-point 1-5, 2-5
doublewords 2-2, 2-3

DS 5-102
DTLB; data TLB 1-5, 3-4, 3-5, 4-19, 5-162, 5-164, A-4
dynamic displacement 4-4, 4-5, 5-56, 5-63, 5-64, A-4

E

E-bit; TLB field 3-4
effective memory address A-4
EIEM; external interrupt enable mask register 2-12, 4-15, A-4
EIR; external interrupt request register 2-12, 2-14, 4-12, 4-15, A-4
equivalent map A-4
EXCLUSIVE OR 5-105
execute access 3-7, A-1
external branch 4-4, 5-56
external interrupt 1-3, 2-9, 2-12, 2-14, 4-1, 4-14
EXTRACT SIGNED 5-127
EXTRACT UNSIGNED 5-126
extract/deposit condition 5-2, 5-7
EXTRS 5-127
EXTRU 5-126

F

FABS 6-35
FADD 6-30
fault 4-9, A-5
fault-tolerant 1-1, 1-2, 1-7
FCMP 6-43
FCNVFF 6-39
FCNVFX 6-41
FCNVFXT 6-42
FCNVXF 6-40
FCPY 6-38
FDC 5-167
FDCE 3-14, 5-169
FDIV 6-33
FIC 4-19, 5-168
FICE 3-14, 5-170
FLDDS 6-27
FLDDX 6-23
FLDWS 6-26
FLDWX 6-22
FLOATING-POINT ABSOLUTE VALUE 6-35
FLOATING-POINT ADD 6-30
FLOATING-POINT COMPARE 6-43
FLOATING-POINT CONVERT FROM FIXED-POINT TO FLOATING-POINT 6-40
FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT 6-41
FLOATING-POINT CONVERT FROM FLOATING-POINT TO FIXED-POINT AND TRUNCATE 6-42
FLOATING-POINT CONVERT FROM FLOATING-POINT TO FLOATING-POINT 6-39
floating-point coprocessor 1-1, 1-3, 5-172
FLOATING-POINT COPY 6-38
FLOATING-POINT DIVIDE 6-33
FLOATING-POINT LOAD DOUBLEWORD INDEXED 6-23
FLOATING-POINT LOAD DOUBLEWORD SHORT 6-27
FLOATING-POINT LOAD WORD INDEXED 6-22
FLOATING-POINT LOAD WORD SHORT 6-26
FLOATING-POINT MULTIPLY 6-32
FLOATING-POINT REMAINDER 6-36
FLOATING-POINT ROUND TO INTEGER 6-37
FLOATING-POINT SQUARE ROOT 6-34
FLOATING-POINT STORE DOUBLEWORD INDEXED 6-25
FLOATING-POINT STORE DOUBLEWORD SHORT 6-29
FLOATING-POINT STORE WORD INDEXED 6-24
FLOATING-POINT STORE WORD SHORT 6-28
FLOATING-POINT SUBTRACT 6-31
FLOATING-POINT TEST 6-44
FLUSH DATA CACHE 4-19, 5-150, 5-167
FLUSH DATA CACHE ENTRY 5-150, 5-169
FLUSH INSTRUCTION CACHE 4-18, 4-19, 5-150, 5-168
FLUSH INSTRUCTION CACHE ENTRY 5-150, 5-170
FMPY 6-32
following instruction 1-6, 2-9, 2-10, 3-8, 4-2, 4-3, 4-7, 5-8, 5-62, A-5
FPR; floating-point register 2-1, 5-9
FREM 6-36
FRND 6-37
FSQRT 6-34
FSTDS 6-29
FSTDX 6-25
FSTWS 6-28
FSTWX 6-24
FSUB 6-31
FTEST 6-16, 6-44

G

GATE 5-61
GATEWAY 3-8, 4-5, 4-8, 4-16, 5-56, 5-58, 5-61, 5-62
gateway access 3-8, A-1
general register 2-10
GR; general register 1-3, 1-4, 1-5, 1-7, 2-6, 5-9, A-5
group 1 interrupts 2-10, 4-9, 4-11, 4-14
group 2 interrupts 2-10, 4-9, 4-11, 4-14
group 3 interrupts 2-10, 4-1, 4-9, 4-12, 4-15
group 4 interrupts 4-2, 4-10, 4-12, 4-21

H

H-bit; hash chain 3-11
H-bit; PDIR Entry 3-12
H-bit; PSW 2-8, 4-2, 4-8, 4-21, 5-140, A-5

halfwords 2-2, 2-3, 2-5
hash algorithm A-5
higher-privilege transfer trap 2-8, 4-2, 4-8, 4-21
HP Precision Architecture 1-1, 1-2, 1-5
HPMC; high-priority machine check 2-9, 4-1, 4-10, 4-11, 4-14, A-5
HT; hash table 3-11, A-5

I

I-bit; PSW system mask 2-8, 2-9, 2-14, 4-1, 4-12, 4-14, 4-15, 5-140, 5-141, 5-142, 5-143, A-5
I-cache; instruction cache 1-4, 2-1, 3-6, 3-14, 4-2, 5-150, A-6
IA offset 2-6, 4-2, 5-8, 5-56, 5-57
IA queue back 4-7, 5-8
IA queue front 4-7, 5-8
IA queue next 4-7, 5-8
IA queues 4-1, 4-2, 4-13
IA relative branch 4-4, 4-5, 5-56, 5-57, 5-60, 5-61, 5-63, A-6
IA space 2-7, 5-8, 5-56
IA; instruction address 2-12, 4-3
IAOQ; instruction address offset queue 2-9, 4-1, 4-5, 5-8, 5-140, A-5
IASQ; instruction address space queue 2-9, 3-2, 4-1, 4-5, 4-13, 5-8, 5-140, A-6
IDCOR 5-113
IDENTIFY COPROCESSOR 5-174, 5-179
IDENTIFY SFU 5-173, 5-176
IDTLBA 3-5, 4-17, 5-162
IDTLBP 3-5, 4-17, 5-164
IEEE 754 floating-point standard 1-2, 1-6, 2-5
IIA queues 4-10, 4-11, 4-21, 5-140, 5-142, 5-143
IIAOQ; interruption instruction address offset queue 2-12, 2-13, 4-10, 5-140, A-6
IIAQ; interruption instruction address queue 2-9
IIASQ; interruption instruction address space queue 2-12, 2-13, 4-10, 5-140, A-6
IIR; interruption instruction register 2-9, 2-13, 4-10, 4-16, 4-17, 4-18, 4-19, 4-20, 4-21, A-6
IITLBA 4-17, 5-163
IITLBP 4-17, 5-165
illegal instruction 5-1
illegal instruction trap 4-8, 4-16, 5-1, 5-61, 5-155, 5-157
implementation-dependent 5-171
INCLUSIVE OR 5-104
indexed addressing mode 1-2, 1-5
INSERT DATA TLB ADDRESS 5-162, 5-164
INSERT DATA TLB PROTECTION 5-162, 5-164
INSERT INSTRUCTION TLB ADDRESS 5-163, 5-165
INSERT INSTRUCTION TLB PROTECTION 5-163, 5-165
INSERT ITLB ADDRESS 3-5
INSERT ITLB PROTECTION 3-5
instruction address translation 2-9, 3-2
instruction cache 3-2
instruction fetch 4-2
instruction memory protection trap 3-6, 4-1, 4-16, 5-62
instruction page fault 4-1, 4-16
instruction pipelining 4-2
instruction set 1-1, 1-5
instruction TLB miss fault 4-1, 4-15
interlock 5-16
INTERMEDIATE DECIMAL CORRECT 5-113
interrupt 2-1, 2-8, 4-9, A-6
interruption 2-7, 2-9, A-6
interruptions 4-9
interspace branch 2-6, 2-7, 4-4, 4-5, 4-7, 5-56, 5-58, 5-65, 5-66, A-7
interval timer 2-10, 2-12, 5-149, A-7
intraspace branch 4-4, 4-5, 4-7, 5-56, 5-58, A-7
IOR; interruption offset register 2-9, 2-13, 4-10, 4-18, 4-19, 4-20, A-7
IPR; interruption parameter registers 2-13, 5-140, 5-142, 5-143, A-6
IPSW; interruption processor status word 2-7, 2-8, 2-14, 4-10, 5-10, 5-140, 5-142, 5-143, A-7
ISR; interruption space register 2-9, 2-13, 4-10, 4-18, 4-19, 4-20, A-7
ITLB; instruction TLB 1-5, 3-4, 3-5, 4-1, 4-19, 5-163, 5-165, A-7
IVA; interruption vector address 2-12, 4-11, A-7

L

L-bit; PSW 2-8, 4-2, 4-8, 4-21, 5-140, A-7
LDB 5-28
LDBS 5-42
LDBX 5-36
LDCWS 5-44
LDCWX 5-38
LDH 5-27
LDHS 5-41
LDHX 5-35
LDI 5-53
LDIL 5-54
LDO 5-53
LDSID 5-144
LDW 5-26
LDWAS 4-17, 5-43
LDWAX 4-17, 5-37
LDWM 5-32
LDWS 5-40
LDWX 5-34
level one 2-4, 2-7, 2-9, 2-12, 2-13, 3-1, 5-144, 5-145, 5-148
level two 2-4, 2-7, 2-9, 2-12, 2-13, 3-1
level zero 2-4, 2-6, 2-7, 2-9, 2-12, 2-13, 3-1, 5-2, 5-37, 5-49, 5-64, 5-65, 5-66, 5-140, 5-144, 5-145, 5-147, 5-148, 5-149, 5-151, 5-152, 5-153, 5-154, 5-156, 5-157, 5-160, 5-161, 5-162, 5-163, 5-164, 5-165

LHA 4-16, 4-17, 5-157
linkage 4-7
links 5-56
LOAD AND CLEAR WORD INDEXED 4-20, 5-16, 5-19, 5-38
LOAD AND CLEAR WORD SHORT 4-20, 5-16, 5-44
LOAD BYTE 5-28
LOAD BYTE INDEXED 5-36
LOAD BYTE SHORT 5-42
LOAD HALFWORD 5-16, 5-27
LOAD HALFWORD INDEXED 5-20, 5-35
LOAD HALFWORD SHORT 5-22, 5-41
LOAD HASH ADDRESS 3-12, 5-157
LOAD IMMEDIATE LEFT 5-15, 5-54
LOAD OFFSET 5-15, 5-52, 5-53
LOAD PHYSICAL ADDRESS 4-18, 5-155
LOAD SPACE IDENTIFIER 5-144
LOAD WORD 5-16, 5-26
LOAD WORD ABSOLUTE INDEXED 5-15, 5-37
LOAD WORD ABSOLUTE SHORT 5-15, 5-43
LOAD WORD AND MODIFY 5-18, 5-32
LOAD WORD INDEXED 5-20, 5-34, 5-37
LOAD WORD SHORT 5-22, 5-40
local branch 4-4, 5-56
long displacement addressing mode 1-2
long pointer A-7
lower-privilege transfer trap 2-8, 4-2, 4-8, 4-21
LPA 4-16, 4-17, 5-155
LPMC; low-priority machine check 2-9, 2-14, 4-1, 4-15

M

M-bit; PSW 2-9, 4-1, 4-10, 4-14, A-8
machine state register 1-3
main memory 1-3, 1-4, 1-5, 2-1, 2-2
masking an interruption 4-11
memory A-8
memory access protection 1-2
memory address space A-8
memory array 1-3
memory hierarchy 1-4, 1-7, 2-1
memory management 1-2, 3-1
memory-mapped I/O 1-6, 2-1, A-8
MFCTL 5-149
MFSP 5-148
microcode 1-1, 1-2
MOVB 5-67
MOVE AND BRANCH 5-7, 5-67
MOVE FROM CONTROL REGISTER 4-17, 5-149
MOVE FROM SPACE REGISTER 5-148
MOVE IMMEDIATE AND BRANCH 5-68
MOVE TO CONTROL REGISTER 2-14, 4-15, 4-17, 5-146
MOVE TO SPACE REGISTER 4-17, 5-145
MOVE TO SYSTEM MASK 5-143
MOVIB 5-68
MSM A-9
MTCTL 5-146
MTSAR 5-147
MTSM 2-10, 4-13, 4-17, 5-142, 5-143
MTSP 5-145
multiple-precision arithmetic 1-6
multiprocessing 1-1, 1-7, 3-14
multiprocessor 4-3, A-8

N

N-bit; PSW 2-8, 4-1, 4-2, 4-3, 4-7, 5-2, 5-60, 5-61, 5-63, 5-64, 5-65, 5-66, 5-67, 5-68, 5-70, 5-71, 5-72, 5-73, 5-74, 5-75, 5-76, 5-77, 5-78, 5-79, 5-82, 5-83, 5-84, 5-85, 5-86, 5-87, 5-88, 5-89, 5-90, 5-91, 5-92, 5-93, 5-94, 5-95, 5-96, 5-97, 5-98, 5-99, 5-102, 5-103, 5-104, 5-105, 5-106, 5-107, 5-108, 5-109, 5-111, 5-113, 5-114, 5-115, 5-118, 5-119, 5-120, 5-121, 5-123, 5-124, 5-125, 5-126, 5-127, 5-128, 5-129, 5-130, 5-131, 5-132, 5-134, 5-135, 5-136, 5-175, 5-176, 5-177, 5-178, 5-179
NaN A-8
negation flag 5-2
next instruction 4-7
Next PDIR Entry Index 3-11, 3-12
no overflow 5-5
non-access data page fault 4-19, 5-151, 5-152, 5-153, 5-154
non-access data TLB miss fault 4-18, 5-151, 5-152, 5-153, 5-154, 5-155, 5-166, 5-167, 5-168
non-access instruction TLB miss fault 4-18, 5-168
nonexistent bit 2-6, 2-10, 2-12, 5-145
nonexistent hardware 5-172
nonexistent register 2-6, 2-7, 2-9, 2-12, 2-13, 5-65, 5-140
NOP 5-104
null instruction 5-1, 5-2, 5-145, 5-147, 5-150, 5-158, 5-159, 5-160, 5-161, 5-162, 5-163, 5-164, 5-165, 5-166, 5-167, 5-168, 5-169, 5-170
nullification 4-3, 4-7, 4-21, 5-71
nullify 2-8, A-8

O

offset 2-3, 3-2, 4-4, 5-158
optimizing compilers 1-1, 1-2, 1-4
OR 5-104
overflow 5-5, 5-101, 5-117
overflow trap 2-9, 4-17, 5-84, 5-86, 5-89, 5-92, 5-95, 5-97, 5-99, 5-101, 5-115, 5-117, 5-119

P

P-bit; PSW system mask 2-8, 2-9, 2-11, 3-7, 3-8, 4-1, 4-16, 5-141, 5-142, 5-143, 5-151, 5-152, 5-153, 5-154, A-8
packed decimal 1-5, 2-5
page A-8

page reference trap 3-4, 4-20, 5-26, 5-27, 5-28, 5-29, 5-30, 5-31, 5-32, 5-33, 5-34, 5-35, 5-36, 5-39, 5-40, 5-41, 5-42, 5-45, 5-46, 5-47, 5-48, 5-51, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
Page Within Space 3-12
PDC 5-166
PDIR Entry Index 3-11
PDIR; physical page directory 3-4, 3-5, 3-6, 3-11, 4-19, A-8
PDTLB 4-17, 5-158
PDTLBE 3-14, 4-17, 5-160
physical address 2-2, 3-1, 3-4, A-8
pipelining 1-2
PITLB 4-17, 5-159
PITLBE 3-14, 4-17, 5-161
power failure interrupt 2-9, 2-14, 4-1, 4-14
privilege level 2-1, 2-3, 2-6, 2-7, 2-9, 2-10, 2-12, 2-13, 2-14, 3-2, 3-3, 3-5, 3-6, 3-8, 4-2, 4-5, 4-7, 4-10, 5-56, 5-58, 5-61, 5-62, 5-65, 5-173, 5-174, A-1, A-9
privilege level promotion 3-8
privileged instruction 4-17
privileged operation trap 4-16, 5-37, 5-43, 5-49, 5-140, 5-141, 5-142, 5-143, 5-155, 5-157, 5-158, 5-159, 5-160, 5-161, 5-162, 5-163, 5-164, 5-165
privileged register trap 4-17, 5-145, 5-146, 5-149
PROBE READ ACCESS 4-18, 4-19, 5-151
PROBE READ ACCESS IMMEDIATE 4-18, 4-19, 5-152
PROBE WRITE ACCESS 4-18, 4-19, 5-153
PROBE WRITE ACCESS IMMEDIATE 4-18, 4-19, 5-154
PROBER 5-151
PROBERI 5-152
PROBEW 5-153
PROBEWI 5-154
process attributes 3-6
processing resource 2-1, 2-6
processor module 1-3, 1-4
protection identifier 2-11, 2-12, 3-6, 3-7, 3-8, 4-2, 4-16, 4-19, 5-151, 5-152, 5-153, 5-154, A-9
protection mechanism 1-6, 2-1
PSW system mask 2-8, 5-141, 5-142, 5-143, A-9
PSW; processor status word 1-6, 2-7, 2-8, 3-6, 3-7, 4-2, 4-13, 5-10, 5-140, A-9
PURGE DATA CACHE 4-19, 4-20, 5-16, 5-150, 5-166
PURGE DATA TLB 3-5, 5-158
PURGE DATA TLB ENTRY 3-5, 5-160
PURGE INSTRUCTION TLB 3-5, 5-159
PURGE INSTRUCTION TLB ENTRY 3-5, 5-161

Q

Q-bit; IPSW system mask 5-140
Q-bit; PSW system mask 2-8, 2-9, 2-13, 2-14, 4-10, 4-11, 4-13, 5-140, 5-141, 5-142, 5-143, A-9
quad-precision 2-5
quadruple-word floating-point 1-5, 2-5

R

R-bit; PDIR Entry 3-12
R-bit; PSW system mask 2-8, 2-9, 2-10, 4-1, 5-140, 5-141, 5-142, 5-143, A-9
read access 3-6, 3-7, 5-151, 5-152, A-1
real-time 1-1
recovery counter 2-9, 2-10, 4-1, 4-3, 5-2, A-9
recovery counter trap 2-9, 2-10, 4-1, 4-14
reserved bit 2-6, 2-7, 2-10, 2-12, 2-14
reserved register 2-10
RESET SYSTEM MASK 5-142
RETURN FROM INTERRUPTION 4-2, 4-9, 4-13, 5-140
RFI 2-10, 4-13, 4-17, 5-140, A-9
RISC 1-2
RISC; reduced instruction set computer 1-1, 1-2
Roman-8 1-5, 2-4
RSM 2-10, 4-17, 5-142, A-9

S

SAR; shift amount register 2-10, 2-12, 5-78, 5-121, 5-124, 5-125, 5-128, 5-130, 5-132, 5-135, 5-146, 5-149, A-9
SET SYSTEM MASK 5-141
SFU; special function unit 1-3, 1-7, 5-172, 5-173, 5-175, 5-176, 5-177, 5-178, A-9
SH1ADD 5-87
SH1ADDL 5-88
SH1ADDO 5-89
SH2ADD 5-90
SH2ADDL 5-91
SH2ADDO 5-92
SH3ADD 5-93
SH3ADDL 5-94
SH3ADDO 5-95
SHD 5-123
SHIFT AND ADD 5-5
SHIFT DOUBLE 5-121, 5-123
SHIFT ONE AND ADD 5-87
SHIFT ONE AND ADD LOGICAL 5-88
SHIFT ONE, ADD AND TRAP ON OVERFLOW 5-89
SHIFT THREE AND ADD 5-93
SHIFT THREE AND ADD LOGICAL 5-94
SHIFT THREE, ADD AND TRAP ON OVERFLOW 5-95
SHIFT TWO AND ADD 5-90
SHIFT TWO AND ADD LOGICAL 5-91
SHIFT TWO, ADD AND TRAP ON OVERFLOW 5-92
short displacement addressing mode 1-2, 1-5
short pointer 5-144, A-9
signed integer 1-5, 2-5
signed overflow 5-3
single-precision 2-5
single-word floating-point 1-5, 2-5
space identifier 1-5, 2-3, 2-7, 2-9, 2-13, 3-2, 3-3, 3-5, 3-12, 4-4, 4-5, 4-7, 5-8, 5-144, A-10

SPECIAL OPERATION ONE 5-176
SPECIAL OPERATION THREE 5-178
SPECIAL OPERATION TWO 5-177
SPECIAL OPERATION ZERO 5-175
SPOP0 5-175
SPOP1 5-173, 5-176
SPOP2 5-177
SPOP3 5-178
SR; space register 2-7, 3-1, 3-2, 5-9, A-10
SSM 4-13, 4-17, 5-141, A-9
static displacement 4-4, 5-56, 5-57, 5-60, 5-61, A-10
STB 5-31
STBS 5-48
STBYS 5-50
STH 5-30
STHS 5-47
STORE BYTE 5-31
STORE BYTE SHORT 5-48
STORE BYTES SHORT 5-24, 5-50
STORE HALFWORD 5-16, 5-30
STORE HALFWORD SHORT 5-22, 5-47
STORE WORD 5-16, 5-29
STORE WORD ABSOLUTE SHORT 4-20, 5-15, 5-16, 5-49
STORE WORD AND MODIFY 5-18, 5-33
STORE WORD SHORT 5-22, 5-46, 5-49
STW 5-29
STWAS 4-17, 5-49
STWM 5-33
STWS 5-46
SUB 5-96
SUBB 5-98
SUBBO 5-99
SUBI 5-118
SUBIO 5-119
SUBO 5-97
SUBT 5-100
SUBTO 5-101
SUBTRACT 5-96
SUBTRACT AND TRAP ON CONDITION 5-100
SUBTRACT AND TRAP ON CONDITION OR OVERFLOW 5-101
SUBTRACT AND TRAP ON OVERFLOW 5-97
SUBTRACT FROM IMMEDIATE 5-118
SUBTRACT FROM IMMEDIATE AND TRAP ON OVERFLOW 5-119
SUBTRACT WITH BORROW 5-98
SUBTRACT WITH BORROW AND TRAP ON OVERFLOW 5-99
symmetric processor 3-14
SYNC 5-150
SYNCHRONIZE CACHES 4-2, 5-150
system reset 1-3

T

T-bit; PSW 2-8, 4-2, 4-8, 4-21, A-10
T-bit; TLB field 3-4, 4-20, 5-164, 5-165
taken branch 2-8, 2-10, 4-2, 4-4, 4-7, 4-8, 4-16, 5-9, 5-57, A-10
taken branch trap 2-8, 4-2, 4-8, 4-21, 5-60, 5-61, 5-63, 5-64, 5-65, 5-66, 5-67, 5-68, 5-70, 5-71, 5-72, 5-73, 5-74, 5-75, 5-76, 5-77, 5-78, 5-79
temporary register 2-14
TLB dirty bit fault 5-29, 5-30, 5-31, 5-33, 5-39, 5-45, 5-46, 5-47, 5-48, 5-51, 5-182
TLB dirty bit trap 3-5, 4-20, 5-183, 5-186, 5-187
TLB entry 1-5, 2-3, 3-6, 4-2, 5-162, 5-163, 5-164, 5-165
TLB miss A-10
TLB; translation lookaside buffer 1-2, 1-4, 1-5, 1-6, 2-3, 2-11, 2-14, 3-4, 3-5, 3-11, 3-14, A-10
trap 4-9, 4-16, 4-19, A-10

U

UADDCM 5-109
UADDCMT 5-110
unaligned data reference trap 3-6, 4-19, 5-15, 5-16, 5-18, 5-20, 5-22, 5-26, 5-27, 5-29, 5-30, 5-32, 5-33, 5-34, 5-35, 5-40, 5-41, 5-46, 5-47, 5-180, 5-181, 5-182, 5-183, 5-184, 5-185, 5-186, 5-187
unconditional branch 4-3, 4-4, 4-5, 4-7, 4-21, 5-56, 5-60
undefined instruction 5-1
undefined operation 2-13, 5-15, 5-126, 5-127, 5-129, 5-131, 5-134, 5-136, 5-141, 5-174
unified cache 1-4
uniprocessor 1-3
UNIT ADD COMPLEMENT 5-109
UNIT ADD COMPLEMENT AND TRAP ON CONDITION 5-110
unit condition 5-2, 5-6
UNIT XOR 5-108
unpacked decimal 1-5
unsigned integer 1-5, 2-5
unsigned overflow 5-3
unused bit 2-6, 2-10
unused register 2-6
UXOR 5-108

V

V-bit; PSW 2-8, 5-5, 5-102
VARIABLE DEPOSIT 5-128
VARIABLE DEPOSIT IMMEDIATE 5-130
VARIABLE EXTRACT SIGNED 5-125
VARIABLE EXTRACT UNSIGNED 5-124
VARIABLE SHIFT DOUBLE 5-121
VDEP 5-128
VDEPI 5-130
VEXTRS 5-125
VEXTRU 5-124

virtual address 2-10, 3-1, 3-2, 3-4, 3-5, A-10
virtual address register 1-3
virtual address space 1-1
virtual addressing 1-1, 1-5, 2-7
virtual memory 1-2, 2-1, 2-3
VSHD 5-121

W

WD-bit; write disable bit 2-11, 3-7, A-10
words 2-2, 2-3
write access 3-7, 5-153, 5-154, A-1

X

X-bit; PSW 2-8, 4-1, 4-2, 4-3, 4-7, 4-20, 5-2, 5-16, A-11
XOR 5-105, 5-108

Z

ZDEP 5-134
ZDEPI 5-136
ZERO AND DEPOSIT 5-134
ZERO AND DEPOSIT IMMEDIATE 5-136
ZERO AND VARIABLE DEPOSIT 5-132
ZERO AND VARIABLE DEPOSIT IMMEDIATE 5-135
ZVDEP 5-132
ZVDEPI 5-135

READER COMMENT SHEET

Precision Architecture Computers

PRECISION ARCHITECTURE AND INSTRUCTION
Reference Manual

09740-90014 June 1987

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below, and use additional pages if necessary.

Is this manual technically accurate? ☐ **Yes** ☐ **No**

Are the concepts and wording easy to understand? ☐ **Yes** ☐ **No**

Is the format of this manual convenient in size, arrangement, and readability? ☐ **Yes** ☐ **No**

Comments:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM: **Date** _____

Name _____

Company _____

Address _____ _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 718 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
Information Technology Group - Hardware Documentation
19483 Pruneridge Avenue
Cupertino, California 95014-9974

FOLD FOLD

Manual Part Number 09740-90014
Printed in U.S.A. 0687

HEWLETT PACKARD