Advanced Computer Architecture
Advanced Technology
Apple Computer, Inc.

Scorpius Architectural Specification
Revision 1.0

Confidential and Proprietary Information of Apple Computer, Inc.

The information contained in this document is copyrighted in the name of Apple Computer, Inc., and is highly confidential and proprietary. It may only be accessed by authorized Apple employees and/or authorized Apple independent contractors on a "need to know" basis.

Unauthorized use, misuse, access, copying, or disclosure of any or all of the information contained in this document may constitute a violation of your Apple employment agreement or independent contractor's agreement and may result in termination of your employment with Apple and/or in civil or criminal liability.

By proceeding into this document, you acknowledge the copyrighted, confidential, and/or proprietary nature of the information contained in it and you agree to use the information only for the purpose for which it is intended, to maintain the confidentiality of the information, and to refrain from any and all unauthorized use, access, copying, and/or disclosure of such information.

Scorpius Architectural Specification

Contents

Chapter 3. Addressing

Chapter 4. Interrupts and Traps

Chapter 5. Inter-PU Communication & Coordination

Chapter 6. Cache Control Operations

Chapter 7. Measurement Facilities

Chapter 8. Instructions

Appendix A. Instruction Formats and Operation Codes

Appendix B. Real Memory Organization in Antares

Appendix C. Machine Reset in Antares

1. Introduction

1.1 Scope

This document provides a specification of the Scorpius CPU architecture and a systems-programming-level reference for the Antares CPU, a particular implementation of the Scorpius architecture. Scorpius is a tightly-coupled multiprocessor CPU with efficient support for fine-grained parallelism; the architecture was developed to take advantage of the inter-connectivity of single-chip VLSI implementations.[1] Scorpius is intended to be the processing element of a high-performance personal computer system constructed with a minimal number of components.

A Scorpius CPU comprises four independent processing units (PUs) which share access to separate instruction and data caches, a Memory Management Unit, and a Memory/Bus Interface. In addition to communicating through memory, PUs can communicate and can coordinate their activities via *broadcast* instructions, which permit one PU to send data and addresses simultaneously to other PUs and to suspend its execution until other PUs complete execution of their activities. Multiple Scorpius CPUs can be connected via an Interprocessor Bus to form a multiprocessor system in which each CPU has its own local memory which it can share with other CPUs. Support for inter-CPU messaging is provided by *interrupt-on-write* pages. The Interprocessor Bus also provides a means of communicating with other processors (e.g., IO processors).

This chapter gives an overview of the Scorpius CPU, briefly describes the organization of Scorpius-based systems, and introduces terms and notation used in the remainder of the document. Chapter 2 describes CPU organization and presents a programming model of the CPU. Address space organization, translation table structure, and the address translation process are described in

[1] However, multi-chip implementations are feasible. A multi-chip prototype of the Scorpius CPU, called Venus, is currently being developed.

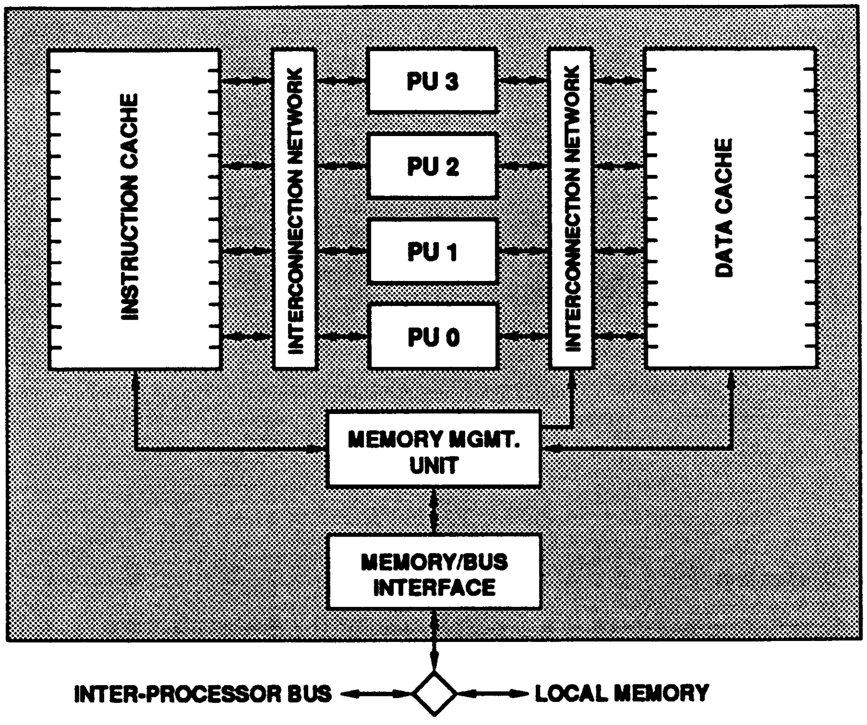


Figure 1.1. Major CPU Elements

Chapter 3. Chapter 4 describes interrupt and trap processing, and gives a summary description of each interrupt and trap. Inter-PU coordination and communication is effected via broadcast and semaphore instructions, which are discussed in Chapter 5. The instruction and data caches are architecturally visible in Scorpius, and instructions are provided to invalidate and flush cache lines. These and other cache control instructions are discussed in Chapter 6. Scorpius provides a pair of event counters with associated controls for use in measuring various aspects of CPU performance, such as instruction execution rates, PU utilization, and cache miss ratios. These measurement facilities are described in Chapter 7. The Scorpius instruction set is described in Chapter 8. In most chapters, descriptions of the Antares implementation are included. Also, the Antares versions of certain implementation-dependent aspects of the CPU↔system interface are described in appendices.

The intent of an CPU architectural specification is to provide a description which is sufficiently complete so that different implementations of that architecture can be built, possibly by different design teams, with confidence that a program executed on any one implementation will produce identical results when executed on any other implementation. These implementations are said to be instruction-level compatible. Instruction-level compatibility is frequently, but not always, required. Sometimes it is sufficient to provide only application-level compatibility, and permit changes in operations performed only by the operating system, so that some operating system functions become model-dependent. To help identify such functions, architectural aspects subject to change in future implementations (to the extent currently recognized) are identified in the text.

1.2 The Scorpius CPU

The major elements of a Scorpius CPU are shown in Figure 1.1. (The shaded region encloses those elements contained on the Antares CPU chip.) The CPU has four identical and independent processing units, or PUs; each is a 32-bit RISC (Reduced Instruction Set Computer). The four PUs access instruction and data caches via interconnection networks. In addition to providing PU˜cache data transfer paths, these networks provide a direct inter-PU communication path for broadcast operations and global register access, as well as a path for interrupt routing. In Antares, instruction and data caches are divided into four banks, and each network is a 5 x 4 crossbar switch, permitting simultaneous instruction and data accesses by all four PUs.

Scorpius provides a flat — unsegmented — virtual address space of 4096 megabytes (MB). A 4-megabyte area at the high end of each address space is reserved for the system kernel; the remaining 4092 megabytes, called *user space*, are available for the user and for other parts of the operating system. The kernel region is not paged, but instead maps directly to the first 4 megabytes of real memory. A single instance of the kernel, then, is common to all address spaces. User space is pageable. The standard page size is 8192 bytes (8KB), but it is possible to define special frame buffer regions in which space is allocated in super-page units, which can range from 256KB to 8MB.

An address space is defined by a set of virtual-to-real page mappings which are recorded in a translation table. Each address space has its own translation table. At any instant, only one address space can be active on a CPU; the four PUs always execute in the same address space. A global register holds a pointer to the start of the translation table for the currently-active address space. Translation table have a simple, two-level structure, composed of a first-level directory and one or more second-level page tables. In addition to virtual-real mappings, translation table entries identify pages as system, read-only, non-cacheable, or interrupt-on-write.

Translation of virtual addresses to real addresses is done by the Memory Management Unit (MMU) using mappings obtained from translation table entries. To avoid reading directory and page table entries on every translation, the MMU maintains the most recently used mappings in a Translation Buffer. Antares has virtually-addressed caches; address translation is required only on a cache miss or on an access to a non-cached page. (The Antares MMU also maintains the CPU's global registers.) A 32-bit virtual address translates into a 36-bit real address, comprising a 4-bit node number and a 32-bit intra-node address. A node number identifies a position on the Inter-Processor Bus (IPB); the node at which a particular real page resides is said to be the *owner* of that page. On a cache miss or a non-cached memory access, the MMU sends a memory access request to the Memory/Bus Interface (MBI), which examines the node number of the real address accompanying the request. If the node number is the same as that of the CPU generating the request, then the request is directed to local memory; otherwise, the request is sent to the specified node, or *remote memory*, via the IPB.

Pages (other than super-pages) can be marked "interrupt-on-write". A store to an interrupt-on-write page causes a message interrupt to be presented to the node owning that page when the store is performed. The interrupt-on-write page can reside in either local or remote memory, and must also be non-cacheable. Interrupt-on-write pages provide a mechanism for transmitting messages between nodes and for coordinating activities of different nodes.

When a CPU receives a message interrupt or an external interrupt (such as an IO interrupt), it examines the status of its four PUs. If one of the PUs is halted, it is assigned to process the interrupt; only if all four PUs are busy is it necessary to actually interrupt PU execution. Interrupt processing, then, frequently can be done in parallel with application execution. Each PU has a flag which indicates if its state must be saved on interrupt. If a PU sets this flag prior to halting, state saving overhead on interrupt processing can be eliminated.

PUs. Scorpius PUs have a small register-oriented instruction set in which all data access to memory is done by register load and store instructions. (Register and word size is 32 bits.) Each PU has 16 general-purpose registers — a total of 64 for the CPU — and 7 local registers. Local registers include product, remainder, prefix, and various state saving registers. In addition, the four PUs share 8 global registers, including interrupt, event counter, and global status registers.

All Scorpius instructions are 16 bits in length. Instructions are tightly encoded, with operation code lengths generally reflecting the number of operands (zero, one, or two). There are only two address modes: register, and base plus displacement. Base plus displacement addressing provides a displacement of up to 64 words from the base register address, with the base register limited to registers 0–3. However, prefixing can be used to increase the displacement range, transform register addressing into base plus displacement addressing (with any register as base), and provide signed displacements.

The 16-bit instruction length limits the size of immediate and displacement fields in Scorpius instructions. However, a large proportion of immediate and displacement values encountered in programs are small enough to be contained in these fields. When necessary, larger values can be created by prefixing the immediate or displacement field value. Each PU has a local register called the Prefix Register, whose state (empty or not empty) is represented by a Prefix Valid flag. Values are loaded into the Prefix Register by a Prefix instruction. If the Prefix Register is empty when a Prefix instruction is executed, the immediate field of the Prefix instruction is stored in the low-order bits of the Prefix Register and sign extended, and the Prefix Valid flag set to not empty. If a second Prefix instruction then is executed, the contents of the Prefix Register are shifted left and the immediate field of the second Prefix instruction stored in the low-order bits of the Prefix Register. When a instruction with a prefixable immediate or displacement is executed, the Prefix Valid flag is examined. If the Prefix Register is not empty, the contents of the Prefix Register are concatenated with the instruction's immediate or displacement field to form the effective immediate or displacement value. Prefixing also is used to define fields for field manipulation instructions.

Like many RISCs, the Scorpius PU has delayed branching. The instruction immediately following a branch, called the *branch shadow* instruction, always is executed, regardless of whether or not the branch is taken. In many implementations, this eliminates the delay which otherwise would result from a a taken branch; compilers usually can move a useful instruction into the branch shadow. Implementations of Scorpius also may have delayed loads. The value loaded from memory into a register by a load instruction may not be immediately available; if the instruction following the load attempts to use that value, it may be delayed. While this is of concern from a performance viewpoint, it is not a functional concern; all implementations provide the necessary interlocks to insure that using instruction does not execute until the register load completes.

Other important aspects of the Scorpius instruction set include the following:

- load byte and store byte instructions with auto-incrementing to speed string handling;
- load and store multiple instructions with auto-incrementing on load and auto-decrementing on store to facilitate register saving and restoring on procedure calls;
- multi-gauge arithmetic instructions which operate simultaneously on both half-words or all four bytes of a word (depending on the mode selected), for high-performance graphics;
- extract, insert, deposit, and double shift instructions to aid in bit field manipulation;
- cache control instructions, including cache line prefetch, flush and invalidate; and
- broadcast instructions, which a PUs uses to send data (or an address) simultaneously to to other PUs, and to suspend its execution until other PUs complete execution of parallel activities.

In Antares, multiply and divide are asynchronous operations which can be overlapped by other instructions.

1.3 Scorpius-Based Systems

With Antares and other single-chip implementations of the Scorpius CPU architecture, systems can be constructed using relatively few components. Figure 1.2 shows the major components of a single-CPU system using the

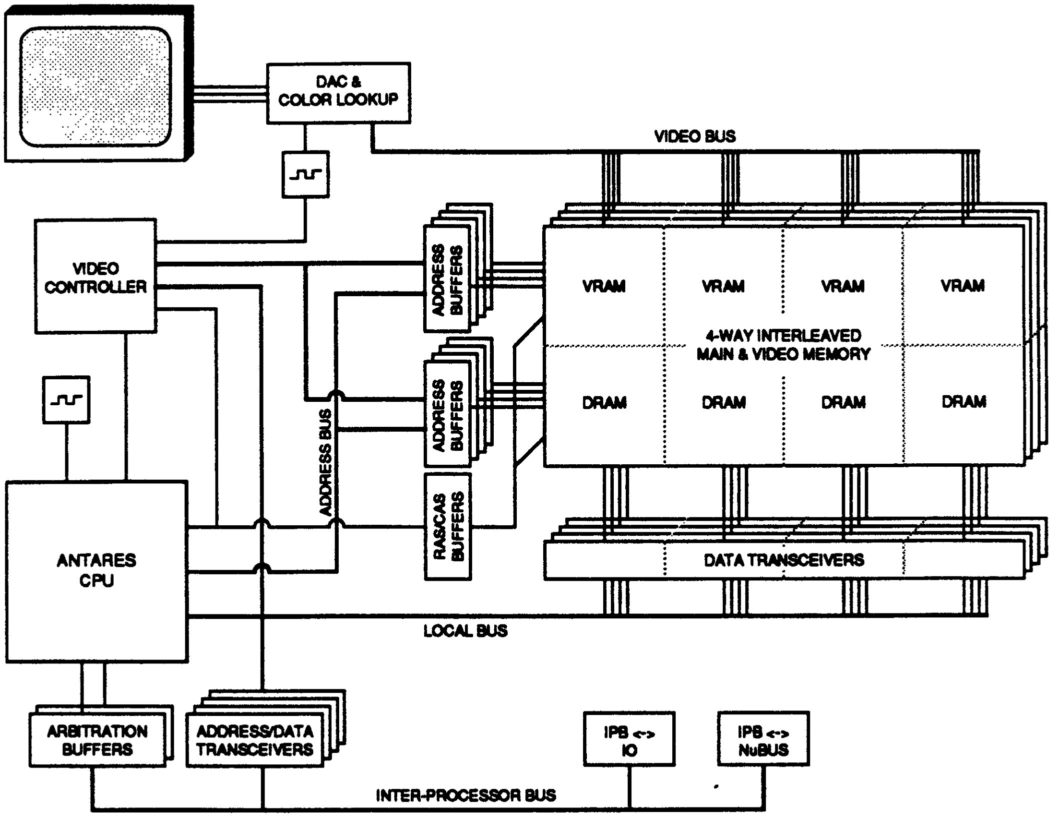


Figure 1.2. Single-CPU System Components

Antares CPU. Because of the parallel processing capabilities of the CPU, and its multi-gauge arithmetic, no separate graphics processor is required. Incorporation of RAM, ROM, and bus control into the Antares CPU chip further reduces the number of components. In this system, memory is four-way interleaved so that data transfers between CPU and memory take place at a maximum rate of one word per cycle. Note that memory is composed of both dynamic and video RAMs. A frame buffer region is created using a single super-page allocated in that part of real memory composed of video RAMs, a screen image is created in the buffer by the CPU using normal load and store operations, and the image then written to the screen over the video bus.

The system of Figure 1.2 can be extended into a multi-CPU system by connecting additional CPUs, each with its own local memory and IPB interface, to the IPB. Sixteen nodes can be directly addressed on the IPB; additional notes can be added via gateways. In a typical multi-CPU system, one CPU will provide the video interface and have its local memory constructed of both VRAMs and DRAMS; the remaining CPUs will have local memory constructed of only DRAMS. However, it is possible that very high performance graphics systems may be developed in which, for example, four CPUs share screen display responsibilities, each operating on a quarter of the screen.

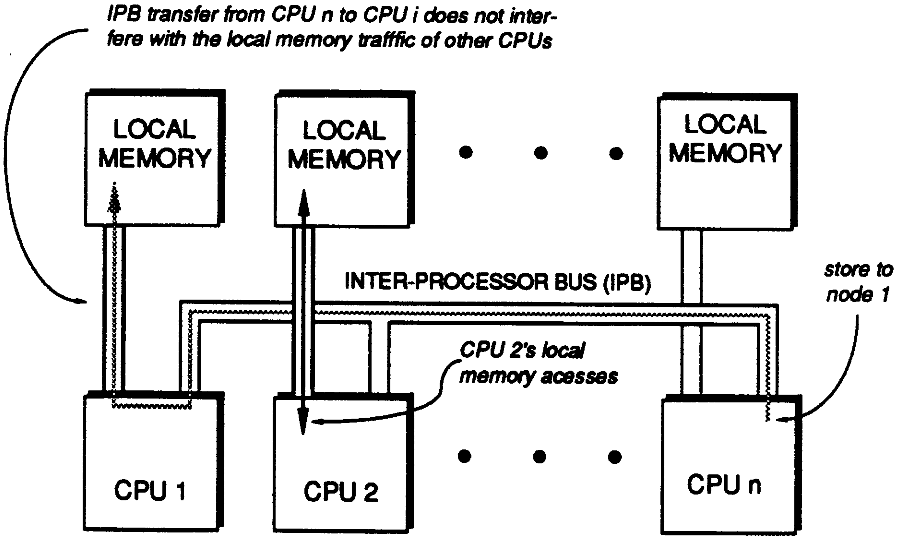

IPB transfer from CPU n to CPU i does not interfere with the local memory trafffic of other CPUs
LOCAL MEMORY
LOCAL MEMORY
LOCAL MEMORY
INTER-PROCESSOR BUS (IPB)
store to node 1
CPU 2's local memory acesses
CPU 1
CPU 2
CPU n

Figure 1.3. Multi-CPU System

A Scorpius virtual address translates into a real address comprising a node number and a intra-node address, so any CPU in a multi-CPU system can share pages with any other CPU in that system. However, the system organization is different from that of a conventional shared memory multiprocessor in which all processors access the same real memory; the operating system model appropriate for the shared memory multiprocessor may be inappropriate for a Scorpius-based multi-CPU system. (From the operating system viewpoint, a Scorpius-based system perhaps should be considered as a form of distributed system.)

In the shared memory multiprocessor, all processors compete for memory access (often by competing for a memory bus), and all memory accesses have the same expected delay. In most Scorpius-based multi-CPU systems, access to local memory is independent of the IPB and is not delayed by IPB activity involving other CPUs, as illustrated in Figure 1.2. Also, depending on implementation, IPB transfers may have greater latency and possibly a lower transfer rate than local memory transfers so that, even in the absence of conflicts, an IPB transfer may take longer than a local memory transfer. Thus, frequently-accesses pages should be located in local memory. Depending on the number of accesses to a shared page, it can be more efficient to copy it from remote memory to local memory before accessing it rather than to access it over the IPB.

Inter-CPU communication in a Scorpius-based system is based on interrupt-on-write pages; various message passing schemes can be implemented using the interrupt-on-write mechanism and shared pages. (The interrupt-on-write mechanism is the only mechanism provided for synchronization of multiple CPUs.) Because of the differences between a Scorpius-based system and a conventional shared-memory multiprocessor, Scorpius, as currently defined, does not incorporate cache coherence in hardware.

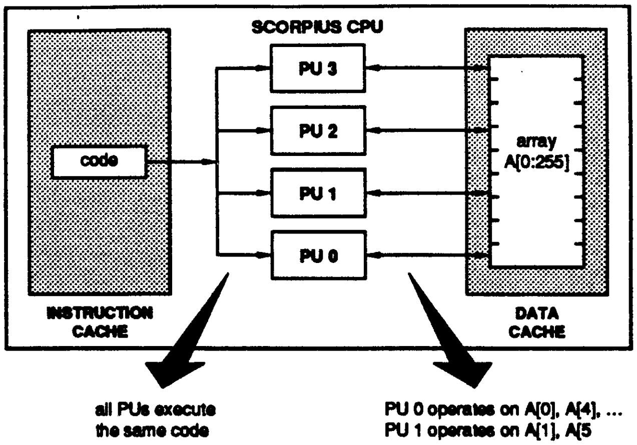


Figure 1.3(a). SIMD Mode Execution

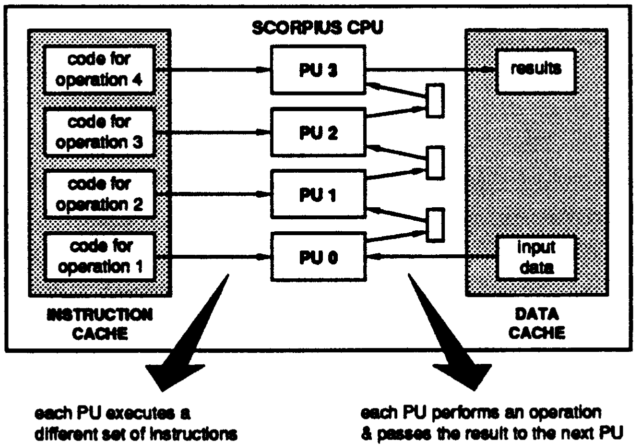


Figure 1.3(b). MISD Mode Execution

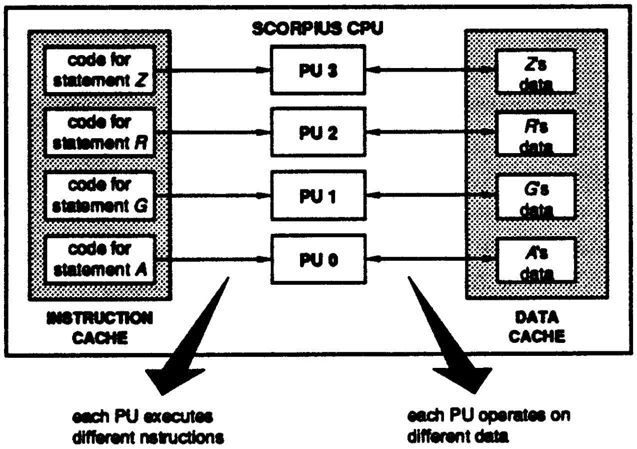


Figure 1.3(c). MIMD Mode Execution

1.4 Parallel Execution

Scorpius programs can execute in any of several parallel modes. These are categorized using (with some liberties) the taxonomy developed by Flynn (Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Trans. on Computers 21, 9* (Sept. 1972), pp948-960). These modes are referred to by acronym in later chapters, and are described below.

SISD (Single Instruction stream, Single Data stream). This mode corresponds to uni-, or serial, processing; only one PU executes. Scorpius programs typically alternate between intervals of serial and of parallel processing. A single PU initiates (and usually participates in) a set of parallel computation activities, and, upon activity completion, may accumulate the results.

SIMD (Single Instruction stream, Multiple Data streams). This mode corresponds to the usual view of parallel processing: each PU executes the same operation on different data streams, as illustrated in Figure 1.3(a), or on different elements of the same data stream. Data access may be ordered or random. In ordered access, inter-PU coordination is implicit, as when each PU operates on every fourth element of a vector. In random access, explicit inter-PU coordination is required, as when PUs operate concurrently on a linked list or take work from a queue. Coordination in this case can be effected through the use of semaphore instructions. This is the easiest form of parallelism to exploit, either with assembly code or by a compiler. For example, the compiler may be able to "unwind" a loop which operates on an array to run on four PUs, with each PU operating on every fourth array element. Optimal performance is easily obtained, since all PUs are doing the same work.[2]

As an example of **SIMD** mode execution, consider the common graphics transformation operation (used in scaling, rotation, and translation) which involves the 1 x 4 matrix multiplication

$$[x^* \; y^* \; z^* \; w^*] = [x \; y \; x \; w] \times \begin{vmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{vmatrix}$$

where

$[x \; y \; x \; w]$ = original coordinate set,

$[x^* \; y^* \; z^* \; w^*]$ = transformed coordinate set,

and the

c_{ij} are fixed (pre-computed) for any given transformation. (For any particular transformation, some of the c_{ij} are known to be 0 or 1.)

[2] In Antares, having each PU operate on every fourth element of a one-dimensional array eliminates data cache conflicts, since the Antares data cache is four-way interleaved with adjacent words located in different banks.

The matrix product can be written as

$$x^* = xc_{11} + yc_{21} + zc_{31} + wc_{41}$$
$$y^* = xc_{12} + yc_{22} + zc_{32} + wc_{42}$$
$$z^* = xc_{13} + yc_{23} + zc_{33} + wc_{43}$$
$$w^* = xc_{14} + yc_{24} + zc_{34} + wc_{44}$$

In a **SIMD** mode implementation of this transformation, PU 0 can be assigned to compute x^*, PU 1 to compute y^*, and so on. Each PU preloads its registers with the appropriate set of constants and, after each *nth* transformation, each PU executes a cache prefetch instruction to prefetch the next line of coordinate data. (Only one prefetch actually takes effect.) By careful scheduling of prefetch and computation operations, very high transformation rates can be realized.

MISD (Multiple Instruction streams, Single Data streams). In this mode, each PU executes a different operation on the same data stream element; data is "pipelined" between PUs (Figure 1.3(b)). For example, consider the computation of

$$y_i = ax_i^3 + bx_i^2 + cx_i + d$$

which might be divided across PUs as follows. (It is assumed that a, b, c, and d are constants and are preloaded into registers of the appropriate PUs.)

- PU 0: load and send x_i, compute and send $cx_i + d$
- PU 1: compute and send x_i^2, bx_i^2
- PU 2: compute $x_i^3 = x_i(x_i^2)$, compute and send ax_i^3
- PU 3: sum intermediate results to form y_i and store y_i

Pipelining of intermediate results from one PU to the next can be done with data broadcast instructions, which also serve to coordinate operations. For example, PU 1 sends x_i^2 to PU 2 by executing a Send instruction; to receive this value, PU 2 executes a Receive instruction. If this Receive is executed before PU 1's Send, PU 2's execution is blocked until the Send is executed. Pipelining also could be done through memory, using Wait and Resume instructions for coordination.

This is a relatively difficult form of parallelism to code or for which to compile code. It is not easy to balance PU execution times to optimize performance. However, carefully crafted hand-coded MISD processing sometimes is useful in improving the performance of critical programs. One stage of the Antares graphics pipeline uses two PUs executing in SIMD mode and two PUs executing in MISD mode.

MIMD (Multiple Instruction streams, Multiple Data streams). This mode is analogous to multiprocessing: each PU executes a different and independent set of instructions which operate on different and independent data elements (Figure 1.3(c)). These might correspond to independent expressions within a single statement, to independent statements, or to certain types of procedures. It is easy to exploit this form of parallel execution at the assembly code level, and it is not too difficult for the compiler to generate MIMD code. However, it may be hard for the compiler to determine independence (because of pointers, for example), and it also can be hard to obtain optimal performance (allocate comparable work to each PU). The independence problem is eased somewhat by compiler directives (e.g., C pragmas) which can be used to identify independent program units.

1.5 Notation and Terms

This section describes notation and terms used throughout the remainder of this document. Notation used in instruction operation descriptions is used only in Chapter 8 and is described in that chapter.

numbers. Unless otherwise specified, all numbers are decimal. Hexadecimal numbers are specified using C language notation in which the hexadecimal number is prefixed by "0x" or "0X". For example, decimal 127 is written in hexadecimal as 0XFF. Bit positions in entities such as registers are indicated in brackets; bracketed numbers separated by a colon indicate a range. For example, "bits <7:0>" specifies bits 7 through 0. Single binary digit and certain multiple binary digit numbers are indicated in quotes, as in "if bit <1> = "1" ". Multi-digit binary numbers are specified by appending the letter "B". For example, decimal 15 is written in binary as 1111B. The letters K and M appended to a number indicate the multipliers 1024 and 1048576; these often appear in conjunction with the letter B, indicating that the unit of measure is bytes. Thus, 64KB, which usually is read as "64 kilobytes", specifies 65536 bytes. Similarily, 2MB usually is read as "2 megabytes" and specifies 2097152 bytes.

undefined and unpredictable operations. The operation of the CPU may be described in certain cases as undefined or as producing unpredictable results. While any given implementation may produce predictable results in such cases, different implementations may produce different results; the behavior of operations described in this way is not reliable.

2. CPU Organization

2.1 Introduction

This chapter describes the elements and organization of the Scorpius CPU. Data and address formats are described, and the programming model is presented. The programming model comprises the elements of the CPU which are visible to the programmer (i.e., can be operated on by instructions). These elements include general registers, status register and program counter, special registers, and, because the Scorpius caches are architecturally visible, the instruction and data caches. An overview of the Scorpius instruction set follows. A detailed description of prefixing, which is used to extend the range of immediate and displacement fields, is given next. Following a description of Scorpius condition codes, the chapter concludes with a discussion of the rationale for and operation of multi-gauge arithmetic.

2.2 Data and Address Formats

Various Scorpius instructions operate on 32-bit *words*, 16-bit *halfwords*, 8-bit *bytes*, and *bits*; instructions themselves always are a halfword in length. Only words and bytes can be directly loaded into a register from memory or stored to memory from a register; arithmetic operations can be performed on words, half-words, and bytes. The 32 bits of a word are numbered, right to left, from 0 to 31. Bit 0, the rightmost bit, is the least significant bit. Higher-numbered bits often are referred to as the *high-order* bits, and lower-numbered bits often are referred to as the *low-order* bits.

Halfwords and bytes within a word are positioned as shown in Figure 2.1; this figure also shows bit ordering within halfwords and bytes. Byte 0 is the most significant (leftmost) byte, while byte 3 is the least significant (rightmost) byte. This ordering is compatible with that of Motorola 680x0 processors, and sometimes is referred to as "big-endian".

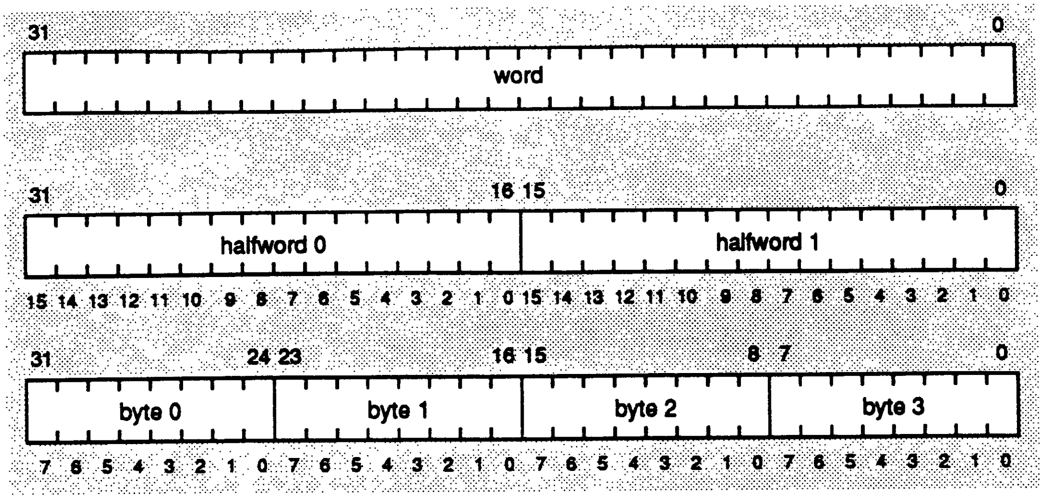


Figure 2.1. Scorpius Data Formats

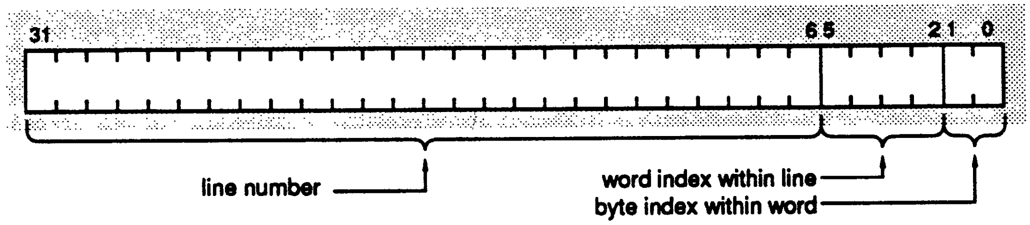
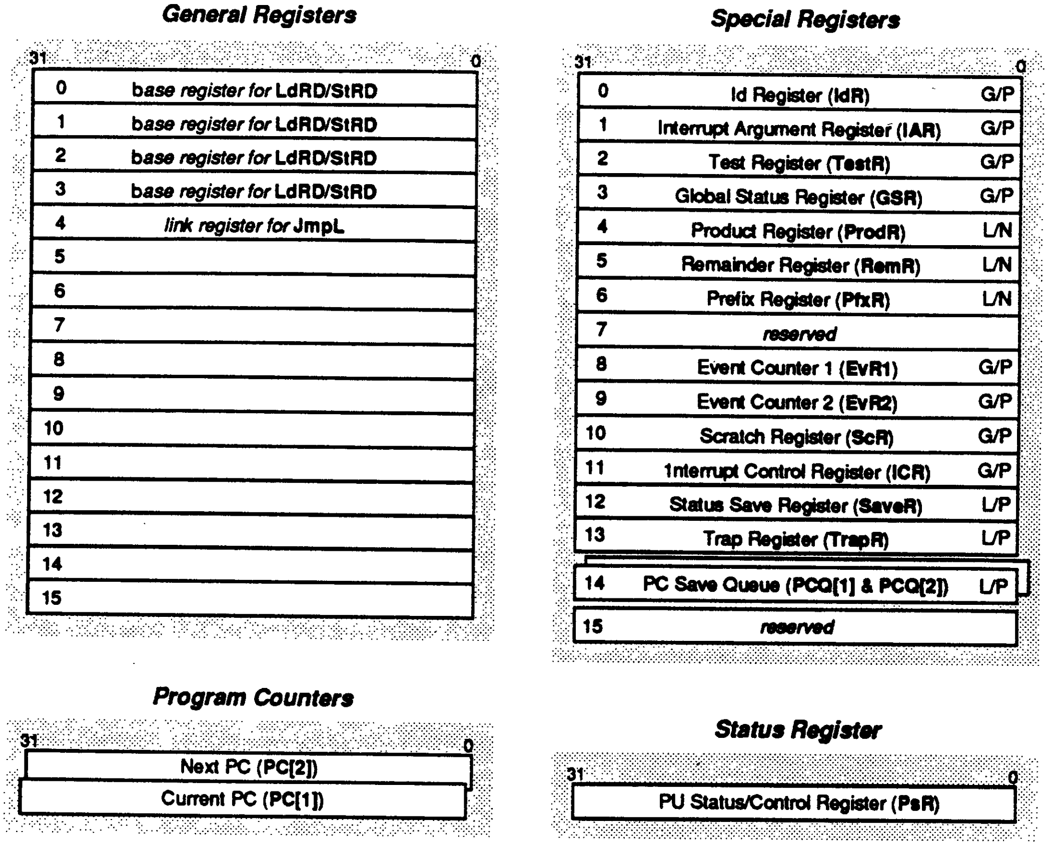


Figure 2.2. Scorpius Address Format

Unless otherwise specified, instructions and data are transferred between memory and the CPU in 64-byte (16-word) blocks called *lines*, which are stored in the instruction cache and data cache. (The term "line" or "cache line" tends to be used both for a physical location in a cache and for a block of 16 memory words which can be stored in that location. The intended meaning usually is clear from the context.) Scorpius instruction and data caches are architecturally visible, and instructions are provided to perform operations on cache lines including prefetch, invalidate, and flush.

Scorpius instruction and data addresses are byte addresses, 32 bits in length, spanning a virtual address space of 4096 megabytes. (Address space organization is discussed in Chapter 3.) While all addresses are byte addresses, memory accesses for instructions and data are constrained to the appropriate boundaries. A halfword boundary is a byte address with bit <0> = "0", a word boundary is a byte address with bits <1:0> = "00", and a line boundary is a byte address with bits <5:0> ="000000". Instructions always must be aligned on a halfword boundary; the low-order bit of an instruction address is ignored. Word operands always must be aligned on word boundaries; the low-order two bits of the operand address of a load or store word instruction are ignored. Cache lines, by definition, are aligned on cache line boundaries; line transfers



General Registers

31 … 0

0	*base register for* LdRD/StRD
1	*base register for* LdRD/StRD
2	*base register for* LdRD/StRD
3	*base register for* LdRD/StRD
4	*link register for* JmpL
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Special Registers

31 … 0

0	Id Register (IdR)	G/P
1	Interrupt Argument Register (IAR)	G/P
2	Test Register (TestR)	G/P
3	Global Status Register (GSR)	G/P
4	Product Register (ProdR)	L/N
5	Remainder Register (RemR)	L/N
6	Prefix Register (PfxR)	L/N
7	*reserved*	
8	Event Counter 1 (EvR1)	G/P
9	Event Counter 2 (EvR2)	G/P
10	Scratch Register (ScR)	G/P
11	Interrupt Control Register (ICR)	G/P
12	Status Save Register (SaveR)	L/P
13	Trap Register (TrapR)	L/P
14	PC Save Queue (PCQ[1] & PCQ[2])	L/P
15	*reserved*	

Program Counters

31 … 0

Next PC (PC[2])

Current PC (PC[1])

Status Register

31 … 0

PU Status/Control Register (PsR)

Figure 2.3. Programming Model: Registers

between the CPU and memory always are done on line boundaries. The low-order six bits of the operand address of a cache control instruction are ignored.

From the viewpoint of the cache, memory comprises a set of lines numbered 0, 1, ..., 0X3FFFFFF. The mapping between these lines and the much smaller number of cache line locations is implementation-dependent. Addresses can be viewed as comprising a line number in bits <31:6>, a intra-line word index in bits <5:2>, and an intra-word byte index in bits <1:0>, as shown in Figure 2.2.

2.3 Programming Model

The programming model comprises the general register set, status register and program counters, the special register set, and the instruction and data caches. The various registers of the programming model are illustrated in Figure 2.3 and described below. Each PU has its own general register set, status register, and program counters; these registers are said to be *local* to the PU. Each PU also has its own copy of certain special registers, while other special registers are common to all PUs; these are called *global* registers. In Figure 2.3, local and global special registers are marked "L" and "G".

Current PC (PC[1]) and Next PC (PC[2])

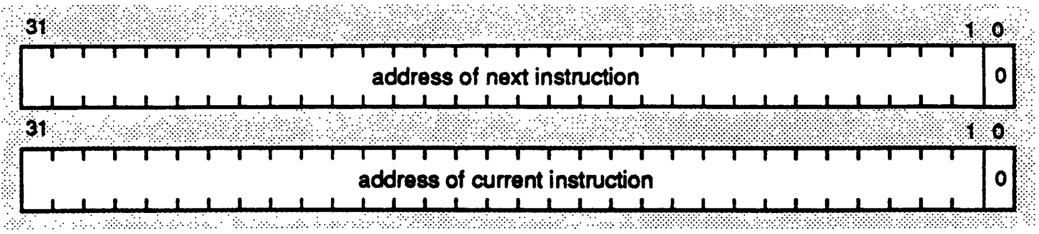


Figure 2.4. Program Counters

PUs execute in one of two modes: *user mode* or *system mode*. The current mode is determined by the setting of a flag in the PU Status/Control Register. Generally, applications execute in user mode, while the operating system kernel and other parts of the operating system execute in system mode. Execution in system mode confers certain privileges. Some special registers can be accessed only in system mode, certain instructions can be executed only in system mode, and pages marked "system only" can be accessed only in system mode. In Figure 2.3, special registers which can be accessed only in system mode are marked "P" (for privileged), while those which can be accessed in either mode are marked "N" (for non-privileged).

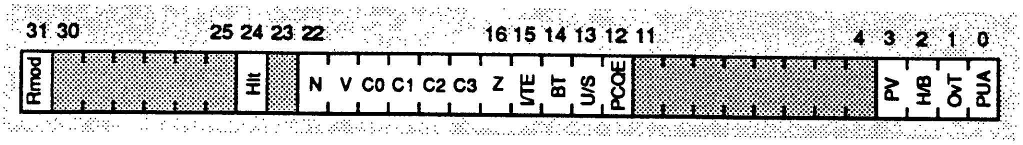
General Registers. Each PU has 16 32-bit general registers, numbered 0–15, so there are 64 general registers for the CPU as a whole. With two exceptions, registers are inter-changeable; any register can be used for any purpose. Registers 0–3, only, can be used as base registers for the Load/Store Register + Displacement instructions, **LdRD/StRD**, and register 4 is used by the Jump and Link instruction, **JmpL**, to store the return address.[1]

Any individual general register can be loaded from memory or have its contents stored to memory via Load/Store Register + Displacement and Load/Store Register instructions. The Load Byte instruction loads the addressed byte, right-justified with zero fill, into a general register; the Store Byte instruction stores the rightmost byte of a general register to memory. From 1 to 15 registers can be loaded from memory or have their contents stored to memory via Load/Store Multiple instructions.

Program Counters. Scorpius has two program counters (PCs), called the Current PC and the Next PC. Current PC holds the address of the currently-executing instruction; Next PC holds the address of the next instruction to be executed. Two program counters are required because of delayed branching. On a taken branch or jump, Current PC holds the address of the branch shadow instruction, and the branch target address is stored in Next PC. For sequential

[1] Using prefixing, any register can be used as a base register for base plus displacement addressing; see section 2.5.

PU Status/Control Register (PsR)



field name	*bit position(s)*	*length*	*description*
Rmod	<31>	1	Register modified flag. Set to "1" on interrupt/trap recognition if the address register of Load/Store Byte or Load/Store Multiple requires adjustment prior to return from interrupt; "0" otherwise (Section 4.6).
Hlt	<24>	1	Halt flag. Set to "1' when a PU executes a Wait instruction specifying itself as a target; cleared to "0" when another PU executes a Start or a Resume instruction with the halted PU specified as a target (Section 5.2).
N	<22>	1	Negative condition code flag (Section 2.6).
V	<21>	1	Overflow condition code flag (Section 2.6).
C0–C3	<20:17>	4	Carry condition code flags (Section 2.6).
Z	<16>	1	Zero condition code flag (Section 2.6).
I/TE	<15>	1	Interrupt/Trap Enable flag. When set to "1", enables recognition of interrupts and traps by the PU; when "0", disables interrupt/trap recognition (Sections 4.3, 4.4).
BT	<14>	1	Taken Branch Trap enable flag. When set to "1", causes a taken branch trap to be generated whenever the PU attempts to execute a taken branch or a jump instruction; when "0", no trap is generated (Sections 4.4 ,4.8).
U/S	<13>	1	User/System mode flag. Set to "1" when the PU is executing in user mode and to "0" when the PU is executing is system mode.
PCQE	<12>	1	PCQ Enable flag. When set to "1", causes PsR and PC contents to be transferred to SaveR and PCQ on interrupt/trap recognition (Section 4.6).
PV	<3>	1	Prefix Valid flag. Set to "1" if the contents of the Prefix Register are valid and to "0" otherwise (Section 2.5).
H/B	<2>	1	Halfword/Byte mode flag. When set to "1", specifies that multi-gauge arithmetic instructions are to operate on halfwords; when "0", specifies that these instructions are to operate on bytes (Section 2.6).
OVT	<1>	1	Overflow Trap enable flag. When set to "1", specifies that a trap is to be generated when overflow occurs on an arithmetic operation; when "0", specifies that no trap is to be generated on overflow (Sections 4.4 ,4.8).
PUA	<0>	1	PU Available flag. When set to "1", advises the operating system that PU state does not have to be saved and restored when the PU processes an interrupt (Section 4.10).

Figure 2.5. PU Status/Control Register Fields

code, the address in Next PC usually is equal to the address in Current PC plus two. Instructions must start on halfword boundaries, so program counter bit <0> always is "0".

The contents of Current PC can be read by executing a Load Program Counter instruction, which loads the address in the Current PC, plus 2, into a general register. In addition to the normal incrementing which takes place in execution of sequential code, program counters are modified when a taken branch or jump instruction is executed, or when a return from interrupt takes place. When an interrupt or a trap is recognized by an interrupt/trap enabled PU, the contents of the Current and Next PCs are saved in a special register pair called the PC save queue; on return from interrupt, the contents of the PC Save Queue are transferred to Current PC and Next PC (see Sections 4.6 and 4.7).

Status Register. The PU Status and Control Register (PsR) contains flags which control PU execution modes, enable or disable generation of certain traps and recognition of interrupts, and record information about the state of the PU and about the results of arithmetic operations. Figure 2.5 shows the PsR and briefly describes each of its flags. Shaded areas in this figure indicate fields reserved for future use.

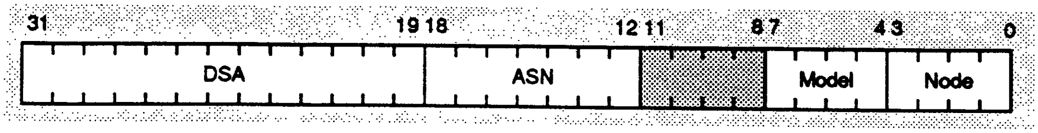
PsR bit <31> (Rmod flag) , bit <24> (Hlt flag), and bits <22:16> (condition codes) are set and cleared only by hardware. Other PsR bits can be set, tested, or cleared via Set Mode, Test Mode, and Clear Mode instructions. Bits <15:8> can be accessed only in system mode; an attempt to access these bits while in user mode causes an operation fault trap to be be generated. Bits <7:0> can be accessed in either system mode or user mode. An attempt to set or clear a reserved bit causes unpredictable results. Testing a reserved bit causes the Z flag to be set.

When an interrupt or trap is recognized, the contents of the PsR are transferred to a special register called the Save Register, and PsR bits <24> and <15:0> then are cleared to "0".[2] Among other things, this clears the mode flag (selecting system mode), disables interrupt/trap recognition, and clears the Halt and PU Available flags. Interrupt/trap recognition also causes a transfer of control to a kernel interrupt/trap address. (The kernel is distinguished from other system code by the range of addresses in which it runs.) On return from interrupt, the contents of the Save Register are transferred to the PsR. In preparing for the the initial dispatch of an address space, the kernel initializes a PU's PsR by setting or clearing Save Register bits as required; return from interrupt causes the Save Register contents to be transferred to the PsR (see Sections 4.6 and 4.7).

Special Registers. Some special registers are local (one instance per PU), while others are global (one instance per CPU). Both local and global special

[2] In Antares, this occurs only if the PU is interrupt/trap enabled (PsR bit <15> = "1"). If the PU is disabled (in which case the interrupt or trap represents an error), the PsR is not saved in the Save Register (also, PC contents are not saved in the PC Save Queue).

Id Register (IdR)



field name	bit position(s)	length	description
DSA	<31:19>	13	Directories Starting Address. The concatenation of the DSA and ASN fields provide bits <31:12> of the real address of the translation table directory for the currently active address space (Section 3.4).
ASN	<18:12>	7	Address Space Number. Address space number of the currently-active address space.
Model	<7:4>	4	Model number. "Hardwired" number assigned each implementation of the Scorpius CPU architecture.[3]
Node	<3:0>	4	Node Number. IPB location, assigned during machine powerup.

Figure 2.6. Id Register Fields

registers are read or written via Move From Special and Move To Special instructions, which move values between general and special registers. An attempt to move to or from a privileged special register (one of the special registers marked "P" in Figure 2.3) while in user mode causes an operation trap to be generated. Also, an attempt to access a non-existent (reserved) special register while in user mode causes generation of an operation fault trap. (The result of an attempt to access a non-existent special register while in system mode is undefined.)

#0 — Id Register. Fields of the Id Register (IdR) identify the model of the CPU, its position on the Inter-Processor Bus (IPB), the number of the currently-active address space, and the starting location of address translation tables in the system. The IdR is a global, privileged, register. Figure 2.6 shows the IdR and briefly describes each of its fields (bits <11:8> are reserved for future use).

The DSA and ASN fields of the IdR specify the starting address of the directory, or first level translation table, for the current active address space. (Translation tables and the translation process are described in Chapter 3.) The DSA and ASN fields (IdR bits <31:12>) can be read or written; other IdR fields can only be read. Generally, the contents of the DSA field are written only on operating system initialization after powerup, and the contents of the ASN field are written only on an address space switch.[4]

[3] Venus is model number 0; Antares is model number 1.

[4] In Antares, writing to the IdR causes the Translation Buffer and its associated Directory Buffer to be flushed; see section 3.6.

The Model field of the IdR contains a "hardwired" model number; this number is used by implementation-dependent code to determine which implementation of the Scorpius architecture it is running on.

The Node field contains the CPU's node number; this number corresponds to the CPU's location on the Inter-Processor Bus (IPB) and is assigned during machine powerup (see Appendix C). A Scorpius real address comprises a 32-bit intra-node real byte address and a 4-bit node number. When performing an address translation, the MMU compares the node number of the translated address with the number in the IdR's node field. If these match, the memory access for which the translation was performed is sent to local memory; if they do not, the access is sent to the appropriate remote memory via the IPB.

#1 — Interrupt Argument Register. The Interrupt Argument Register (IAR) is a privileged global register which holds the argument associated with a pending Message interrupt. This argument is the real intra-node address of the message destination. The IAR is shown in Figure 4.2(b). The contents of IAR can be read via a Move From Special instruction; the contents are valid from the point at which the Message interrupt is presented up to the point at which the Message Interrupt Pending flag in the Interrupt Control Register is cleared (see Section 4.3). The IAR can be written (for test purposes) via a Move To Special instruction; however, the CPU should be disabled for external interrupts so that a real Message interrupt is not presented while testing.

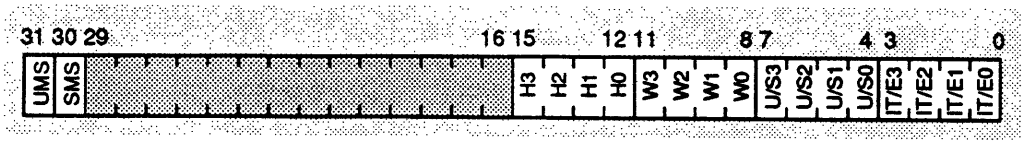
#2 — Test Register. The Test Register (TestR) is a global, privileged register provided for diagnostic purposes. The Test Register can be read or written; its functions are implementation-dependent.

#3 — Global Status Register. The Global Status Register (GSR) is a privileged global register which serves two functions: it holds the user and system mode semaphore flags (in bits <31:30>), and it records the state and mode of the four PUs (in bits <16:0>). Figure 2.7 shows the GSR and briefly describes its fields. Bits <29:16> are reserved for future use; the values returned in these bit position when the GSR is read are undefined. The GSR cannot be written; the result of attempting to execute a Move To Special instruction with the GSR as the destination register is unpredictable.

Semaphore operations are performed by Lock and Unlock instructions. The Lock instruction examines the semaphore flag corresponding to the PU's mode. If the flag is "1", it is changed to "0", and Lock instruction execution completes. If the flag initially is "0", the Lock instruction waits until it becomes "1", changes it to "0", and then completes. The semaphore flag is unconditionally set to "1" via an Unlock instruction. Semaphore operations are discussed in Section 5.4.

The state and mode flags in bits <15:0> are used in deadlock detection and analysis. The CPU monitors the halt and wait flags in the GSR. If it finds that all four PUs are either halted or waiting, it generates a Deadlock interrupt. This is a non-maskable interrupt which is presented to and immediately recognized by PU 0. PU 0 uses the state and mode information to help analyze the cause of the deadlock and to determine how to initiate recovery (see Section 5.5).

Global Status Register (IdR)



field *name*	*bit* *position(s)*	*length*	*description*
UMS	<31>	1	User Mode Semaphore flag. Set to "0" if the user semaphore is locked and to "1" if the semaphore is unlocked (Section 5.4).
SMS	<30>	1	System Mode Semaphore flag. Set to "0" if the system semaphore is locked and to "1" if the semaphore is unlocked (Section 5.4).
H3-H0	<15:12>	4	Halt flag copies. Hi is "1" if PU i is halted (PsR bit <24> = "1") and is "0" otherwise.
W3-W0	<11:8>	4	Wait flags. Wi is "1" if PU i is in wait state (see Section 5.5 for definition) and is "0" otherwise.
U/S1-US3	<7:4>	4	User/System Mode flag copies. U/Si is "1" if PU i is in user mode (PsR <13> = "1") and is "0" if PU i is in system mode (PsR <13> = "0").
IT/E3-IT/E0	<3:0>	4	Interrupt/Trap Enabled flag copy. IT/Ei is "1" if PU i is interrupted/trap enabled (PsR <15> = "1") and is "0" otherwise.

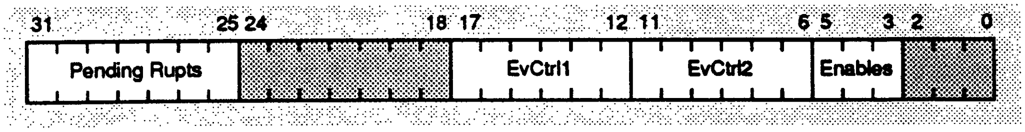
Figure 2.7. Global Status Register Fields

#4 — Product Register. The Product Register (ProdR) is a local, non-privileged register which holds the upper 32 bits of the 64-bit product produced by a 32-bit multiply instruction (**Mul** or **MulU**), or the four product bytes or two product halfwords produced by a multiply partial instruction (**MulP** or **MulPU**).

#5 — Remainder Register. The Remainder Register (RemR) is a local non-privileged, register which holds the remainder produced by a divide instruction (**Div**, **DivE**, **DivU**, or **DivUE**). RemR also is used by the divide extended instructions (**DivE** and **DivUE**) in forming the 64-bit dividend; for these instructions, the 64-bit dividend is formed by concatenating the contents of the Remainder Register with the contents of the specified general register.

#6 — Prefix Register. The Prefix Register (PfxR) is a local, non-privileged, register which can be loaded with a constant value to be used in extended the range of a displacement or an immediate, or with a field description (position and length) for field manipulation instructions. Values are loaded in the PfxR by Prefix Immediate and Mask Generate instructions; these also set the Prefix Valid flag in the PsR (bit <3>). Instructions which use the PfxR clear the Prefix Valid flag. The PfxR also can be read or written via

Interrupt Control Register (ICR)



field name	bit position(s)	length	description
Pending Rupts	<31:25>	7	Pending Interrupt flags. This field contains a flag for each interrupt type; a Pending Rupt flag is set to "1" if an interrupt of the corresponding type is pending and is "0" otherwise (Section 4.3).
EvCtrl1	<17:12>	6	Event Counter 1 Control flags. The flags of this field specify the source of events (PU or MMU) to be counted by event counter 1, and specify the event to be counted (Chapter 7).
EvCtrl2	<11:6>	6	Event Counter 2 Control flags. The flags of this field specify the source of events (PU or MMU) to be counted by event counter 2, and specify the event to be counted (Chapter 7).
Enables	<5:3>	3	Interrupt Enable flags. This field contains flags which control the generation of Event Counter Overflow interrupts and the presentation of external (IO and Message) interrupts (Section 4.3).

Figure 2.8. Interrupt Control Register Fields

Move From/To Special instructions, but these instructions do not affect the Prefix Valid flag. (This flag can be set or cleared via Set/Clear Mode instructions.) Bits <1:0> of the PfxR are unused; writing the PfxR (via a Move To Special instruction) does not affect these bits. When reading the PfxR using a Move From Special instruction, "0" is returned in these bit positions. A detailed description of prefixing and PfxR use is presented in Section 2.5.

#8, #9 — Event Counters 1 and 2. The two event counters (EvR1 and EvR2) are 32-bit global, privileged, registers which, under control of event counter control flags in the Interrupt Control Register, can be used to count various PU and MMU events as well as PU active time. Setup and use of these counters is discussed in Chapter 7.

#10 — Scratch Register. The Scratch Register (ScR) is a global, privileged, register used in saving registers on interrupt/trap recognition. When an interrupt or a trap is recognized and PU state must be saved (the PUA flag in the PsR = "0"), the kernel uses the system mode semaphore to enter a critical section, stores the contents of one general register in ScR and uses that register — typically register 0 — to form the address of a save area in memory. After saving registers, including the register temporarily moved to ScR, the kernel exits the critical section, making the ScR available to some other PU.

Trap Register (TrapR)

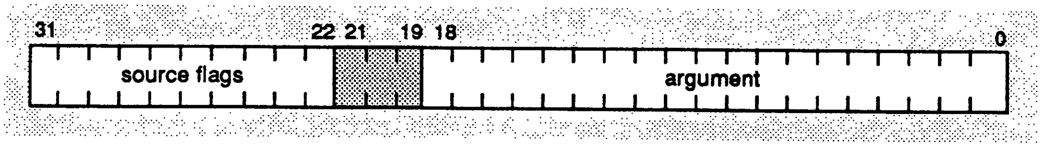


Figure 2.9. Trap Register Fields

#11 — Interrupt Control Register. The Interrupt Control Register (ICR) is a global, privileged, register which contains interrupt pending and enable (mask) flags, together with control flags for the two event counters. Figure 2.8 shows the fields of the ICR and briefly describes their function; ICR bits <24:18> and <2:0> are reserved fields. There is a pending interrupt flag for each type of interrupt; when an interrupt occurs, the corresponding flag is set in the ICR and a PU selected to process the interrupt; the kernel clears the interrupt pending flag as part of interrupt processing. Figure 4.2(a) shows the individual pending interrupt and interrupt enable flags, which are discussed at length in Section 4.3. Individual event counter control flags are described in Chapter 7.

The contents of the ICR can be read via a Move From Special instruction; the values returned in reserved field bits are undefined. The ICR can be written via a Move To Special instruction. However, pending interrupt flags are set to "1" only by hardware; software can only clear a pending interrupt flag to "0". If an attempt is made to write a "1" to a pending interrupt flag, the state of the flag does not change. (See ***Pending Interrupt Flags*** in Section 4.3.)

#12 — Status Save Register. The Status Save Register (SaveR) is a local, privileged, register. When an interrupt or a trap is recognized, the contents of the PsR are moved to the SaveR, from where they can be examined via a Move From Special instruction.[5] The fields of the SaveR correspond to those of the PsR (Figure 2.5); when reading the SaveR, the values returned in reserved field bits are undefined. Following interrupt/trap recognition, the contents of the SaveR remain valid only up to the point at which the PU reenables interrupt/trap recognition.

The SaveR can be written via a Move to Special instruction. On return from interrupt (which is effected by executing a Return From interrupt instruction pair), the contents of the SaveR are copied to the PsR and the contents of the PCQ are copied to the Current and Next PC. To dispatch a newly-initiated address space, each PU sets SaveR and PCQ as appropriate and performs a return from interrupt. (See Sections 4.6 and 4.7).

#13 — Trap Register. The Trap Register (TrapR) is a local, privileged, register which contains trap source flags and, for certain traps, a trap

[5] In Antares, the transfer of the PsR to SaveR on interrupt or trap recognition takes place only if the PU was interrupt/trap enabled — PsR bit <15> = "1" — at the time of recognition.

PC Save Queue (PCQ)

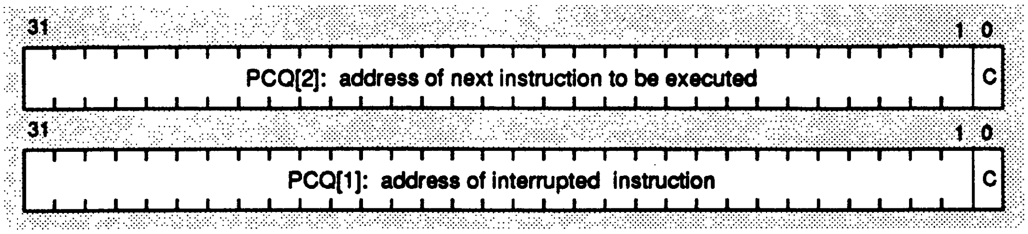

31
1 0
PCQ[2]: address of next instruction to be executed
C
31
1 0
PCQ[1]: address of interrupted instruction
C

Figure 2.10. The PC Save Queue

argument. The Trap Register is shown in Figure 2.9; Figure 4.2(c) shows the individual trap source flags. There is a trap flag for each trap (except the PU Check trap, which is identified by its kernel entry address); a trap source flag is set to "1" by hardware when the corresponding trap is recognized, and cleared by the kernel as part of trap processing. TrapR bits <21:19> are reserved.

Following interrupt/trap recognition, the contents of TrapR remain valid only up to the point at which the PU reenables interrupt/trap recognition. The Trap Register can be read via a Move From Special instruction and written via a Move To Special instruction. When reading TrapR, the values returned in bits <21:19> are undefined. While trap source flags can be set to "1" by software, as well as cleared to "0", software setting of a trap source flag is ignored and does not result in trap recognition.

#14 — PC Save Queue. This register (Figure 2.10) is a FIFO register pair, comprising two 32-bit register, PCQ[1] and PCQ[2]. (The register pair sometimes is referred to as PCQ.) When an interrupt or a trap is recognized, the instruction address in Current PC is transferred to PCQ[1], and the instruction address in Next PC is transferred to PCQ[2].[6] On return from interrupt, these transfers are reversed. Following interrupt/trap recognition, the contents of PCQ remain valid only up to the point at which the PU reenables interrupt/trap recognition.

Because instructions must start on halfword boundaries, the low-order bit of an instruction address is ignored when fetching instructions. In the PCQ, this bit position is used to hold a Correction flag, or C flag. When the Correction flag in PCQ[1] is set on interrupt/trap recognition, it is necessary to adjust the address in PCQ[1] prior to returning from interrupt (see Section 4.6). The Correction flag in PCQ[2] can always be ignored. (Address adjustment may be required because of the "folding" of branch, prefix, and other instructions.)

[6] In Antares, the transfer of Current PC and Next PC to PCQ[1] and PCQ[2] on interrupt or trap recognition takes place only if the PU was interrupt/trap enabled — PsR bit <15> = "1" — at the time of recognition.

The contents of PCQ[1], only, can be read via a Move From Special instruction; PCQ[2], only, can be written via a Move To Special instruction. Whenever PCQ[2] is written, the original contents of PCQ[2] are transferred to PCQ[1]. Thus, correcting the address in PCQ[1] or saving PCQ contents prior to an address space switch requires that moves be done in the correct order. (See ***PC Save Queue Access*** in Section 4.6.)

Caches. In Scorpius, the instruction and data caches are architecturally visible and constitute part of the programming model (cache control instructions are discussed in Chapter 6). The extent of the visibility of the caches depends on the implementation. All implementations have a cache line size of 64 bytes, and separate instruction and data caches with software maintenance of coherence between the two. Antares has a virtually-addressed caches which may contain synonyms. Coherence and synonym issues are discussed in Section 3.8. Also, the size of the data cache (64 lines) is visible in Antares, which provides an instruction, Read Data Tag by Index, to read data cache line tags (used in clearing the cache on an address space switch).

2.4 Instruction Set Overview

This section provides an overview of the Scorpius instruction set. A specification of each instruction, including format and potential exceptions, is given in Chapter 8. A summary of the instruction set is shown in Figure 2.11. This figure organizes the 81 Scorpius instructions, by function, into eight groups. The discussion in this section follows this organization. In this discussion, the term "register", unqualified, always refers to a general register.

Scorpius instructions are 16 bits in length, limiting the size of displacement and immediate fields. In many cases, the effective values of immediates and displacements can be extended by prefixing. The Prefix Register can be loaded with a value by executing one or more Prefix Immediate (**PfxI**) instructions. When a prefixable instruction is executed and the Prefix Register is not empty, its contents are concatenated with the instruction's immediate or displacement field to form the effective immediate or displacement. The Prefix Register also is used to hold field descriptions for field manipulation instructions. In Antares, prefix instruction execution frequently can be folded, or combined, with the execution of some other instruction so that it effectively executes in 0 time. Most displacement and immediate values are small, and can be specified directly by the instruction's displacement or immediate field, so that the corresponding operation typically requires one cycle and 16 bits of instruction stream. When a larger value is required, prefixing can be used; in most cases, the operation will still be carried out in one cycle, although 32 bits of instruction stream will be used. For very large values, two prefix instructions may be needed, which may add a cycle to the operation. Prefixing is discussed in the next section.

Most arithmetic and logical instructions use a two (register) address format in which one operand register also is the result register. Certain instructions,

MNEMONIC	OPERATION	MNEMONIC	OPERATION
	LOAD, STORE, AND MOVE		*ARITHMETIC*
LdI	Load Immediate	**Add/Sub**	Add/Subtract
LdR/StR	Load/Store Word (Register)	**AddC/SubC**	Add/Subtract with Carry
LdRD/StRD	Load/Store Word (Base + Disp.)	**AddI/SubI**	Add/Subtract Immediate
LdB/StB	Load/Store Byte	**AddP/SubP**	Add/Subtract Partial
LdM/StM	Load/Store Multiple	**CLZ**	Count Leading Zeroes
Lcc	Load Condition	**Div**	Divide
LdCP	Load Carry Partial	**DivE**	Divide Extended
LdPC	Load Program Counter	**DivU**	Divide Unsigned
LdPU	Load PU Number	**DivUE**	Divide Unsigned Extended
Mov	Move Register	**Mul**	Multiply
MovFS	Move From Special	**MulU**	Multiply Unsigned
MovTS	Move To Special	**MulP**	Multiply Partial
	BRANCH, COMPARE, & JUMP	**MulPU**	Multiply Partial Unsigned
Bcc	Branch Relative on Condition	**Neg**	Negate
Cmp	Compare Register		*BROADCAST & SEMAPHORE*
CmpI	Compare Immediate	**Rcv**	Receive
CmpP	Compare Partial	**Rsm**	Resume PUs
Jmp	Jump Relative	**Send**	Send
JmpL	Jump and Link	**Strt**	Start PUs
JmpR	Jump Register	**Wait**	Wait PUs (or Halt)
TstF	Test Field	**Lock**	Lock Semaphore
TstM	Test Mode	**Unlk**	Unlock Semaphore
	LOGICAL & SHIFT		*CACHE CONTROL*
And	And	**CDC**	Create Data Cache line
AndC	And Complement	**FDC**	Flush Data Cache line
Not	Not	**IDC**	Invalidate Data Cache line
Or	Or	**IIC**	Invalidate Instruction Cache line
XOr	Exclusive Or	**IICA**	Invalidate Instruction Cache
Dsh	Shift Double	**PDC**	Prefetch Data Cache line
ShL	Shift Left	**RDTX**	Read Data Tag by Index
ShR	Shift Right	**UDC**	Update Data Cache line
		VDC	Validate Data Cache line
	FIELD MANIPULATION		
ClrF	Clear Field		*CONTROL & MISCELLANEOUS*
Dep	Deposit	**ClrM**	Clear Mode
ExtS	Extract Signed	**Prmpt**	Preempt PUs
ExtU	Extract Unsigned	**Res**	Restart PUs
Ins	Insert	**Rti**	Return from Interrupt
Msk	Define Field	**SetM**	Set Mode
PfxI	Prefix Immediate	**Trap**	System Call
SetF	Set Field		

Figure 2.11. Scorpius Instruction Set Summary

including the field manipulation instructions, use the Prefix Register as an implicit operand register.

Addressing Modes. All memory accesses for operands in Scorpius are performed by load and store instructions using one of two addressing modes: (base) register mode and base plus displacement mode. In register mode, the operand address simply is the value contained in a specified register. For certain instructions, this value may be incremented or decremented as part of instruction execution. In base plus displacement addressing, the operand address is formed by adding the value in the instruction's displacement field to the contents of a specified register. For Load Word/Store Word instructions, the address mode can be either register or base plus displacement, depending on the Prefix Register state.

Load, Store, and Move Instructions. The Load Immediate (**LdI**) instruction loads an 8-bit immediate value in the designated register; this immediate value can be extended by prefixing. The Load Word and Store Word instructions **LdR** and **StR** load a register with a word from memory or store the contents of a register to memory. If the Prefix Register is empty, the operand address is provided by a register. If the Prefix Register is not empty, the operand address is formed by using the contents of a register as a base address to which is added, as displacement, the value in the Prefix Register.

The Load Word and Store Word instructions **LdRD** and **StRD** load a register with a word from memory or store the contents of a register to memory using base plus displacement addressing. The base register must be register 0, 1, 2, or 3; the displacement range is 1 through 64 words (which can be extended via prefixing). In terms of frequency, displacements tend to be small (as when accessing local variables on the stack or elements of many structures). **LdRD** and **StRD** provide a compact means of accessing variables in these cases. For larger displacements, prefixing can be used with **LdRD/StRD**; for other base registers, prefixing can be used with **LdR** /**StR** .

The Load Byte instruction **LdB** loads a byte from memory into the low-order 8 bits of a register and clears the high-order bits; the Store Byte instruction **StB** stores the low-order 8 bits of a register to memory. **LdB** and **StB** use register addressing with auto-increment; after loading or storing a byte, the operand address in the base register is incremented. These instructions speed string handling.

Load and Store Multiple instructions (**LdM** and **StM**) are provided to facilitate register saving and restoring on procedure calls. These two instructions use register addressing; **StM** decrements the operand address in the base register by the number of registers stored (x 4, to reflect the numbers of bytes loaded), and **LdM** increments the operand address in the base register by the number of registers loaded (again x 4). The auto-increment/decrement features of these instructions reduce the number of instructions required by procedure call protocols.

The Load Condition Code instruction **Lcc** sets the low-order bit of a register to "1" if condition codes match the encoding in its **cc** field and to "0" otherwise; the remaining bits of the register are cleared. The Load Carry Partial instruction **LdCP** sign extends the carry condition flags resulting from a preceding multigauge arithmetic operation and stores the result in a register. If the Halfword/Byte Mode flag (PsR bit <2>) is "1", halfword carry condition code flags C0 and C2 are sign extended into the two halfwords of the result register. If this flag is "0", byte carry condition code flags C0, C1, C2, and C3 are sign extended into the four bytes of the result register. (Condition codes and multigauge arithmetic are discussed later in this chapter.)

The Load Program Counter instruction **LdPC** loads the address of the current instruction plus 2 into a register. The Load PU Number instruction **LdPU** loads the number (0-3) of the PU on which the instruction is executed into a register. This is the only way in which a PU can determine its identity. One use of this instruction is in offsetting array addresses for each PU when a loop operation on an array is unwound across four PUs, as in the example shown in Figure 5.2.

The Move Register instruction **Mov** moves the contents of one register, the source register, to a second register, the destination register. **Mov** does not affect condition codes; a Move Register instruction in which the same register is specified as both source and destination is used as a "NOP" instruction. The Move From Special and Move To Special instructions **MovFS** and **MovTS** move values between general and special registers.

Delayed Loads. In most Scorpius implementations, including Antares, the majority of instructions execute at a rate of one instruction per cycle. Certain instructions, such as **LdM** and **StM**, are "multi-cycle" instructions. (Some instructions effectively require 0 cycles because of folding.) Also, in most implementations, the load instructions **LdR**, **LdRD**, and **LdB** execute in one cycle; however, the contents of the register being loaded are not available until the following cycle (assuming a cache miss does not occur). Similarly, the contents of the last register loaded by **LdM** are not available until one cycle after the **LdM** completes execution. For this reason, loads from memory are called *delayed loads*. If an instruction following a **LdR**, **LdRD**, or **LdB** instruction attempts to use the contents of the register loaded by the **LdR**, **LdRD**, or **LdB** instruction, or if the instruction following a **LdM** instruction attempts to use the last register loaded by the **LdM** instruction, that instruction will incur a one-cycle delay. Such delays can usually be avoided by appropriate ordering of instructions. (If a useful instruction cannot be inserted between the instruction loading a register and the instruction using the contents of that register, it is not necessary to insert a NOP instruction; hardware delays the instruction attempting to access the register until the register contents can be made available.)

Branch, Compare, and Jump Instructions. A branch instruction is a conditional transfer of control; a jump instruction is an unconditional transfer of control. (Usually, when the distinction is not important, both are referred to as branches.)

As a debugging aid, Scorpius provides a Taken Branch trap which, when enabled (by setting the Taken Branch Trap Enable flag in the PsR), causes a trap to be generated whenever a conditional branch is taken or a jump instruction is executed.

The Branch on Condition instruction **Bcc** causes control to be transferred if the condition codes in the PsR correspond to the value encoded in the instruction's **cc** field. The address to which control is transferred is called the *branch target* address, or simply the target address. This address is formed by shifting the signed value in the **Bcc** instruction's displacement field left one position and adding it to the contents of Current PC. The 8-bit displacement field provides a branch range of -256 to +255 instruction locations. This range accommodates a very large proportion of branches; for branches out of this range, the condition code test can be inverted and the **Bcc** instruction used to branch around a Jump Relative or Jump Register instruction. Later versions of the Scorpius architecture may extend prefixing to include branch and jump displacements, as discussed in the next section.

The Compare Register instruction **Cmp** compares the contents of two registers and sets the condition code flags in the PsR as if the contents of one register were subtracted from the other. The Compare Immediate instruction **CmpI** compares an immediate value with the contents of a register, and sets the condition code flags as if the contents of the register were subtracted from the immediate. If the Prefix Register is empty, the immediate value is taken from the **CmpI** instruction's immediate field, which provides an immediate range of 0-255. If the Prefix Register is not empty, the immediate is formed by concatenating the contents of the Prefix Register and the **CmpI** instruction's immediate field. Most immediate values used in comparisons are small, and can be accommodated in the **CmpI** instruction's immediate field; when a large value is required, the instruction can be prefixed — usually without a time penalty.

Depending on the value of the Halfword/Byte Mode flag in the PsR, the Compare Partial instruction **CmpP** compares either the bytes or the halfwords in two registers. When comparing bytes, condition codes C0, C1, C3, and C3 are set to reflect the result of the comparison; when comparing halfwords, C0 and C2 are set. The Z condition code is set if any of the comparisons yields an equality. In addition to its use in multigauge arithmetic, **CmpP** can be used in scanning strings.

The Jump Relative instruction **JmP** unconditionally transfers control to a target address formed by shifting the signed displacement field of the instruction left one position and adding the result to the contents of Current PC. The 11-bit displacement field provides a range of -1024 to +1023 instruction locations. The Jump Register instruction **JmpR** transfers control to a target address contained in a register. The Jump and Link instruction **JmpL** transfers control to a target address contained in a register after storing a link address in register 4. The link address is formed by adding 2 to the address in Next PC (i.e., the link address is the address of the sequential instruction following the branch shadow instruction). **JmpL** and **JmpR** can be used to implement procedure call and return operations.

The Test Field instruction **TstF** tests a field in a register for zero or negative (high-order bit of the field = "1"), and sets the condition codes accordingly. The rightmost bit position and length of the field are specified by values in the Prefix Register, as described in the next section. The Test Mode instruction **TstM** sets the N condition code to the value of a specified **PsR** bit.

Delayed Branches. All Scorpius implementations have *delayed branches*; the sequential instruction following a branch or jump always is executed, regardless of whether or not the branch is taken. This instruction sometimes is called the *branch shadow* instruction, and is described as being executed "in the shadow of a branch". In most cases, it is possible to place a useful instruction in the branch shadow; in the remaining cases, a NOP instruction must be used. Delayed branching makes it possible to eliminate the pipeline "hole" which otherwise could occur on a taken branch or a jump.

Logical and Shift Instructions. The And, And Complement, Not, Or, and Exclusive Or instructions (**And**, **AndC**, **Not**, **Or**, and **XOr**) perform the indicated logical operation using the contents of registers A and B as operands and store the result in register B. The Shift Left and Shift Right instructions **ShL** and **ShR** perform a logical shift operation, shifting the contents of a register left or right with zero fill. The Shift Double instruction **Dsh** shifts the double word formed by concatenating the contents of registers A and B right a number of bit positions specified by Prefix Register bits <11:7> and stores the lower 32 bits of the result in register A. If A=B, the effect is to rotate the contents of register A right by the number of bits specified in the Prefix Register.

Field Manipulation Instructions. The Clear, Deposit, Extract, Insert, and Set Field instructions, as well as the Test Field instruction described earlier, operate on a field of a register using a field description contained in the Prefix Register. Prefix Register bits <6:2> specify the field length minus one, while bits <11:7> specify the rightmost (low-order) bit position of the field. A field description can be loaded into the Prefix Register via a Prefix Immediate instruction or by a Define Field (**Msk**) instruction; the latter permits the field position to be specified by the contents of a register (i.e., to be computed).

The Clear Field and Set Field instructions **ClrF** and **SetF** clear each bit of a field in a register to "0" or set each bit to "1"; bits outside the field remain unchanged. The Deposit instruction **Dep** extracts a right-justified field from a source register and stores it in a destination register at the specified position, clearing destination register bits outside the field. The Insert instruction **Ins** is similar to Deposit, except that destination register bits outside the field are left unchanged. Extract Signed and Extract Unsigned, **ExtS** and **ExtU** extract a field at a specified position from a source register and store it, right-justified, in a destination register. **ExtS** sign-extends the field according to the field's high-order bit; **ExtU** clears bits outside the field to "0". With an appropriate field definition, **ExtS** instruction can be used to effect an arithmetic right shift.

Arithmetic Instructions. Scorpius provides a complete set of instructions for arithmetic operations on 32-bit integers, as well as add, subtract, and multiply instructions which operate on all four bytes or on both halfwords of their operands. The latter instructions — Add, Subtract, and Multiply Partial — are described in the section on multi-gauge arithmetic later in this chapter. In the current discussion. arithmetic operands are 32-bit signed integers except as noted.

The Add and Subtract instructions **Add** and **Sub** add or subtract the values in registers A and B and store the result in register B. Add with Carry (**AddC**) adds the CO (carry) condition code flag and the value in register A to the value in register B and stores the result in register B. Subtract with Carry (**SubC**) adds the CO condition code flag and the one's complement of the value in register A to the value in register B and stores the result in register B.

The Add Immediate instruction **AddI** adds an immediate value to the value in register. If the Prefix Register is empty, the immediate value is taken from the **AddI** instruction's immediate field. This 8-bit field provides an immediate range of 1-256. If the Prefix Register is not empty, the immediate value is formed by concatenating the value in the Prefix Register with the instruction's immediate field. The Subtract Immediate instruction **SubI** subtracts an immediate in the range 1-16 from the value in a register. **SubI** is not prefixable; to subtract an immediate value greater than that provide by the **SubI** instruction, **AddI** with a negative prefix value is used.

The Count Leading Zeroes instruction **CLZ** counts the number of leading zeroes (the number of consecutive "0" bits counting down from bit <31>) in a register and stores that number in another register.

Divide instructions divide a 32-bit or 64-bit dividend by a 32-bit divisor, producing a quotient which is stored in a general register and a remainder which is stored in special register 5, the Remainder Register (RemR). Dividend and divisor may be either both signed or both unsigned. Divide and Divide Unsigned (**Div** and **DivU**) divide a 32-bit integer in register B by a 32-bit integer in register A, store the quotient in register B, and store the remainder in RemR. Divide Extended and Divide Unsigned Extended (**DivE** and **DivUE**) form a 64-bit dividend by concatenating the contents of the Remainder Register with the contents of register B; the Remainder Register provides bits <63:32> of the dividend (and the sign, for **DivE**), and register B provides dividend bits <31:0>. This extended dividend is divided by value in register A, the quotient is stored in register B and the remainder is stored in RemR.

The Multiply and Multiply Unsigned instructions **Mul** and **MulU** multiply the value in register B by the value in register A and store the resulting 64-bit product in special register 4, the Product Register (ProdR) and register B. The high-order 32 bits of the product (and sign, for **Mul**) are stored in ProdR, while the low-order 32 bits are stored in register B.

The Negate instruction **Neg** generates the two's complement of the value in a register and stores it in another register.

In Antares, multiply and divide instructions are asynchronous operations whose execution can be overlapped with the execution of other instructions. Depending on the amount of overlap possible, the effective cost of a multiply or divide can be as small as one cycle. If a multiple or divide instruction is executing and a subsequently-issued instruction attempts to use the result registers (general or special) of the multiply or divide, that instruction is delayed until the multiply or divide completes.

Broadcast and Semaphore Instructions. Broadcast instructions permit one PU to send data values and activity starting addresses to one or more other PUs, wait for other PUs to complete execution, or halt. The PUs addressed by a broadcast instruction are called the *targets* of that instruction, and are specified by a 4-bit PU Mask field of the broadcast instruction.

The Resume instruction **Rsm** causes each target PU, if halted, to resume execution at the address in the target PU's Current PC. The Resume instruction does not complete execution until all its targets have halted and then resumed. The Start instruction **Strt** sends an address to each halted target PU and causes the target PU to start execution at that address. The Start instruction does not complete execution until all its PUs have halted and then started. The Send instruction sends the value in a register to each target PU; the target PU must execute a Receive (**Rcv**) instruction to receive the value and store it in a register. The Send instruction does not complete execution until all of its targets have received the broadcasted value. If a PU issues a Receive instruction before initiation of a Send instruction, the PU waits for the data value to be sent.

The Wait instruction serves two functions. If a Wait instruction is issued with the issuing PU specified as a target, PU execution is halted and the Halt flag in the PsR is set to "1". If the issuing PU is not specified as a target of the Wait, that PU waits until all PU specified as targets have halted execution and then continues execution.

In addition to using broadcast instructions, PUs can coordinate their activities by means of semaphore instructions. The Global Status Register contains user mode and system mode semaphores (bits <31> and <30>), which are operated on by Lock and Unlock (**Unlk**) instructions. The Lock instruction examines the semaphore determined by the PU's mode (system or user). If the semaphore initially is "1', it is cleared to "0" (locked) and Lock instruction completes. If the semaphore initially is "0", Lock instruction execution waits for the semaphore to be set to "1" (unlocked); the semaphore then is cleared and Lock instruction completes. If several PUs are waiting for the semaphore to be set, one is selected in accordance with implementation-dependent rules. The Unlock instruction sets — unlocks — the appropriate (user mode or system mode) semaphore.

Broadcast and semaphore operations are discussed in Chapter 5.

Cache Control Instructions. Cache control instructions serve three functions. They are used to maintain coherence between the instruction and data caches of a single CPU and between the caches of different CPUs in a multi-

CPU system. They provide a means of flushing instruction and data caches when required, as on a task switch in some implementations (such as Antares). They also can be used to improve performance by eliminating unnecessary transfers of of lines to and from memory, and fetching lines in advance of their use. A brief summary of these instructions is given here; Chapter 6 provides a detailed description.

Cache control instructions, except for **IICA** and **RDTX**, operate on a cache line specified by a memory address in a general register. With the exception of the prefetch instructions, this address can be prefixed. If the Prefix Register is not empty, its contents are added to the value in the register to form the operand address; otherwise, the address in the register is the operand address.

The Create Data Cache line instruction **CDC** is used to avoid transferring a line from memory into the data cache when the initial contents of that line are no longer needed (as when the line will be completely overwritten). **CDC** selects the cache line location of the least-recently-used line in the data cache set specified by its operand address, writes the line currently in that location to memory if it is modified, and creates a new line in that location with the desired address.

The Flush Data Cache line, Invalidate Data Cache line, Update Data Cache line, and Validate Data Cache line instructions (**FDC**, **IDC**, **UDC**, and **VDC**) provide four different ways of disposing of a data cache line. All four do nothing if the addressed line is not in the cache. **FDC** and **UDC** both write the line to memory if it is modified; **FDC** then marks the cache line invalid, while **UDC** marks the cache line unmodified. **IDC** and **VDC** do not write the line to memory if it is modified; **IDC** marks the cache line invalid, while **VDC** marks the cache line unmodified.

The Invalidate Instruction Cache line instruction **IIC** marks the specified cache line invalid, while the Invalid Instruction Cache instruction **IICA** invalidates all instruction cache lines. The Read Data Tag by Index instruction **RDTX** returns the tag associated with each data cache line location 0, 1, 2, ..., *n*-1 (*n* = 64 for Antares). This tag contains the address of the line residing in that location together with valid, modified, and privilege (system/user) flags. In performing a task switch in Antares, which requires flushing the instruction and data caches, the operating system uses **RDTX** to examine data cache line locations and obtain line addresses, and **FDC** to flush and invalidate lines. **RDTX** is implementation-dependent; its operation can differ from one Scorpius CPU version to another, and it may not implemented in all CPUs.

The Prefetch Data Cache line instruction **PDC** is used to prefetch a line from memory in advance of its use to avoid delays. If the line specified by the operand address is not in the data cache, **PDC** initiates a memory read request for the line and completes execution without waiting for the line to be read.

2.5 Prefixing

Prefixing provides an efficient means of extending the intrinsic immediate and displacement fields of Antares instructions, and of providing field descriptions (position and length) for field manipulation instructions. A prefix is constructed in the Prefix Register using one or more Prefix instructions. The Prefix Register (PfxR) is a 32-bit local register, bits <1:0> of which are unused; the Prefix Valid (PV) flag in the Program Status/Control Register (PsR) indicates if the PfxR is empty ("0") or full ("1"). If PV is "0", execution of a Prefix instruction causes that instruction's immediate field to be transferred into PfxR bits <13:2> with sign extension, and PV is set to "1". If PV is "1", execution of a Prefix instruction causes the contents of the PfxR to be shifted left 12 and the instruction's immediate field to be transferred into PfxR bits <13:2> without sign extension. Multiple Prefix instructions can be used to construct prefixes of up to 30 bits in length. A value also can be loaded into the Prefix Register by a Move To Special instruction; however, PV is not set as a result of this instruction. A Set Mode instruction can be used to set PV to "1".

When a prefixable instruction is executed, the PV flag is examined; if it is "1", the contents of the Prefix Register are used to extend or form that instruction's immediate or displacement field, and PV is cleared to "0". However, the Prefix Register contents remain valid and, if desired, PV can be set to "1" again via a Set Mode instruction.

Immediate and displacement fields in Antares instructions are encoded, in most cases, to obtain maximum coverage. For Add and Load Immediate instructions, an immediate value of *i* is encoded in the instruction's immediate field as $i - 1$.[7] When that instruction is executed, the immediate value used in its execution is decoded by extracting the value of the instruction's immediate field and incrementing it by 1. Prefixing causes the contents of the Prefix Register to be concatenated with an instruction's immediate or displacement field to form an effective immediate or displacement; the incrementing carried out in decoding the immediate or displacement field is applied to the concatenation of the Prefix Register and immediate or displacement fields. This can result in a carry from that part of the effective immediate or displacement obtained from the instruction field and that part obtained from the Prefix Register. This carry must be anticipated in specifying the value of a prefix. An immediate or displacement value *v* subject to encoding can be separated into Prefix instruction immediate values and a prefix using instruction immediate or displacement value as follows.[8]

1. Let *k* be the length of the using instruction's immediate or displacement field; the *k* low-order bits of *v* are the value of the

[7] The immediate field of Compare Immediate is not subject to encoding.

[8] Assembler macros, which generate the required number of Prefix instructions with appropriate immediate field values, relieve the programmer of these details.

CPU system. They provide a means of flushing instruction and data caches when required, as on a task switch in some implementations (such as Antares). They also can be used to improve performance by eliminating unnecessary transfers of of lines to and from memory, and fetching lines in advance of their use. A brief summary of these instructions is given here; Chapter 6 provides a detailed description.

Cache control instructions, except for **IICA** and **RDTX**, operate on a cache line specified by a memory address in a general register. With the exception of the prefetch instructions, this address can be prefixed. If the Prefix Register is not empty, its contents are added to the value in the register to form the operand address; otherwise, the address in the register is the operand address.

The Create Data Cache line instruction **CDC** is used to avoid transferring a line from memory into the data cache when the initial contents of that line are no longer needed (as when the line will be completely overwritten). **CDC** selects the cache line location of the least-recently-used line in the data cache set specified by its operand address, writes the line currently in that location to memory if it is modified, and creates a new line in that location with the desired address.

The Flush Data Cache line, Invalidate Data Cache line, Update Data Cache line, and Validate Data Cache line instructions (**FDC**, **IDC**, **UDC**, and **VDC**) provide four different ways of disposing of a data cache line. All four do nothing if the addressed line is not in the cache. **FDC** and **UDC** both write the line to memory if it is modified; **FDC** then marks the cache line invalid, while **UDC** marks the cache line unmodified. **IDC** and **VDC** do not write the line to memory if it is modified; **IDC** marks the cache line invalid, while **VDC** marks the cache line unmodified.

The Invalidate Instruction Cache line instruction **IIC** marks the specified cache line invalid, while the Invalid Instruction Cache instruction **IICA** invalidates all instruction cache lines. The Read Data Tag by Index instruction **RDTX** returns the tag associated with each data cache line location 0, 1, 2, ..., n-1 (n = 64 for Antares). This tag contains the address of the line residing in that location together with valid, modified, and privilege (system/user) flags. In performing a task switch in Antares, which requires flushing the instruction and data caches, the operating system uses **RDTX** to examine data cache line locations and obtain line addresses, and **FDC** to flush and invalidate lines. **RDTX** is implementation-dependent; its operation can differ from one Scorpius CPU version to another, and it may not implemented in all CPUs.

The Prefetch Data Cache line instruction **PDC** is used to prefetch a line from memory in advance of its use to avoid delays. If the line specified by the operand address is not in the data cache, **PDC** initiates a memory read request for the line and completes execution without waiting for the line to be read.

2.5 Prefixing

Prefixing provides an efficient means of extending the intrinsic immediate and displacement fields of Antares instructions, and of providing field descriptions (position and length) for field manipulation instructions. A prefix is constructed in the Prefix Register using one or more Prefix instructions. The Prefix Register (PfxR) is a 32-bit local register, bits <1:0> of which are unused; the Prefix Valid (PV) flag in the Program Status/Control Register (PsR) indicates if the PfxR is empty ("0") or full ("1"). If PV is "0", execution of a Prefix instruction causes that instruction's immediate field to be transferred into PfxR bits <13:2> with sign extension, and PV is set to "1". If PV is "1", execution of a Prefix instruction causes the contents of the PfxR to be shifted left 12 and the instruction's immediate field to be transferred into PfxR bits <13:2> without sign extension. Multiple Prefix instructions can be used to construct prefixes of up to 30 bits in length. A value also can be loaded into the Prefix Register by a Move To Special instruction; however, PV is not set as a result of this instruction. A Set Mode instruction can be used to set PV to "1".

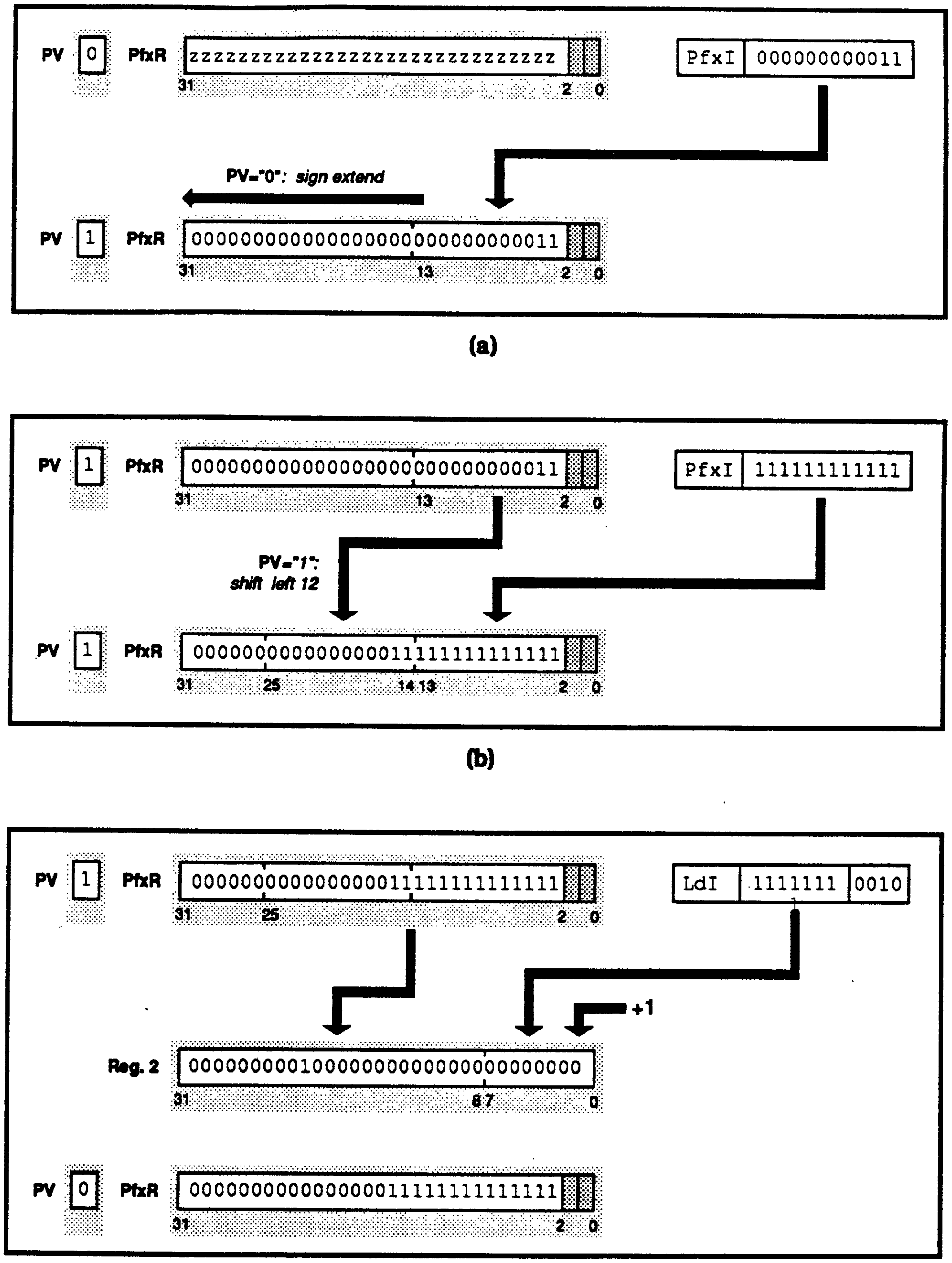
When a prefixable instruction is executed, the PV flag is examined; if it is "1", the contents of the Prefix Register are used to extend or form that instruction's immediate or displacement field, and PV is cleared to "0". However, the Prefix Register contents remain valid and, if desired, PV can be set to "1" again via a Set Mode instruction.

Immediate and displacement fields in Antares instructions are encoded, in most cases, to obtain maximum coverage. For Add and Load Immediate instructions, an immediate value of i is encoded in the instruction's immediate field as $i - 1$.[7] When that instruction is executed, the immediate value used in its execution is decoded by extracting the value of the instruction's immediate field and incrementing it by 1. Prefixing causes the contents of the Prefix Register to be concatenated with an instruction's immediate or displacement field to form an effective immediate or displacement; the incrementing carried out in decoding the immediate or displacement field is applied to the concatenation of the Prefix Register and immediate or displacement fields. This can result in a carry from that part of the effective immediate or displacement obtained from the instruction field and that part obtained from the Prefix Register. This carry must be anticipated in specifying the value of a prefix. An immediate or displacement value v subject to encoding can be separated into Prefix instruction immediate values and a prefix using instruction immediate or displacement value as follows.[8]

1. Let k be the length of the using instruction's immediate or displacement field; the k low-order bits of v are the value of the using instruction's immediate or displacement (prior to encoding)

[7] The immediate field of Compare Immediate is not subject to encoding.

[8] Assembler macros, which generate the required number of Prefix instructions with appropriate immediate field values, relieve the programmer of these details.



(a)

(b)

(c)

Figure 2.12. Immediate Prefixing Example

2. Subtract 1 from v; bits $<k+11:k>$ of $v-1$ compose the value of the immediate field of the Prefix instruction immediately preceding the using instruction.

3. if $|v| > 2^{k+11} - 1$, a second Prefix instruction is required; the immediate field of this instruction comprises bits $<k+23:k+12>$ of $v-1$.

Immediate Prefixing. Add, Load, and Compare Immediate instructions are prefixable; Subtract Immediate is not.[9] When an Add or a Load Immediate instruction is executed, the Prefix Valid flag is examined.If this flag is "1", the effective immediate for the instruction is formed by concatenating the contents of the PfxR with the instruction's immediate field (or, equivalently, shifting the prefix left and adding the immediate) and then adding "1" to the result. Effective immediate bits <31:8> are taken from PfxR bits <25:2> and effective immediate bits <7:0> are taken from the instruction's immediate field. Bit <31> of the effective immediate is treated as a sign bit; bits <31:26> of the PfxR are ignored.

If the Prefix Valid flag is "1" when a Compare Immediate instruction is executed, the effective immediate is formed as described above, except that "1" is not added to the result of concatenating the PfxR contents and instruction's immediate field, since the immediate field of Compare Immediate is not encoded.

As an example of the use of prefixing in extending the range of immediates, suppose it is desired to load the constant 0x400000 into register 2. The Load Immediate instruction has an immediate field of $k = 8$ bits; specifying Prefix (**PfxI**) and Load Immediate (**LdI**) field values according to the process outlined above results in the following instruction sequence.

```
PfxI  0x3
PfxI  0xFFF
LdI   0x00,2
```

The assembler will encode the **LdI** instruction's immediate field as 0xFF. The execution of this instruction sequence, showing the PfxR and PV contents before and after the execution of each instruction (and assuming PV initially is "0"), is illustrated in Figure 2.12. Figures 2.x(a) and (b) show the results of execution of the first and second Prefix instructions. Figure 2.x(c) shows the result of execution of the Load Immediate instruction. Since PV = "1", the effective immediate generated by this instruction is formed by concatenating PfxR bits <25:2> with the instruction's immediate field and adding "1" to the result, as described earlier. As described earlier, this addition effectively takes place after the concatenation so that a carry, if generated, will propagate through the full effective immediate.

[9] Subtract Immediate is not prefixable because prefixes are signed quantities; a prefixed Add Immediate can be used when it is desired to subtract a constant larger than that provided by the immediate field of the Subtract Immediate.

using instruction's immediate or displacement (prior to encoding)

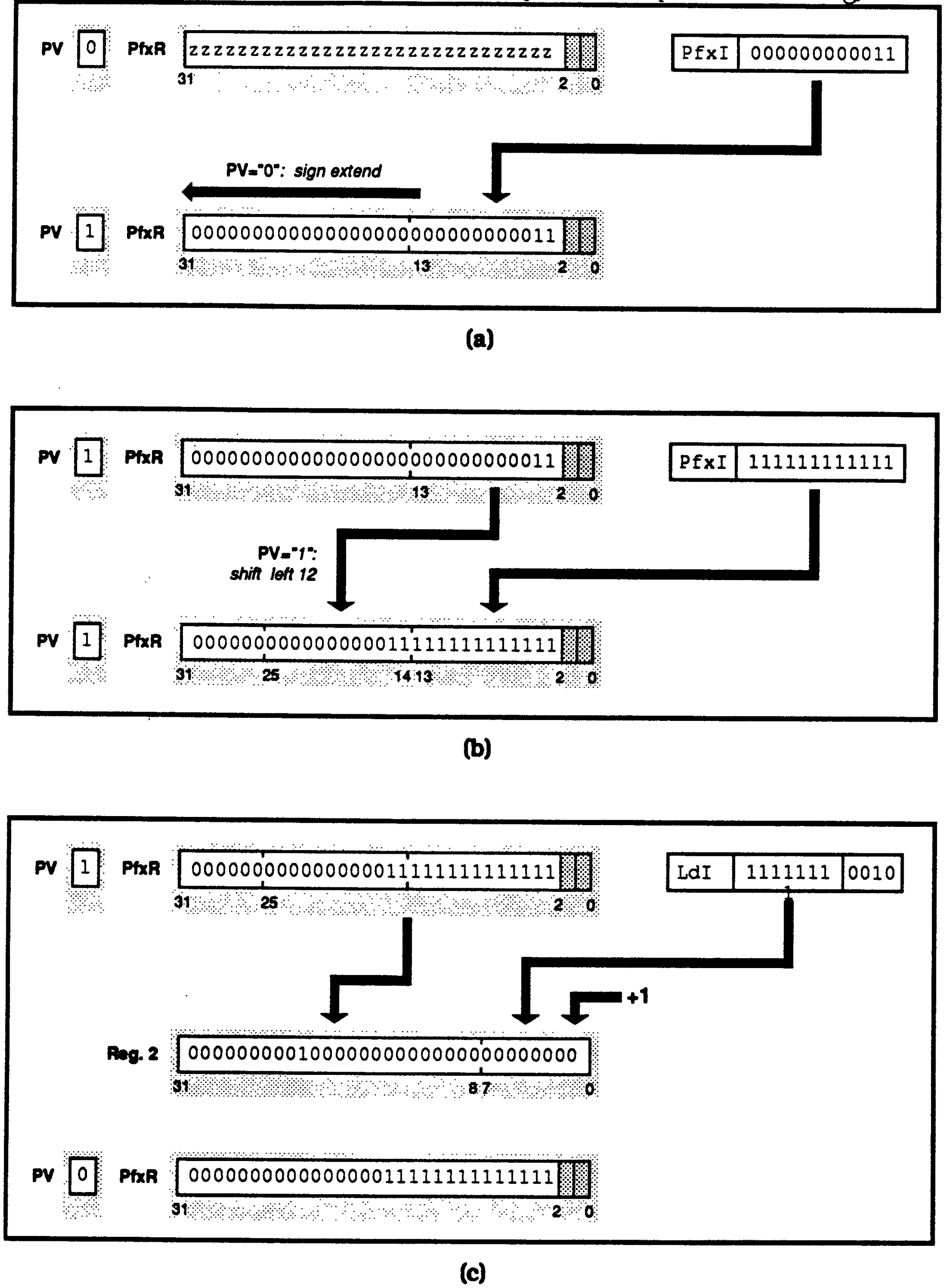


Figure 2.12. Immediate Prefixing Example

2. Subtract 1 from v; bits $<k+11:k>$ of $v - 1$ compose the value of the immediate field of the Prefix instruction immediately preceding the using instruction.

3. if $|v| > 2^{k+11} - 1$, a second Prefix instruction is required; the immediate field of this instruction comprises bits $<k+23:k+12>$ of $v - 1$.

Immediate Prefixing. Add, Load, and Compare Immediate instructions are prefixable; Subtract Immediate is not.[9] When an Add or a Load Immediate instruction is executed, the Prefix Valid flag is examined.If this flag is "1", the effective immediate for the instruction is formed by concatenating the contents of the PfxR with the instruction's immediate field (or, equivalently, shifting the prefix left and adding the immediate) and then adding "1" to the result. Effective immediate bits <31:8> are taken from PfxR bits <25:2> and effective immediate bits <7:0> are taken from the instruction's immediate field. Bit <31> of the effective immediate is treated as a sign bit; bits <31:26> of the PfxR are ignored.

If the Prefix Valid flag is "1" when a Compare Immediate instruction is executed, the effective immediate is formed as described above, except that "1" is not added to the result of concatenating the PfxR contents and instruction's immediate field, since the immediate field of Compare Immediate is not encoded.

As an example of the use of prefixing in extending the range of immediates, suppose it is desired to load the constant 0x400000 into register 2. The Load Immediate instruction has an immediate field of $k = 8$ bits; specifying Prefix (**PfxI**) and Load Immediate (**LdI**) field values according to the process outlined above results in the following instruction sequence.

```
PfxI  0x3
PfxI  0xFFF
LdI   0x00,2
```

The assembler will encode the **LdI** instruction's immediate field as 0xFF. The execution of this instruction sequence, showing the PfxR and PV contents before and after the execution of each instruction (and assuming PV initially is "0"), is illustrated in Figure 2.12. Figures 2.x(a) and (b) show the results of execution of the first and second Prefix instructions. Figure 2.x(c) shows the result of execution of the Load Immediate instruction. Since PV = "1", the effective immediate generated by this instruction is formed by concatenating PfxR bits <25:2> with the instruction's immediate field and adding "1" to the result, as described earlier. As described earlier, this addition effectively takes place after the concatenation so that a carry, if generated, will propagate through the full effective immediate.

[9] Subtract Immediate is not prefixable because prefixes are signed quantities; a prefixed Add Immediate can be used when it is desired to subtract a constant larger than that provided by the immediate field of the Subtract Immediate.

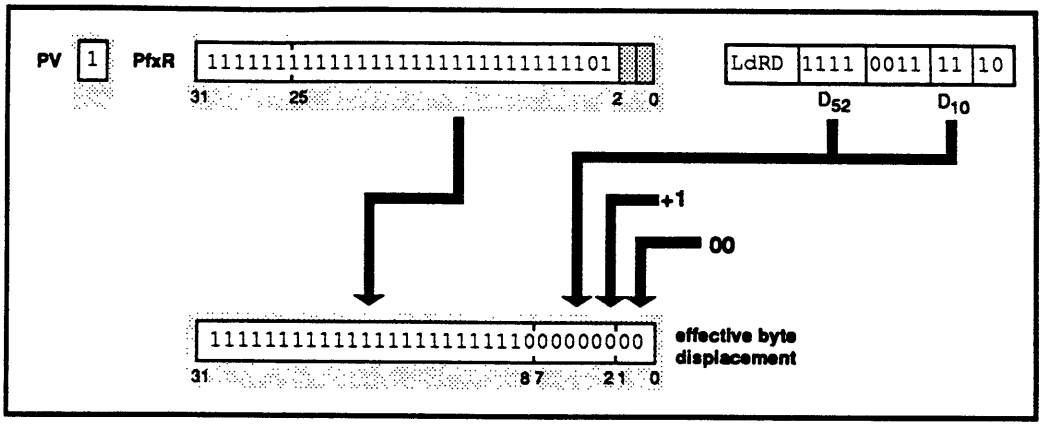


Figure 2.13. Displacement Prefixing Example

Displacement Prefixing. Prefixing can be used to extend the displacement range of **LdRD/StRD** and to permit the use of signed displacements with these two instructions, and to transform **LdR/StR**, cache control, and prefetch instructions from register addressing to base plus displacement addressing.

When a **LdRD/StRD** instruction is executed, the Prefix Valid flag is examined; if this flag is "1", the effective displacement for the instruction is formed by concatenating the contents of the PfxR with the instruction's immediate field, and the Prefix Valid flag is cleared to "0". In effect, the prefix and instruction immediate are concatenated to form a word displacement which is incremented by 1 (since a displacement *d* is encoded as *d* – 1), and then shifted left 2 places to form the required byte displacement. Bits <31:8> of the effective byte displacement are taken from PfxR bits <25:2>, bits <7:2> are taken from the instruction's displacement field, and bits <1:0> are "00". For example, suppose it is desired to execute an **LdRD** instruction with a displacement of –128 words (-128_{10} = 0xF...FF80) from the address in register 2 to load a word into register 3. The **LdRD** displacement field and **PfxI** immediate field values are determined as described earlier. Since the displacement field length of this instruction is *k* = 6 bits in length, the unencoded displacement is 000000_2. –128_{10} – 1 = 0xF...FF7F; extracting bits <17:6> from this gives the **PfxI** immediate value 0xFFD. The desired operation can be performed by the following instruction sequence.

```
PfxI   0xFFD
LdRD   0x00[2],3    ; "0x00" will be encoded as "0xFF"
```

Figure 2.13 illustrates the formation of the effective byte displacement during execution of the **LdRD** instruction (assuming the Prefix Register contents have been established by execution of the above prefix instruction). The effective byte displacement, which will be added to the contents of register 2 to form the operand address, is 0xF...FFE00, or –512_{10}: 4 times the specified word displacement of -128_{10}.

LdR and **StR** instructions, which have no intrinsic displacement, use the contents of the PfxR as a signed displacement if PV = "1". Bits <31:2> of the

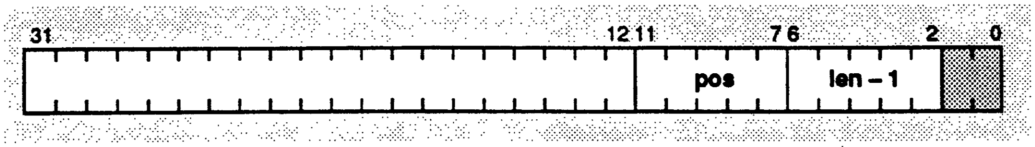


Figure 2.14. Field Description Locations in **PfxR**

effective displacement are taken from PfxR bits <31:2>, and bits <1:0> are "0". Thus, the value loaded into the PfxR by a prefix instruction (or instructions) is an **LdR/StR** word displacement, which is transformed into a byte displacement when used.

Prefixing also permits displacements to be used with certain cache control and prefetch instructions in exactly the same way as **LdR** and **StR**. This usually eliminates the need to use a register to form a prefetch address.

Field Manipulation Instruction Prefixing. The PfxR also is used to hold a field description for the various field manipulation and test instructions (Clear Field, Deposit, Extract, Insert, Set Field, Shift Double, and Test Field). This field description comprises a field length **len** and a rightmost bit position **pos**. Field manipulation instructions use PfxR bits <6:2> as **len** – 1 and PfxR bits <11:7> as **pos** (Figure 2.14). The field description may be loaded into the PfxR by a Prefix instruction or, when **pos** is determined at execution time, by a Mask instruction.

Branch Displacement Prefixing. It is intended that future versions of Scorpius extend prefixing to include relative branch (and jump) displacements. In Antares, trapping on **Bcc/Jmp** when PV = "1" would provide a means of realizing backwards compatibility; programs written for CPUs with branch displacement prefixing could execute on Antares via emulation. However, this presents certain implementation problems, so that the objective has been limited to forward compatibility; programs written for Antares should execute correctly on later Scorpius implementations which have branch prefixing. Forward compatibility requires only that a **Bcc/Jmp** instruction does not appear between the creation of a prefix and its use. For example, in extending the immediate field value of an **AddI** instruction, the sequence **PfxI-AddI -Bcc** is acceptable; the sequence **PfxI-Bcc-AddI** is not. To enforce this usage in Antares, the Prefix Valid flag is cleared whenever a **Bcc/Jmp** instruction is executed.

2.6 Condition Codes

Scorpius provides the traditional four condition codes — Negative (N), Zero (Z), Overflow (V), and Carry (C) — which are set or cleared based on the results of arithmetic and certain other operations. However, Scorpius has <u>four</u> Carry condition codes, C0, C1, C2, and C3, which are set or cleared in various combinations to reflect the results of word, halfword, or byte operations. Condition codes are contained in bits <22:17> of the PsR (Figure 2.5). Figure 2.15

Instructions Which Set Condition Codes				
Add	CLZ	ExtS	Or	SubI
AddC	Cmp	ExtU	RtI	SubP
AddI	CmpI	Ins	Shl	TstF
AddP	CmpP	MovFS	ShR	TstM
And	Dep	Neg	Sub	Xor
AndC	Dsh	Not	SubC	

Instructions Which Do Not Set Condition Codes				
Bcc	IICA	LdR	PfxI	StM
CDC	Jmp	LdRD	PIC	StR
ClrF	JmpL	Lock	Prmpt	StRD
ClrM	JmpR	Mov	Rcv	Strt
Div	Lcc	MovTS	RDTX	Trap
DivE	LdB	Msk	Res	UDC
DivU	LdCP	Mul	Rsm	Unlk
DivUE	LdI	MulP	Send	VDC
FDC	LdM	MulPU	SetF	Wait
IDC	LdPC	MulU	SetM	
IIC	LdPU	PDC	Stb	

Figure 2.15. Instructions Which Set and Do Not Set Condition Codes

divides the Scorpius instruction set into those instructions which set condition codes and those which do not (note that multiply and divide are among the latter). Condition codes are set according to the following rules (for precise definitions of the conditions under which condition codes are set, refer to the individual instruction descriptions in Chapter 8).

N Negative condition code. For arithmetic instructions which store a result, N is set to bit <31> of the stored result. For **Cmp** and **CmpI** instructions, N is set to "1" if the true result is less than zero.[10] For **CmpP**, N is set to the most significant bit of the true halfword or byte arithmetic result. For **TstF**, N is set to "1" if the most significant bit of the field = "1"; for **TstM**, N is set to the value of the specified mode bit.

Z Zero condition code. For word arithmetic operations, Z is set to "1" if the result is zero. For halfword or byte arithmetic operations, Z is set to "1" if either halfword or any byte is zero. Z also is set to "1" if the field operated on by an Deposit, Extract, Insert, or Test Field instruction comprises all "0's".

[10] The true result is the result computed without regard to machine precision.

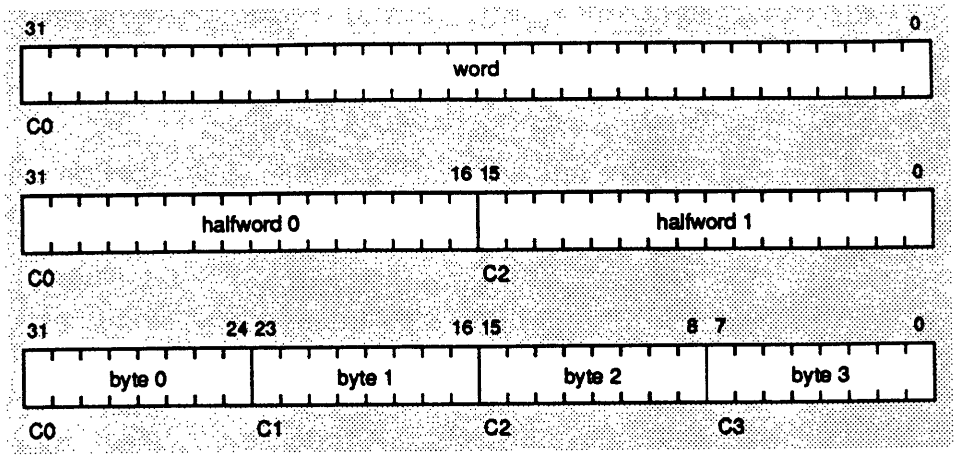


Figure 2.16. Correspondence Between Carry Codes and Operand Lengths

V Overflow condition code. For word arithmetic operations, V is set to "1" if the most significant bit of the result produced by adding two numbers with the same signs differs from the sign of the result, or if the sign of the result produced by subtracting two numbers with unlike signs is different from the sign of the minuend. V is never set by partial arithmetic instructions.

C Carry condition codes, C0-C3. These correspond to word, halfword, and byte operands as shown in Figure 2.16. When a word operation results in a carry, C0 is set to "1" and C1, C2, and C3 are cleared to "0". In halfword mode (**PsR** bit <2> = "1") C0 and C2 are set to "1" if there is a carry from the corresponding halfword, while C1 and C3 cleared to "0". In byte mode, C0, C1, C2, C3 are set to "1" if there is a carry from corresponding byte.

For addition, the carry condition code C is determined from

$$C = (A \& B) \mid ((A \mid B) \& \sim R)$$

where A is the sign of the addend, B is the sign of the augend, and R is the sign of the result.[11] Thus, C is set if both addend and augend are negative, or if either is negative and the sum is positive. For subtraction and comparison, the carry condition code C is determined from

$$C = ((\sim A) \mid (((\sim A) \mid B) \& \sim R)$$

where A is the sign of the minuend, B is the sign of the subtrahend, and R is the sign of the result. C is set if the minuend is positive and the subtrahend is negative. Also, C is set if the result is positive and either the minuend is positive or the subtrahend is negative.

[11] Here, "&" denotes the *and* operation, "|" denotes the *or* operation, and "~" denotes the *not*, or 1's complement, operation.

cc field		*condition code settings*	*interpretation*
encoding	*mnemonic*		
0	F		false (Lcc only)
1	OV	V = 1	overflow
2	LO	C = 0	lower than
3	LT	N = 1	less than
5	EQ	Z = 1	equal
6	LS	C = 0 | Z = 1	less than or same
7	LE	N = 0 | Z = 1	less than or equal
9	NV	V = 0	no overflow
10	HS	C = 1	higher than or same
11	GE	N = 0	greater than or equal
13	NE	Z = 0	not equal
14	HI	C = 1 & Z = 0	higher than
15	GT	N = 0 & Z = 0	greater than

Figure 2.17. **cc** Field Encodings

Condition codes are examined by the conditional branch (**Bcc**) and load condition (**Lcc**) instructions. Also, C0 is an operand of the add performed by the Add with Carry instruction (**AddC**). **Bcc** causes control to be transferred if the value encoded in its **cc** field corresponds to the value of the four condition codes. **Lcc** stores "1" in a result register if the value encoded in its **cc** field corresponds to the value of the four condition codes and "0" otherwise. Figure 2.17 shows the possible encoded values of **cc** and the condition code settings corresponding to those values. For **Bcc**, encodings of 0, 4, 8, and 12 are reserved; for, Lcc, encodings of 4, 8, and 12 are reserved. A **Lcc** instruction with a **cc** field value of 0x0, coded as **LF** and called Load False, always loads "0" into its result register, regardless of the condition code settings (providing a load immediate of 0).

2.7 Multi-Gauge Arithmetic

This section to be added.

3. Addressing

3.1 Introduction

PUs use virtual addresses to access memory for instructions and data. Virtual addresses are 32-bit byte addresses in the range 0x00000000 to 0xFFFFFFFF. For instructions, memory accesses are generated directly by jump (**JmpL/JmpR**), prefetch, and start PU instructions, and indirectly by instruction fetches using the current PC (Program Counter). Instructions must be located on half-word boundaries; the low-order bit of a virtual address used in accessing memory for an instruction is ignored. For data, memory accesses are generated by load register, store register, and prefetch instructions. For word accesses generated by load/store word instructions (**LdR/StR**, **LdRD/StRD**, and **LdM/StM**), the low-order two bits of the address are ignored, so that word loads and stores always are done on word boundaries.

Virtual memory addresses are translated by an address translation mechanism into real addresses, which are used to access memory. (There is no provision for PUs to access memory directly using real addresses.) Real addresses may refer to locations in local memory or remote memory. *Local memory* is memory connected directly to the requesting CPU and accessed via its Memory Bus. *Remote memory* is either memory connected to another CPU (local to that CPU) or memory connected to a NuBus slot; in either case, remote memory is accessed via the Inter-Processor Bus (IPB). The size, composition, and organization of real memory are implementation-dependent. Real memory can comprise DRAM, VRAM, SRAM, and ROM sections, together with various kinds of registers (e.g., control, coprocessor). The organization of real memory in Antares is described in an appendix.

A virtual address space divides into kernel space and user space; the latter can be divided into user and (frame) buffer regions. Only one address space can be active at a time on a given CPU, and the span of control of that address space is local to that CPU. A virtual page of an address space on one CPU may map to a

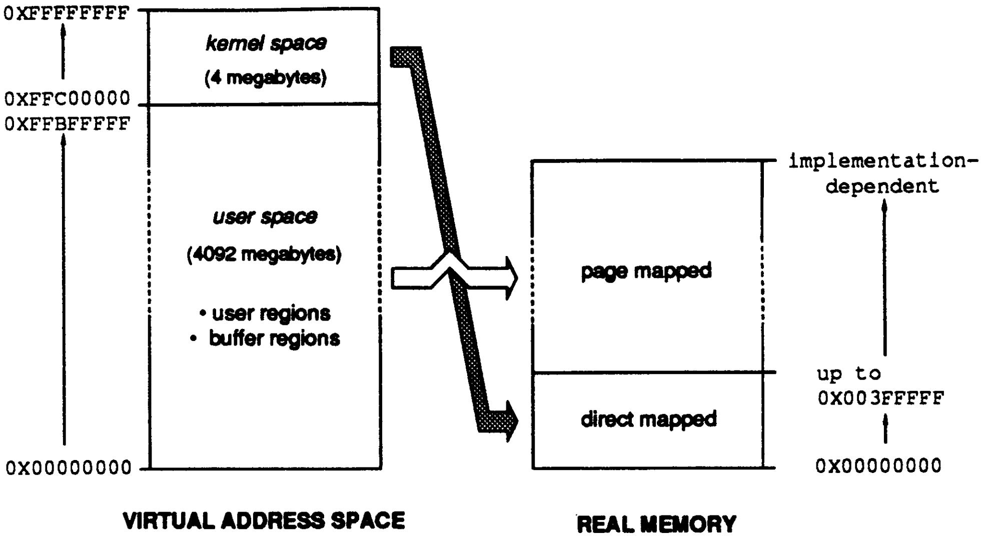


Figure 3.1. Address Space and Real Memory Organization

real memory page located in either local or remote memory. The memory in which the real page resides is said to be the *owner* of the page.

This chapter describes the organization of the Scorpius virtual address space, the privileges which may be associated with parts of that address space, and the translation of virtual addresses into real addresses. Except where noted, addresses are described in terms of byte addresses.

3.2 Address Space Organization

Scorpius provides a flat (unsegmented) virtual address space of 4096 megabytes. A 4-megabyte area at the high end of each address space is allocated for the operating system kernel, and the remaining 4092-megabyte area is available for the user or for parts of the operating system other than the kernel (Figure 3.1). These areas are referred to as *kernel space* and *user space.*

Kernel space is not paged, but instead is directly mapped to the first 4 megabytes of real memory. Any part of this first 4 megabytes of real memory which is not used by the kernel can be allocated for user space pages.

User space is allocated as *user regions* and *buffer regions*; these can appear anywhere in user space. User regions are created for user and system code and data storage, and are allocated in units of a page; the page size is 8192 bytes (8KB). (User space comprises 523776 pages.) User region pages can be specified to be system only or system/user, read only or read/write, cacheable or non-cacheable, or interrupt-on-write. Interrupt-on-write pages are used in inter-node message transmission, discussed later in this chapter. Any virtual page can be mapped to any real page. Starting addresses of both virtual and real pages must fall on a page boundary (address bits 0-12 must be 0).

Buffer regions are created for use as graphics frame buffers, and are allocated in *super-page* units. The size of a super-page can be specified to be 256KB, 512KB, 1MB, 2MB, 4MB, or 8MB, and can be selected so that a single super-page covers the entire frame buffer. The purpose of buffer regions is to reduce the number of Translation Buffer misses generated when drawing lines on the screen. For example, suppose the screen size is 1600 x 1200 pixels with 32-bit — one-word — pixels, so that a frame buffer of almost 7.7MB is needed. If the frame buffer is allocated in the user region, in units of 8KB pages, drawing a vertical line from the top of the screen to the bottom results in accesses to 1200 words in approximately 938 distinct pages, and can be expected to result in 938 Translation Buffer misses. By allocating the frame buffer in a buffer region with a super-page size of 8MB, the Translation Buffer miss rate for frame buffer accesses is effectively reduced to zero. Super-pages can be specified to be system only or system/user, read/only or read/write, or cacheable/non-cacheable. Like user region pages, starting addresses of both virtual and real super-pages must fall on a page boundary corresponding to the super-page size. Allocating the same real page as a user region virtual page and as part of a super-page causes unpredictable results.

It is possible to create more than one buffer region. However, in Antares, a single special Translation Buffer entry is provided for translation of super-page references. Since graphics software typically performs all required operations on one frame buffer before switching to another, the limit of a single Translation Buffer entry for super-pages does not present a performance problem in the intended use of buffer regions.

An address space is defined by a set of virtual to real page mappings which are recorded in a translation table; each address space has its own table. The number of address spaces which may be defined is model-dependent; Antares defines the maximum number of address spaces to be 128. The CPU can access only one address space, called the *active address space*, at a time. An address space becomes active when its *Address Space Number* (ASN) is stored in the ASN field of the ID Register.

Access Privileges. The operating system can specify that certain privileges are to be associated with a user or buffer region page. These privileges are as follows.

- system access only — a "system" page can be accessed only by a PU in system mode (U/S bit in the PsR = "1"). An attempted access by a PU in user mode results in generation of a access privilege violation trap. (A "user" page can be accessed in either system or user mode.)
- read access only — a "read only" page can be accessed only by an instruction or data fetch; an attempted store access results in generation of a data access privilege violation trap.

• non-cacheable	an access to a "non-cacheable" page causes the addressed word to be directly read from or written to memory. Only load/store word data accesses and instruction fetch accesses are permitted to a non-cacheable page; an attempted byte access (via **LdB/StB**) results in generation of a data access privilege violation trap.
• interrupt-on-write	a store access to an "interrupt-on-write" page causes a Message Interrupt to be presented to the owner of that page, as described below. This privilege is available only for user region pages only.

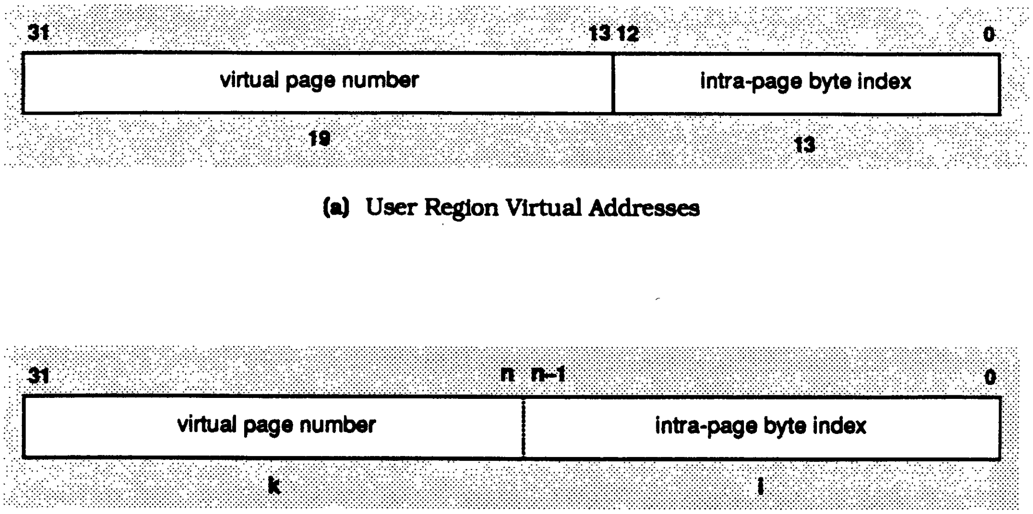
Access privileges for a page are established by setting the appropriate flags in the translation table entry for that page.

Inter-Node Messaging via Interrupt-on-Write Pages. An attempted store to an "interrupt-on-write" page by a store word instruction causes a Message Interrupt to be presented to the node owning that page, called the *destination* node, or simply *destination.*[1] The destination usually, but not necessarily, is a different node than the node at which the store instruction is executed. If that interrupt can be accepted by the owner node, the message word is stored at the appropriate location in the owner's local memory, and the real intra-node byte address of the word referenced as the operand of the store instruction is stored in the Interrupt Argument Register (IAR) of the owner node. If the owner node has external interrupt presentation disabled, or if a Message Interrupt already is pending at the owner node, the newly-presented Message Interrupt is rejected and the message word is not stored. Rejection of a Message Interrupt causes a Message Reject trap to be generated on the PU attempting to execute the store instruction, unless that PU is interrupt/trap disabled; in this case, a PU Check trap, rather than a Message Reject trap, is generated. (Consequently, a PU should send a message — store to an interrupt-on-write page — only while interrupt/trap enabled) The Message interrupt and the Message Reject trap are discussed in Chapter 4.

The kernel must decide how to deal with rejected messages. In a small configuration, it may simply reinitiate execution of the store instruction. In a large configuration, it may use some adaptive (e.g., backoff) algorithm to determine when another attempt to send a message should be made.

A page marked interrupt-on-write must also be marked non-cacheable. If it is not, a data page fault trap is generated when any access to the page results in

[1] This store instruction typically is an **StR** or **StRD** instruction. While execution of an **StM** instruction to an interrupt-on-write page is not an error per se, a Message Interrupt is presented for each word stored, and there is a high probability that stores after the first will result in a Message Reject trap; the **StM** may never complete.



(a) User Region Virtual Addresses

super-page size	rightmost bit position of virtual page number (n)	virtual page number field length (k)	intra-page byte index field length (i)
256KB	18	14	18
512KB	19	13	19
1MB	20	12	20
2MB	21	11	21
4MB	22	10	22
8MB	23	9	23

(b) Buffer Region Virtual Addresses

Figure 3.2. Virtual Address Formats

an address translation which must read the corresponding translation table entry. The requirement that the interrupt-on-write page also be non-cacheable is one of the reasons for inhibiting store byte access to an interrupt-on-write page.[2]

Interrupt-on-write pages provide a basic mechanism for transmitting messages between nodes and, since interrupt processing can be serialized, for coordinating activities at different nodes. (It is the *only* mechanism provided for synchronization of multiple processors.) A variety of message schemes based on interrupt-on-write pages are possible.

[2] Later versions of Scorpius may permit interrupt-on-write pages to be cacheable, as well as non-cacheable, so that message lengths of a line (64 bytes) can be accommodated. To do this safely will require the capability to lock a line into the cache (to prevent inadvertent moveout of an incomplete message).

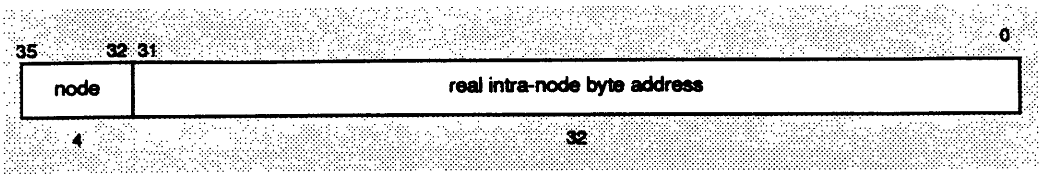


Figure 3.3. Real Address Format

3.3 Address Formats

Virtual Addresses. A virtual address is the concatenation of a *virtual page number* and an intra-page byte index (Figure 3.2). For user region virtual addresses, the virtual page number is 19 bits in length and the intra-page index is 13 bits in length. On instruction fetches, the low-order bit of the virtual address is ignored. On word loads and stores, the low-order two bits of the virtual address are ignored. Virtual page numbers (of 8KB pages) in the range 0X00000 to 0X7FDFF correspond to user space virtual addresses, and are translated into real page numbers using the address translation process described later in this chapter. Virtual page numbers in the range 0X7FE00 to 0X7FFFF correspond to kernel space virtual addresses, which are directly mapped to real addresses. The real address corresponding to a kernel region virtual address can be obtained by subtracting 0X7FE00 from the virtual page number field of the address (retaining the intra-page index). For buffer region virtual addresses, the lengths of the virtual super-page number and intra-page index depend on the super-page size, and are listed in Figure 3.2(b).

Address Arithmetic. Address arithmetic (as in forming an effective address from base register and displacement values) is performed modulo 2^{32}. An attempt to generate an address greater than $2^{32} - 1$ does not cause an exception, but results in an address which "wraps around" the address space.

Real Addresses. A real address comprises a 4-bit node number and a 32-bit intra-node real byte address (Figure 3.3). The intra-node address specifies a location in memory local to (connected directly to) the node specified by the node number. The node associated with the real address resulting from the translation of a virtual address is said to be the owner of the real page referenced by that address.

A real address resulting from the translation of a kernel space virtual address always has the same node number as the CPU on which the address was generated, and so always is directed to local memory. The node number of a real address resulting from the translation of a user space virtual address may be the same as that of the CPU generating the access, in which case the access is directed to local memory, or may differ from the node number of the CPU generating the memory access, in which case the access is directed to the appropriate remote memory via the Inter-Processor Bus. In Antares-based systems, all off-chip real addresses are word addresses, comprising a 4-bit node number and a 30-bit intra-node real word address.

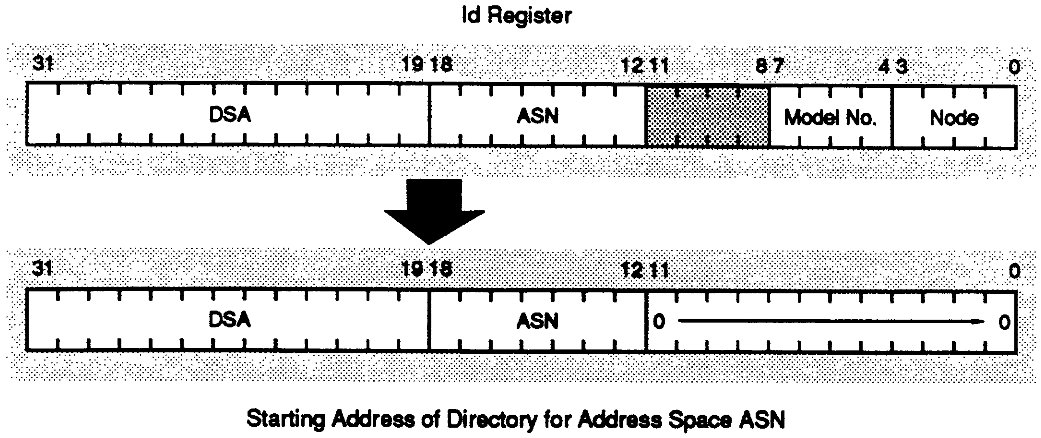


Figure 3.4. Forming a Directory Starting Address

3.4 Translation Tables

Structure. A user space virtual address in the currently-active address space is translated to a real address by hardware via a simple two-level translation table. The first level is called the *directory*; it is fixed-length, and comprises 1024 one-word entries. A directory entry represents a 4MB segment of the virtual address space defined by the table.[3] Each entry, if valid, specifies the address of a second level *page table*. Entries 0 - 1022 span user space; entry 1023 corresponds to kernel space and is never examined by hardware. Page tables are fixed length, each comprising 512 one-word entries. User region and buffer region page tables have the same structure. A page table entry, if valid, specifies the real page number associated with a virtual address.

Directories are maintained in an area of real memory called Directories Space. The real starting address of this area is defined by the Directories Starting Address (DSA) field in the Id Register (IdR). Normally, the DSA is set during system initialization and remains unchanged thereafter. The real address of the first word of the directory for the currently-active address space is defined by the concatenation of the DSA and the current ASN, as shown in Figure 3.4. Antares permits up to 128 address spaces, numbered 0 - 127, to be defined.[4] Page tables must begin on 512-word boundaries. Directories for address spaces 0, 1, ..., 127, must be stored contiguously and on 1024-word boundaries; Directories Space must start on a 128K-word (512KB) boundary.

[3] A segment is defined as the 4MB portion of a virtual address space represented by a directory entry; it has no architectural definition beyond that.

[4] The ASN field is defined for compatibility reasons. It is not used or interpreted by Antares, except that the concatenation of the DSA and ASN fields is used to form the real address of a directory entry. Later versions of Scorpius may incorporate the ASN field in the cache tag (to eliminate cache flushing on an address space switch).

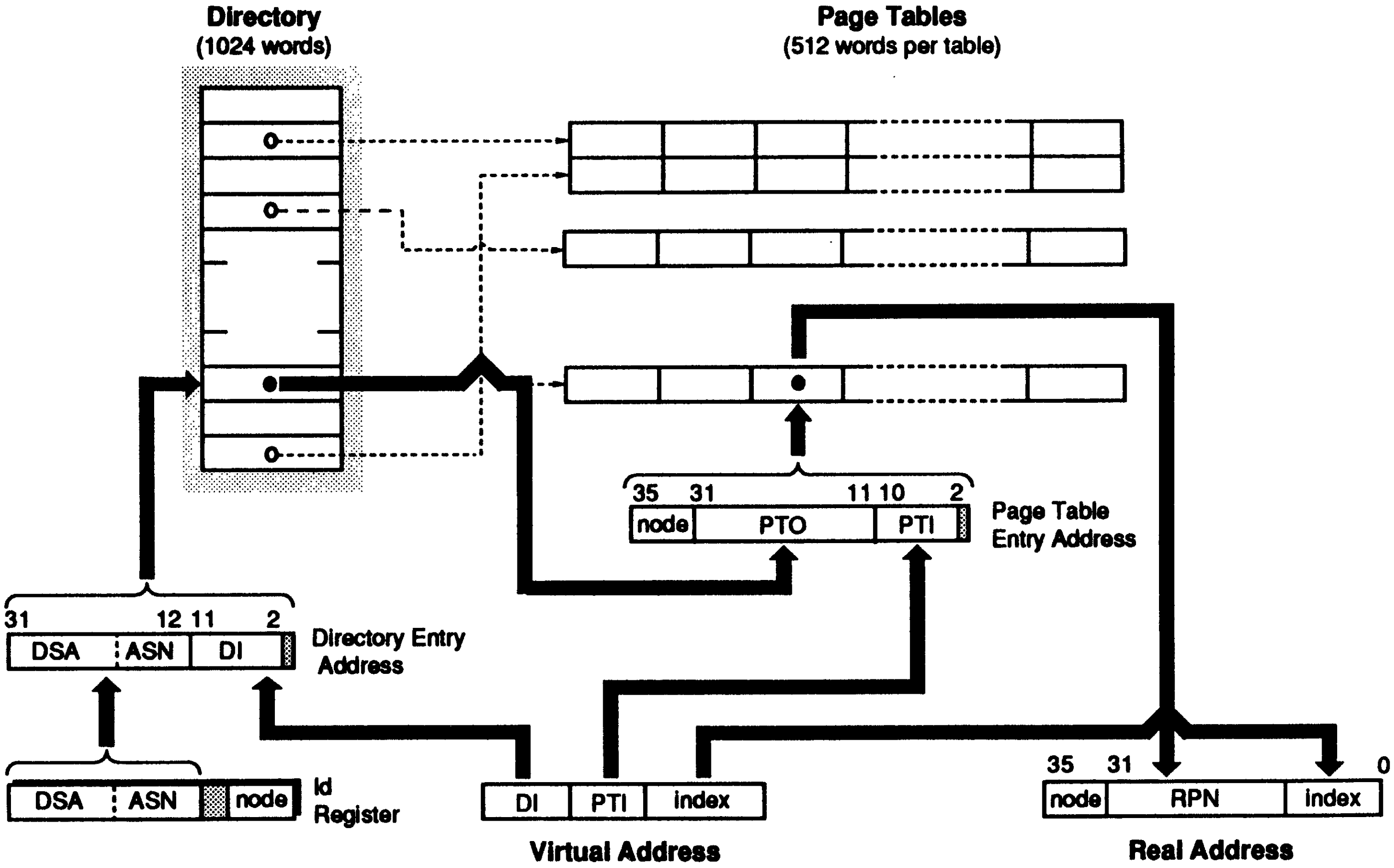
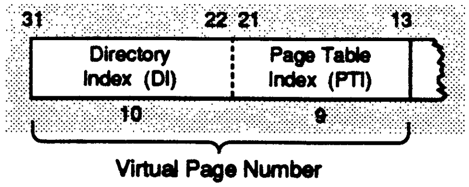


Figure 3.5. Translation Table Structure and Accessing

For address translation purposes, a 19-bit virtual page number divides into a 10-bit directory index and a 9-bit page table index, as shown below.



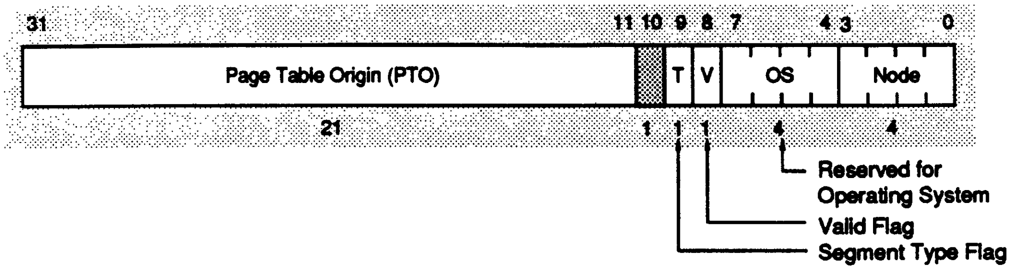
In translating a virtual page number of either a user or buffer region page into a real page number, the real address of a directory entry word is formed by concatenating the DSA and ASN fields from the Id Register with the DI field of the virtual page number, as shown in Figure 3.5. A flag in the directory entry indicates whether or not the corresponding segment is defined; if it is defined, the entry provides bits <31:11> (called the Page Table Origin or PTO) of the starting address of the page table for that segment together with a node number specifying the location of the memory containing the page table. If the node number from the directory entry matches the node number in the IdR, the page table resides in local memory; otherwise, the page table resides in remote memory.

The real intra-node address of the page table entry for a virtual page is formed by concatenating the PTO field from the directory entry with the PTI field from the virtual page number. A flag in the page table entry indicates whether or not a real memory page is associated with that virtual page; if it is, the entry provides the real page number of that page together with the node number of its owner. If the owner's node number matches the node number in the **Idr**, the real page is in local memory; otherwise, it is in remote memory.

Entry Formats. Figures 3.6 - 3.8 show the formats of directory and page table entries and describe the fields of these entries. Translation table entry accessing is discussed below; a later section provides a step-by-step description of the translation process.

The translation of a virtual address to a real address is facilitated via a Translation Buffer or a Translation Lookaside Buffer (depending on the implementation) which holds recently-used virtual-to-real page mappings. This buffer, referred to here as the TB/TLB, is described later in this chapter. The virtual-to-real page mapping held in a TB/TLB entry comprises a virtual page number and the corresponding page table entry; this mapping sometimes is simply called a *translation*. In translating a virtual address, the virtual page number is extracted from the address and compared with the virtual page numbers in each of the TB/TLB entries in which a translation for that address could appear. If a match is found, the real page number is extracted from the TB/TLB entry and used to form the real address. For a TB/TLB entry representing a buffer region page, the SPS field from the entry determines the number of bits used in this comparison.

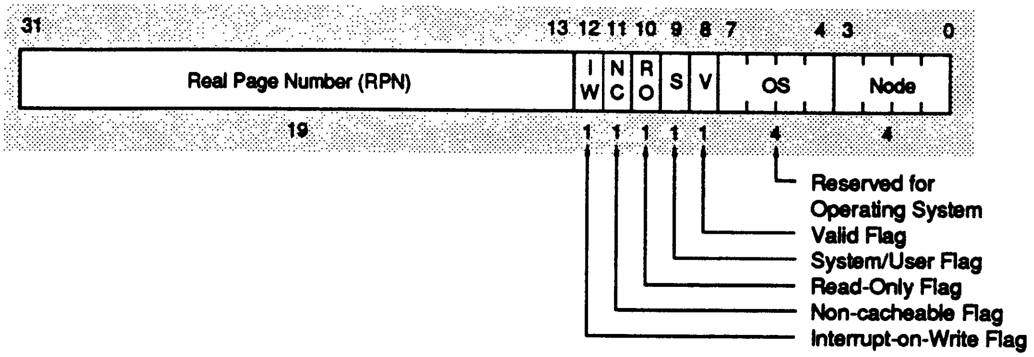
Directory Entry Format



name	bit position(s)	length	description
PTO	<31:11>	21	Page table origin: contains bits <31:11> of real intra-node addresses of entries in the page table for the segment defined by this entry.
—	<10>	1	Reserved for future use.
T	<9>	1	Segment type flag: if T = "0", the segment defined by this entry is in a user region. If T = "1", the segment defined by this entry is in a buffer region.
V	<8>	1	Valid flag. If V = "1", this is a valid directory entry specifying the start of the page table for the segment defined by this entry. If V = "0", this entry is invalid and the contents of the remaining fields are unpredictable. Attempted access to an invalid page causes an instruction or a data page fault trap to be generated.
OS	<7:4>	4	Reserved for use by the operating system (not examined by hardware)
Node	<3:0>	4	Node number. Number of the node at which the page table specified by this entry resides.

Figure 3.6. Directory Entry Format

User Region Page Table Entry Format



name	bit position(s)	length	description
RPN	<31:13>	19	Real page number: contains bits <31:13> of real addresses of bytes in this page.
IW	<12>	1	Interrupt-on-write flag. IF IW = "1", an attempted store to this page causes a Message interrupt to be presented to the node owning this page. No interrupt is presented if IW = "0". If IW = "1", NC also must be "1"; otherwise, a page fault trap is generated.
NC	<11>	1	Non-cacheable flag. If NC = "1", lines of this page are not stored in the cache; any access to this page causes the addressed word to be read from or written to memory. If NC = "0", lines of this page are cached, and transfers between the CPU and memory take place in units of one line. Attempted execution of a LdB/StB operand access to this page causes an access privilege violation trap to be generated.
RO	<10>	1	Read-only flag. If RO = "1", only read access is permitted to this page; an attempted store will cause an access privilege violation trap to be generated. If RO = "0", read and write access are are permitted.

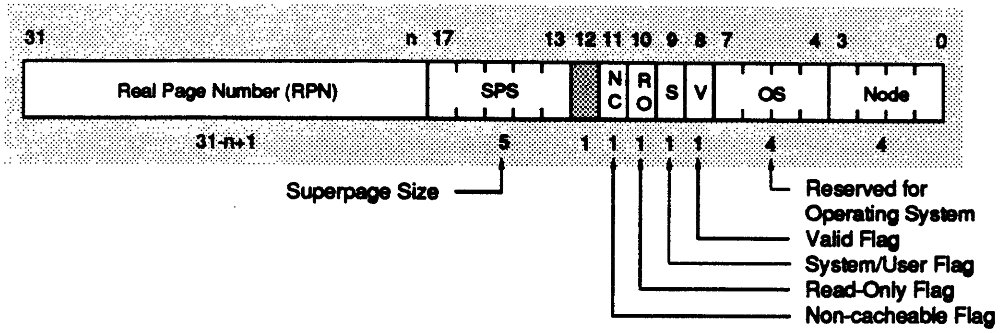
Figure 3.7. User Region Page Table Entry Format

User Region Page Table Entry Format (continued)

name	bit position(s)	length	description
S	<9>	1	System/user flag. If S = "1", access to this page is permitted only to PUs in system mode (PsR bit 13 = "0"). Attempted access to a system page while in user mode causes an access privilege violation trap to be generated. If S = "0", access is permitted regardless of the PU mode.
V	<8>	1	Valid flag. If V = "1", this is a valid entry specifying a real page number, its owner node, and the access privileges associated with the page. If V = "0", the entry is invalid and the contents of the other fields are unpredictable. Attempted access to an invalid page causes an instruction or a data page fault trap to be generated.
OS	<7:4>	4	Reserved for use by the operating system (not examined by hardware.)
Node	<3:0>	4	Node number. Number of the node owning the real page specified by this entry.

Figure 3.7. User Region Page Table Entry Format (continued)

Buffer Region Page Table Entry Format



name	bit position(s)	length	description
RPN	<31:n>	31-n+1	Real page number; contains bits <31:*n*> of real addresses of bytes in this page (left-justified and zero-filled). *n* depends on the super-page size (see the SPS field, below).
SPS	<17:13>	5	Super-page size. This field specifies the super-page size and the right-most bit position of the real page number, as shown below

SPS	super-page size	n
"00000"	256KB	18
"00001"	512KB	19
"00011"	1MB	20
"00111"	2MB	21
"01111"	4MB	22
"11111"	8MB	23

Setting the SPS field to an invalid value (a bit combination other than those listed above) will not generate an exception, but will cause unpredictable results.

name	bit position(s)	length	description
—	<12>	1	Reserved for future use.
NC	<11>	1	Non-cacheable flag. If NC = "1", lines of this page are not stored in the cache; any access to this page causes the addressed word to be read

Figure 3.8. Buffer Region Page Table Entry Format

Buffer Region Page Table Entry Format (continued)

name	bit position(s)	length	description
			from or written to memory. If NC = "0", lines of this page are cached, and transfers between the CPU and memory take place in units of one line. Attempted execution of a LdB/StB operand access to this page causes an access privilege violation trap to be generated.
RO	<10>	1	Read-only flag. If RO = "1", only read access is permitted to this page; an attempted store will cause an access privilege violation trap to be generated. If RO = "0", read and write access are permitted.
S	<9>	1	System/user flag. If S = "1", access to this page is permitted only to PUs in system mode (PsR bit 13 = "0"). Attempted access to a system page while in user mode causes an access privilege violation trap to be generated. If S = "0", access is permitted regardless of the PU mode.
V	<8>	1	Valid flag. If V = "1", this is a valid entry specifying a real page number, its owner node,and the access privileges associated with the page. f V = "0", the entry is invalid and the contents of the other fields are unpredictable. Attempted access to an invalid page causes an instruction or a data page fault trap to be generated.
OS	<7:4>	4	Reserved for use by the operating system (not examined by hardware.)
Node	<3:0>	4	Node number. Number of the node owning the real page specified by this entry.

Figure 3.8. Buffer Region Page Table Entry Format (continued)

To translate a virtual address for which no translation is found in the TB/TLB, the CPU first reads the directory entry corresponding to the segment to which the virtual page belongs; the address of this entry is formed by the concatenation of the DSA and ASN fields from the Id Register and the DI field of the virtual page number, as shown in Figure 3.5. Directory entry read requests always are directed to local memory. In Antares, the CPU maintains a buffer of recently used directory entries to speed translation by eliminating the directory entry read operation. (As discussed in a later section, the operating system is responsible for insuring that this buffer is invalidated when changes are made in directory entries.) If the directory entry has V = "0" (entry not valid), a trap is generated. If the access is an instruction fetch, this is an instruction page fault trap; if the access is a data fetch or store, this is a data page fault.

For implementation simplicity, directory and page table structures are the same for both user region and buffer region pages. A directory entry always represents a 4MB segment of the virtual address space, regardless of whether that segment is a user region segment with 8KB pages or a buffer region segment with super-pages (which can be larger in size than the segment). The page table for a buffer region segment always is the same length as that for a user region segment (512 entries), even though the buffer region maps to a single real super-page. (The translation table representation for a buffer region is discussed later in this section.)

The real memory address of the page table entry for the page being accessed is formed by concatenating the Node and PTO fields from the directory entry with bits <21:13> of the virtual address — the PTI field of the virtual page number. Thus, the page table always is accessed as if the page size were 8KB. If the Node field of the directory entry matches the Node field of the ID Register of the CPU initiating the access, the page table resides in local memory, and a read request for the entry word is sent to local memory. If the directory entry Node field does not match the Id Register Node field, the page table block resides in remote memory, and a read request for the entry word is sent to the specified node over the IPB.

After reading the entry from local or remote memory, the entry's V flag is examined and, if "0", either an instruction or a data page fault trap, depending on the access type, is generated. If the entry is valid, it is checked to see if it is legal. If it is, a TB/TLB entry, comprising the virtual page number and its page table entry, is made. An access privilege check of the memory access is made and, if the access is legal, the address translation is completed by forming the real memory address. This is the concatenation of the Node and RPN fields from the page table entry and the intra-page byte index from the virtual address. The memory read or write operation is then initiated.

Page Faults. A page fault trap results from an <u>invalid</u> or <u>illegal</u> <u>entry</u> in the translation table. The following flag combinations are illegal, and result in the generation of an instruction or a data page fault, depending on the access

RO	NC	IW
"0"	"0"	"1"
"1"	"0"	"1"
"1"	"1"	"1"

type. Generation of a page fault trap can occur only when translating an address not found in the TB/TLB; i.e., in the process of making a TB/TLB entry.

Access Privilege Violations. An access privilege violation trap results from an illegal access to a valid page table entry, and can occur on any access. The following access privilege violations can occur.

- store access to a page with RO = "1"
- load/store byte operand access to a page with NC = "1"
- user mode access to a page with S = "1"

The first two violations result in generation of a data access privilege trap. User mode access to a system page results in generation of either a data access privilege violation trap or an instruction access privilege violation trap, depending on the type of access.

While detection of an invalid/illegal translation table entry, or of an illegal access, results in the generation of a trap, that trap will not be recognized if the PU generating the trap is interrupt/trap disabled; instead, a PU Check Trap is generated (Section 4.4). Kernel region virtual addresses are translated to real addresses directly and are not represented by page table or TB/TLB entries. Consequently, memory accesses to kernel region addresses cannot cause page faults or access privilege violations (except that a user mode access to the kernel region will result in an access privilege violation).

Page table entries are never modified by the hardware.

Buffer Region Table Organization. A buffer region is created by allocating contiguous real memory space in the amount dictated by the super-page size (*n* bytes) and on a super-page boundary. Typically, the real memory allocated to this region is memory constructed from VRAMs. For each 4MB segment in the buffer region, a directory entry is constructed with the Segment Type (T) flag set to "1". For each such directory entry, a page table of 512 words is created. For each bound super-page in the segment represented by a page table, *n*/8192 buffer region page table entries with the super-page size field set to the super-page size are created, all pointing to the same real super-page. (The buffer region commonly is represented by a single super-page, so that all page table entries for the real memory allocated to the buffer region are identical.) Page table entries corresponding to unallocated space are, as usual, marked invalid. When the super-page size is greater than the segment size of 4MB, multiple directory entries, all pointing to the same page table, are created.

As an example, suppose a 2MB buffer region comprising a single super-page is to be created in the upper half of segment z of user space ($0 \leq z \leq 1022$); the lower half of segment z is not allocated. This virtual super-page is to be bound to

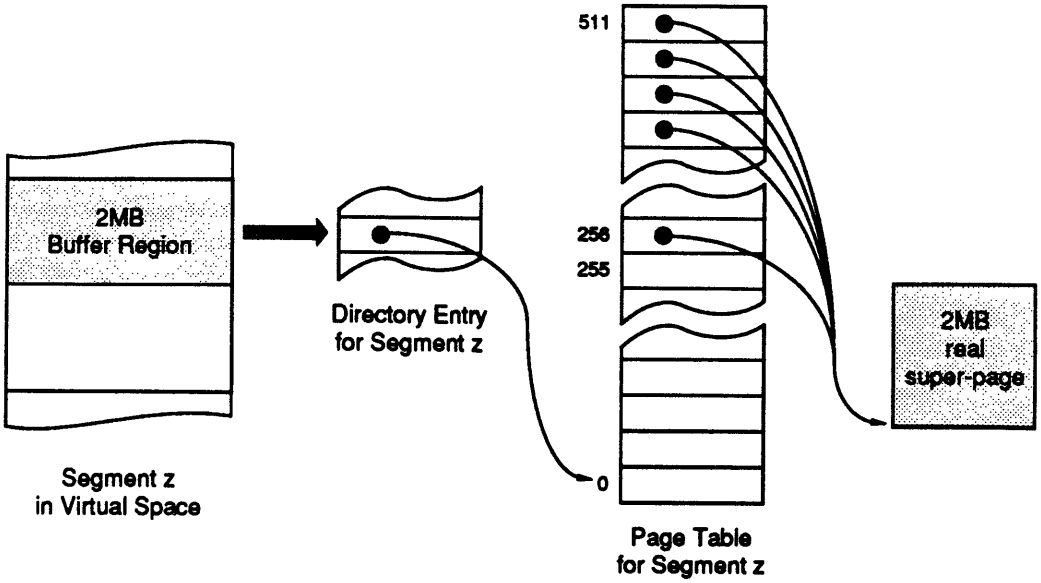


Figure 3.9. Buffer Region Example

a 2MB block of real memory composed of VRAM. The operating system creates a directory entry for segment z with T = "1", allocates a 512-word block of memory for the page table for segment z, and stores bits <31:11> of the starting address of the page table in the directory entry. The buffer region is presented by a single 2MB super-page in the upper half of the segment, and a 2MB page corresponds to 256 8KB pages, so entries 256 - 511 of the page table are created identically; each of these entries has SPS = "00111" and RPN = bits <31:21> of the starting address of the real super-page. Page table entries 0 - 255 all are marked invalid. The resulting translation tables are illustrated in Figure 3.9.

When the initial access to this newly-created buffer region is made, a TB/TLB miss will occur and the associated virtual address is translated in exactly the same way as a user region virtual address. The DI field (address bits <31:22>) is concatenated with the DSA and ASN fields from the IdR to obtain the directory entry address. The directory entry is read and its PTO field extracted and concatenated with the PTI field from the virtual address (address bits <21:13>) to form the address of a page table entry, and the page table entry is read. Because the directory entry has T = "1", the page table entry is recognized as a super-page descriptor; the SPS field is used to select the bits from the virtual address to be used as the intra-page byte index, bits from the virtual address to be used as the virtual page number, and bits from the page table entry to be used as the real page number. Also, T = "1" causes the virtual page number and the page table entry to be stored in a special buffer region TB/TLB entry.[5] Since all 256 entries in that

[5] Antares provides a single buffer region TB entry.

part of the page table which represents the buffer region are identical, the contents of this TB/TLB entry will be the same regardless of where in the buffer region this initial access falls.

Whenever an address must be translated, this special buffer region TB/TLB entry is checked (as well as normal entries). If the entry is valid, the SPS field is used to select which bits from the virtual address are to be compared with the virtual page number stored in the entry. If the result of this comparison is a match, the SPS field is used to select the bits from the virtual address to be used as the intra-page byte index and the bits from the TB/TLB entry to be used as the real page number. (The translation process is described in detail in a later section.)

If the bits comprising the super-page size (SPS) field in the page table entry are set to an invalid combination, no error is detected by the hardware but unpredictable operation can result. Also, all bound space in a segment whose directory entry has T = "1" must be allocated in super-page units. It is the responsibility of the operating system to insure that the SPS field is valid and that segment composition is consistent.

3.5 Translation Table Placement

Translation table entries are read by the MMU using real addresses. The directory entry address formed by concatenating the DSA|ASN field from the IdR with the DI field from the virtual page number is a real word address (to which "00_2" is appended to form a byte address). Directories always reside in local memory (i.e., at the node specified in IdR bits <3:0>). The page table entry address formed by concatenating the Node and PTO fields from a directory entry with the PTI field from the virtual page number (and appending "00_2") also is a real address. Page tables may reside in either local or remote memory.

There are no restrictions on the placement of translation tables in real memory, except that directories must be contiguous, with the directory for address space 0 starting on a 512KB boundary, and page tables must start on a 2048-byte boundaries.

Translation tables may be stored completely in kernel space (if space permits), stored partly in kernel space and partly in user space, or stored completely in user space. Software access to translation tables must use virtual addresses. For translation table elements stored in the kernel region, the virtual word address of an element can be formed by inserting 0XFFC in bits <31:22> of the real address of that element. Translation table elements stored in the user region must be accessed via translation tables; the operating system must construct and maintain the necessary virtual and real mappings. Since user space access is possible only via the translation process, the construction of translation tables in user space requires a "bootstrap" process; a translation table must first be constructed in kernel space to provide access to user space. When construction of user space tables is complete, the table in kernel space can be deallocated.

3.6 The Translation (Lookaside) Buffer

The translation of a virtual page number into a real page number is speeded using a hardware mechanism called the Translation Buffer (TB) or Translation Lookaside Buffer (TLB). The distinction depends on how the cache is addressed in a particular Scorpius implementation. When the cache is accessed with virtual addresses, as in Antares, address translation is done only when the accessed line is not found in the cache; in this case, the translation mechanism is called a Translation Buffer. When the cache is accessed with real addresses, address translation is required on every cache access; in this case, translation usually is done in parallel with the cache access and the translation mechanism is called the Translation Lookaside Buffer. In this specification, when the distinction is not important, the translation mechanism is referred to as the TB/TLB.

The TB/TLB holds the most recently used address translations, just as the data cache holds the most recently used data. If the translation for a virtual address is found in the TB/TLB, it is not necessary to read the directory entry and page table entry from memory. A TB/TLB entry contains a virtual page number and a copy of most or all of the fields of the corresponding page table entry.[6] In translating a virtual address, the TB/TLB is searched for the virtual page number; if found, the real page number is extracted from the TB/TLB entry and used to form the real address. If the virtual page number is not found, the directory entry is read and used to find the page table entry, the page table entry is read and checked for validity, a new TB/TLB entry is made, and the translation process resumed. The new entry will replace an existing entry; the set of entries considered for replacement depend on the implementation. If this set has an entry marked invalid, that entry is replaced; otherwise, an entry is selected using an implementation-dependent algorithm.

In addition to providing a set of entries to hold user region address translations, the TB/TLB provides at least one entry to hold the latest buffer region address translation. (Also, the translation of kernel addresses may be mechanized by a "hardwired" TB/TLB entry.)

The operating system is responsible for insuring that the translations maintained in the TB/TLB are valid. Translations currently held in the TB/TLB may become invalid because of operating system actions such as address space deactivation on task termination, page remapping or state changes (e.g., read-only -> read/write), or address space switching (if ASNs are not kept in TB/TLB entries). Any modification to a translation table directory or page table entry may require invalidation of a TB/TLB entry. When the TB/TLB holds translations for only the currently-active address space, then translation table entries for inactive address spaces may be changed without requiring TB/TLB entry invalidation. Each Scorpius implementation provides a means of in-

[6] In some implementations (although not in Antares), the TB/TLB entry also may contain the Address Space Number of the address space from whose page table the entry was taken.

validating TB/TLB entries together with any supporting buffers; details depend on the implementation.

The Antares Translation Buffer. Antares has virtual instruction and data caches, so that address translation is required only on a cache miss, and a single Translation Buffer which holds translations for both instruction and data accesses. The Translation Buffer has a set of 16 user-region entries. This set is fully-associative (any translation can be stored in any entry) with FIFO replacement. The TB also has a single buffer region entry and a "hardwired" kernel space entry. (Kernel addresses are translated as if kernel space was bound to a 4MB super-page located at real address 0 in local memory.)

In addition to the TB, the Antares MMU has a 4-entry Directory Buffer (DB), which holds the most-recently-used directory entries together with the DI fields of the corresponding virtual addresses. The DB also is fully associative with FIFO replacement. When a TB *miss* occurs (i.e., a translation is not found in the TB), the DI field of the virtual address is compared with the DI fields of DB entries. If a match is found, the page table origin is obtained from the DB entry, eliminating a memory read for the directory entry. In many cases, then, only a single one-word memory read is required in processing a TB miss.

Translation Buffer Invalidation. The Translation Buffer (and Directory Buffer) contents are invalidated on machine reset (see Appendix C) and as a side effect of writing to the IdR via a Move to Special instruction.[7] (The kernel entry is not affected in either case.) Antares does not maintain ASNs in TB entries; the TB must be invalidated on an address space switch or on any change to a translation table entry of the active address space. An address space is switched by executing a Move Special instruction to set the ASN field of the IdR so that DSA|ASN|0X000 points to the start of the directory for the address space to be activated. Although the TB contains only page table entries, it is still necessary to invalidate it whenever a change is made to a directory entry of the currently-active address space because of the Directory Buffer.

Translation Changes and the Cache. Depending on the implementation, a change in the mapping or state of a virtual page may require explicit invalidation of cache entries as well as the TB/TLB entry. In implementations with real-addressed caches and Translation Lookaside Buffers, invalidation of a TLB entry may effectively invalidate associated cache lines. Implementations with virtually-addressed caches which include an ASN in cache line tags do not require invalidation on address space switches but may require explicit invalidation of entries in other cases. The instructions provided for cache invalidation and the details of their operation depend on the implementation.

The Antares caches do not include ASNs in their tags, and must be invalidated on an address space switch, as well as when a change in the mapping or state of a page in the active address space changes. The data cache can be

[7] Other Scorpius implementations may provides instructions to selectively invalidate TB entries.

selectively invalidated, using the Read Data Tag instruction to examine cache line tags and the Invalidate Data Cache line instruction to invalidate lines (after writing modified lines to memory when necessary, using the Flush Data Line instruction). The entire instruction cache can be invalidated using the Invalidate Instruction Cache All (**IICA**) instruction. When instruction cache lines of a single page are to be invalidated, looping through the 128 lines of the page with the Invalidate Instruction Cache Line (**IIC**) instruction may be more efficient than invalidating the entire cache with **IICA**. Later implementations with larger caches can be expected to provide other means for cache invalidation. Cache flushing and invalidation are discussed in Chapter 6.

Translation Changes and the Write Buffer. In Antares, when a data cache miss occurs and the line selected to be replaced is modified, that line is transferred to a Write Buffer so that the missing line can be read from memory without waiting for the line being replaced to be written to memory. (A line written as the result of execution of one of the cache control instructions also is placed in the Write Buffer.) The Write Buffer operates asynchronously; its contents are written to memory when the memory bus becomes idle or when it is required for a subsequent cache line moveout. If a miss occurs while the Write Buffer is waiting for the memory bus, and that miss does not replace a modified line, the memory read for that miss is allowed to proceed; the Write Buffer write continues to wait. (The Write Buffer is not used for single word writes to non-cacheable pages.)

It is possible that, at the time a translation table change or an address space switch is to be made, a line may be in the Write Buffer waiting for the memory bus. Writing that line to memory correctly requires that the correct address translation exist; therefore, the Write Buffer should be flushed prior to a translation table change or an address space switch. To flush the Write Buffer, the kernel can use the following procedure:

- execute a Create Data Cache instruction to create a line at some convenient location in kernel space,
- store some arbitrary value into this line to change its state to modified, and
- execute a Flush Data Cache instruction for this line. The current contents of the Write Buffer will be written to memory prior to completion of execution of this instruction.

Translation Changes and the Pipeline. Changing a page translation requires that the TB be invalidated to insure that the original translation is discarded. If this TB invalidation is performed in user space, rather than kernel space, it is possible that, as a consequence of the pipeline's prefetching of instructions, the pipeline instruction fetch queue may contain instructions fetched using the original translation. These instructions can be discarded by executing a jump instruction immediately after the Move To Special instruction which caused the TB to be invalidated.

Because of the implementation-dependent nature of the TB/TLB and cache operations discussed in this section, their operating system use should be localized. For model-dependent code, the operating system can determine on which Scorpius implementation it is executing by examining the IdR's model number field.

3.7 Address Translation in Antares

This section gives a step-by-step description of the address translation process in Antares. The address translation process in other Scorpius implementations will be similar, but not necessarily identical. The translation process is diagrammed in Figures 3.10(a) - 3.10(c); bracketed numbers in the following discussion refer to "box" numbers in these figures.

TB Search [1]. In Antares, with its virtually-addressed cache, address translation is done only on a cache miss or on an access to a non-cached page. Translation begins with a search of the TB. Each TB entry comprises a virtual page number, called a *tag* in this discussion, and the page table entry for the corresponding real page. (The hardwired TB entry for kernel space essentially is the same as a buffer region entry for a super-page of 4MB with the real page located at real address 0x00000000.)

The TB is fully associative, so the tags of all valid TB entries are simultaneously compared with the appropriate bits of the virtual address. For the 16 user region entries, tags are compared with the virtual page number in address bits <31:13>. For the buffer region entry, the SPS field of the entry how many bits of the virtual address are to be compared. For the kernel entry, virtual address bits <31:22> are compared with the hardwired tag 0x3FF (11 1111 1111B).

TB Hit [2-5]. If the comparison of the selected virtual address bits and a TB entry tag results in a match, the mode of the accessing PU and the access type are checked against the privilege flags in the entry. A privilege violation results in the generation of an access privilege violation trap. Execution of the instruction making the access is terminated, the appropriate trap source flag (instruction access privilege violation or data access privilege violation) is set in the Trap Register, virtual address bits <31:13> are stored in the Trap Register's argument field (data access privilege violation only), and control is transferred to the kernel. An instruction access privilege violation can result only from an attempt to access a system page while in user mode. To determine the specific cause of a data access privilege, the kernel may need to form the page table entry address (just as the MMU does), read the page table entry, and examine the privilege flags for the page; it also may need to examine the operation code of the interrupted instruction. When the trap is recognized, the address of the interrupted instruction is contained in PCQ[1]. (In certain cases, the address in PCQ[1] may require adjustment; see Section 4.6.)

If the access is allowable, the real instruction or data address is formed. (In figure 3.10(a), the abbreviations "VA" and "RA" refer to the virtual address and

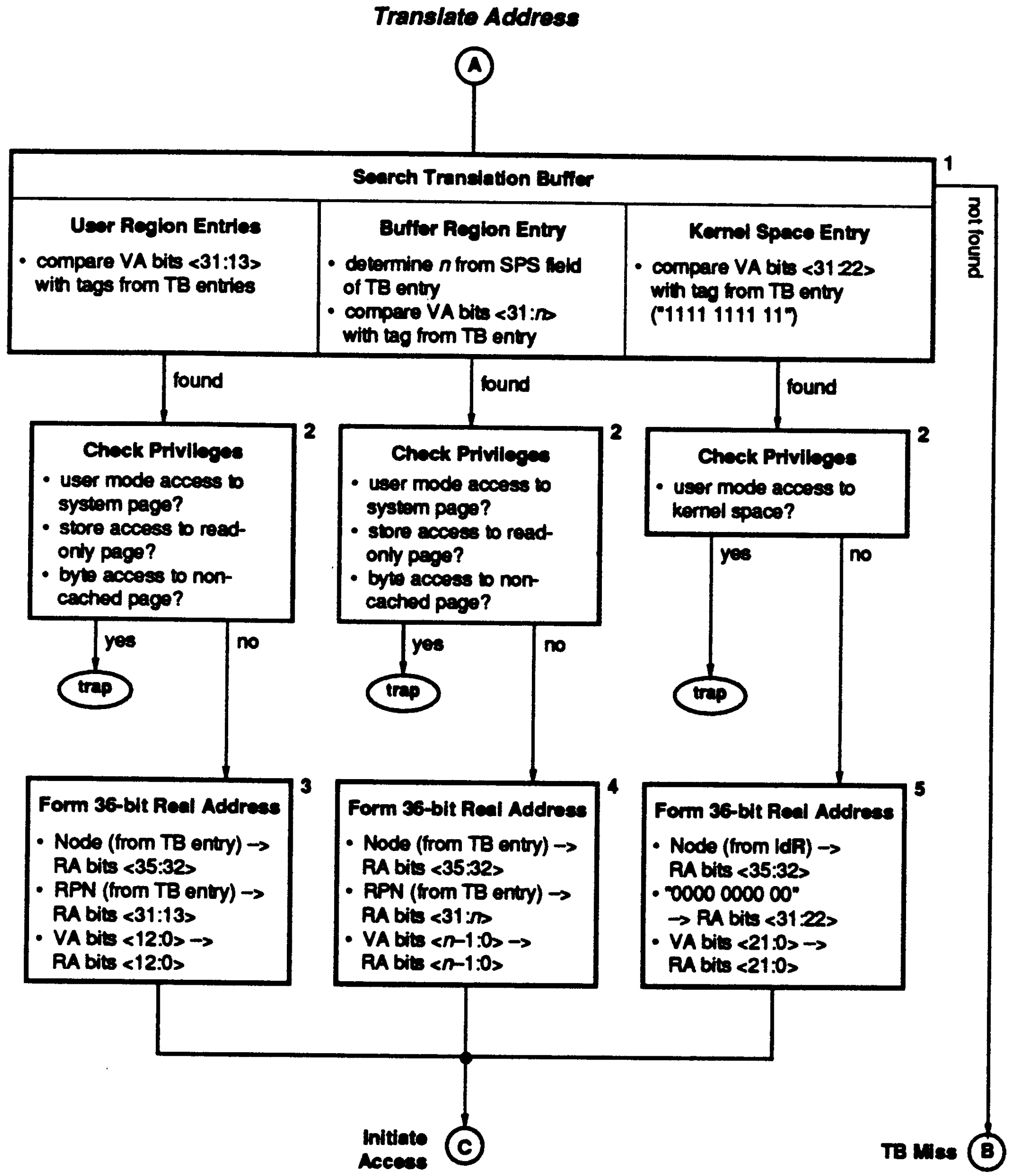

Translate Address
A
Search Translation Buffer
1
not found
User Region Entries
• compare VA bits <31:13> with tags from TB entries
Buffer Region Entry
• determine n from SPS field of TB entry
• compare VA bits <31:n> with tag from TB entry
Kernel Space Entry
• compare VA bits <31:22> with tag from TB entry ("1111 1111 11")
found
found
found
Check Privileges
• user mode access to system page?
• store access to read-only page?
• byte access to non-cached page?
2
Check Privileges
• user mode access to system page?
• store access to read-only page?
• byte access to non-cached page?
2
Check Privileges
• user mode access to kernel space?
2
yes
no
trap
yes
no
trap
yes
no
trap
Form 36-bit Real Address
• Node (from TB entry) –> RA bits <35:32>
• RPN (from TB entry) –> RA bits <31:13>
• VA bits <12:0> –> RA bits <12:0>
3
Form 36-bit Real Address
• Node (from TB entry) –> RA bits <35:32>
• RPN (from TB entry) –> RA bits <31:n>
• VA bits <n–1:0> –> RA bits <n–1:0>
4
Form 36-bit Real Address
• Node (from IdR) –> RA bits <35:32>
• "0000 0000 00" –> RA bits <31:22>
• VA bits <21:0> –> RA bits <21:0>
5
Initiate Access
C
TB Miss
B

Figure 3.10(a). Address Translation in Antares

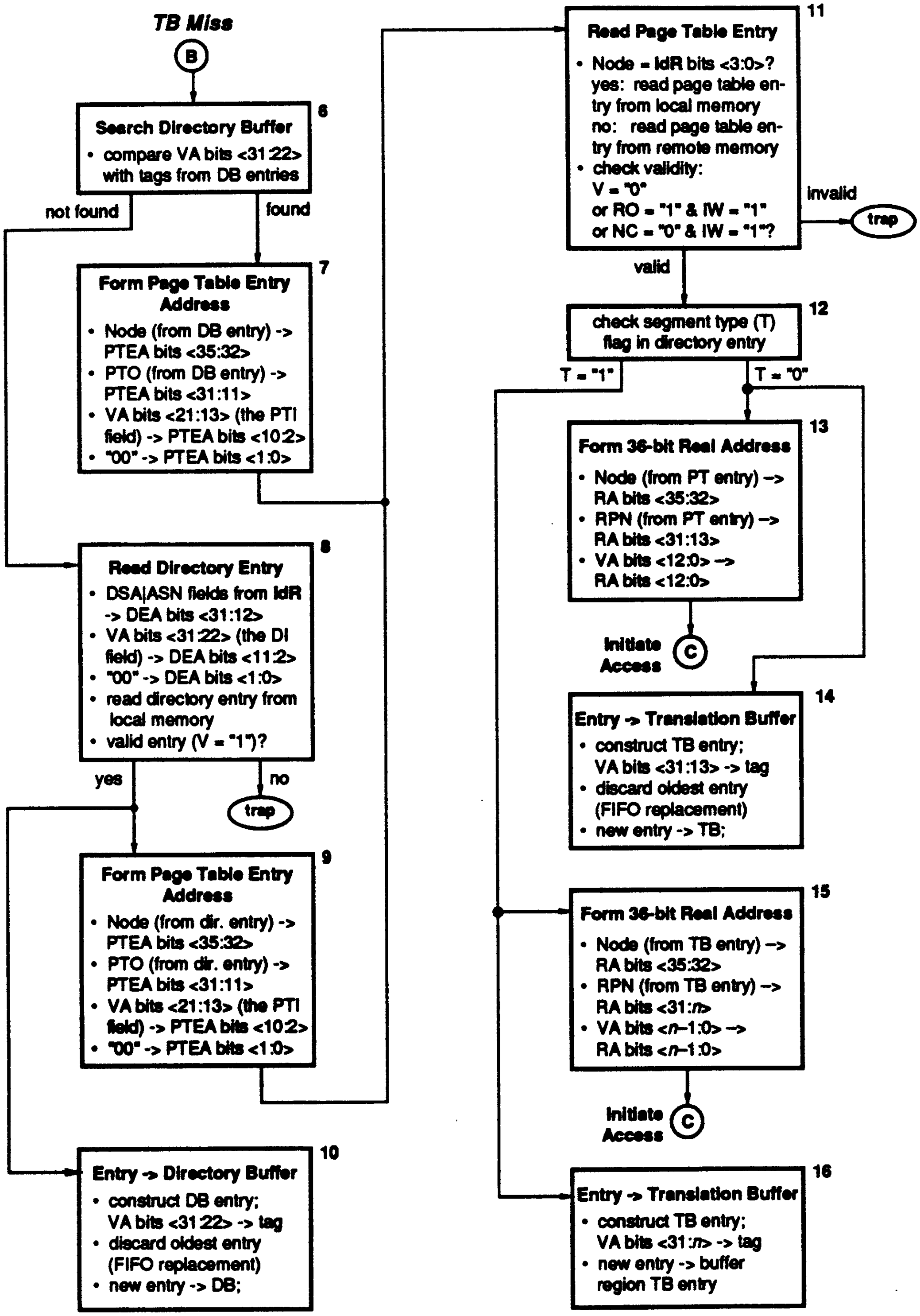

TB Miss
B
6
Search Directory Buffer
• compare VA bits <31:22> with tags from DB entries
not found
found
7
Form Page Table Entry Address
• Node (from DB entry) -> PTEA bits <35:32>
• PTO (from DB entry) -> PTEA bits <31:11>
• VA bits <21:13> (the PTI field) -> PTEA bits <10:2>
• "00" -> PTEA bits <1:0>
8
Read Directory Entry
• DSA|ASN fields from IdR -> DEA bits <31:12>
• VA bits <31:22> (the DI field) -> DEA bits <11:2>
• "00" -> DEA bits <1:0>
• read directory entry from local memory
• valid entry (V = "1")?
yes
no
trap
9
Form Page Table Entry Address
• Node (from dir. entry) -> PTEA bits <35:32>
• PTO (from dir. entry) -> PTEA bits <31:11>
• VA bits <21:13> (the PTI field) -> PTEA bits <10:2>
• "00" -> PTEA bits <1:0>
10
Entry -> Directory Buffer
• construct DB entry; VA bits <31:22> -> tag
• discard oldest entry (FIFO replacement)
• new entry -> DB;
11
Read Page Table Entry
• Node = IdR bits <3:0>? yes: read page table entry from local memory no: read page table entry from remote memory
• check validity: V = "0" or RO = "1" & IW = "1" or NC = "0" & IW = "1"?
invalid
trap
valid
12
check segment type (T) flag in directory entry
T = "1"
T = "0"
13
Form 36-bit Real Address
• Node (from PT entry) –> RA bits <35:32>
• RPN (from PT entry) –> RA bits <31:13>
• VA bits <12:0> –> RA bits <12:0>
Initiate Access
C
14
Entry -> Translation Buffer
• construct TB entry; VA bits <31:13> -> tag
• discard oldest entry (FIFO replacement)
• new entry -> TB;
15
Form 36-bit Real Address
• Node (from TB entry) –> RA bits <35:32>
• RPN (from TB entry) –> RA bits <31:n>
• VA bits <n–1:0> –> RA bits <n–1:0>
Initiate Access
C
16
Entry -> Translation Buffer
• construct TB entry; VA bits <31:n> -> tag
• new entry -> buffer region TB entry

Figure 3.10(b). Address Translation in Antares (continued)

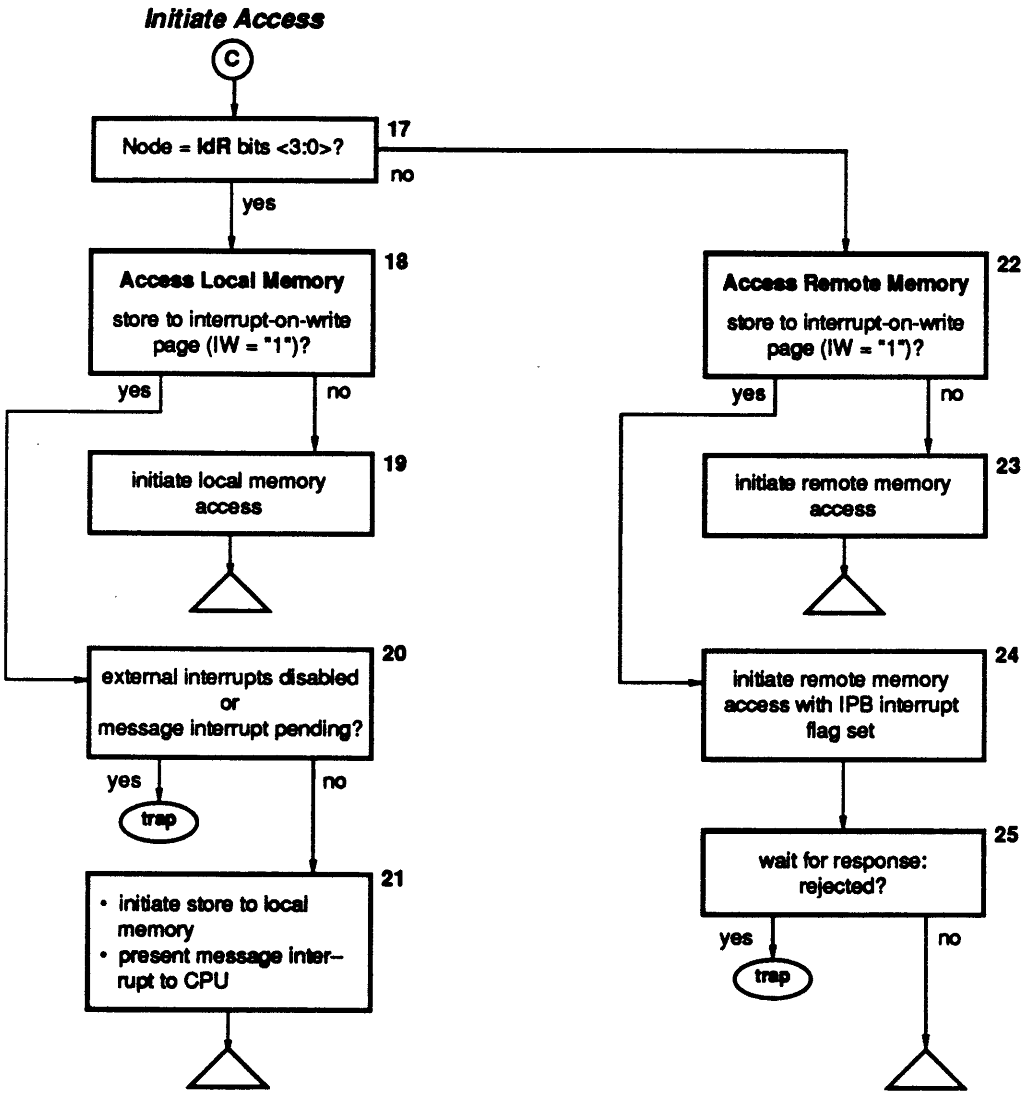
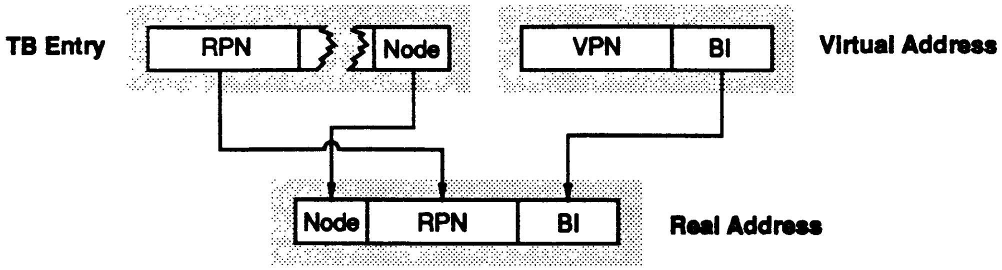

Initiate Access
C
17
Node = IdR bits <3:0>?
no
yes
18
Access Local Memory
store to interrupt-on-write page (IW = "1")?
yes
no
19
initiate local memory access
20
external interrupts disabled or message interrupt pending?
yes
trap
no
21
• initiate store to local memory
• present message inter–rupt to CPU
22
Access Remote Memory
store to interrupt-on-write page (IW = "1")?
yes
no
23
initiate remote memory access
24
initiate remote memory access with IPB interrupt flag set
25
wait for response: rejected?
yes
trap
no

Figure 3.10(c). Address Translation in Antares (continued) its real translation.) For user space pages, this address is formed by concatenating the Node and Real Page Number fields from the TB entry with the intra-page byte index (BI) from the virtual address, as shown below.[8]



For buffer region addresses, the numbers of bits comprising the RPN and BI fields is determined by the SPS field. Kernel space references always are translated into local memory accesses. Real address bits <31:22> (the equivalent of the RPN field) are set to "0", so kernel address 0XFFC00000 translates into real address 0Xn00000000, where "n" represents the node number of the CPU generating the access. Initiation of the memory access for the instruction or data word or line is described later in this section.

TB Miss [6-16]. If no match is found between the virtual page number and the tags of the valid TB entries, a TB miss occurs; both the directory entry and the page table entry for the virtual page must be accessed to effect the translation. (Since the TB entry for the kernel is "hardwired", a TB miss can never occur on a kernel space reference. The following discussion of TB miss processing therefore applies only to user space references.)

The Antares Directory Buffer (DB) is a four-entry, fully-associative cache which holds the four most-recently-referenced directory entries , each with the DI field of the corresponding virtual address (called a *tag* in Figure 3.10(b).) In processing a TB miss, the MMU first searches the DB for an entry whose tag matches the DI field of the virtual address. The DI field always comprises virtual address bits <31:22>, regardless of whether the access is to a user region or a buffer region. If a match is found between the DI field and a DB entry tag, the Node and Page Table Origin (PTO) fields from the DB entry are concatenated with the Page Table (PTI) Index field from the virtual address to form the page table entry address (PTEA). The PTI field always comprises virtual address bits <21:13>, regardless of whether the access is to a user region or a buffer region; access to the page table entry is done in the same way for both kernel and user region pages.

If the DI field does not match any of the DB entry tags, a DB miss occurs; the directory entry must be read from memory. The real address of the directory

[8] While real address formation is described here in terms of byte addresses, Antares off-chip addresses are word addresses; the addresses actually constructed by the MMU always are directed off-chip and so are formed as word addresses.

entry is formed by concatenating the DSA and ASN field from the IdR with the DI field from the virtual address (also see Figure 3.5); the directory always resides in local memory. When the directory entry has been read from memory, its V flag is examined; if the entry is invalid, an instruction or a data page fault trap (depending on the access type) is generated and control transferred to the kernel as described earlier. If the directory entry is valid, its Node and PTO fields are extracted and, together with the PTI field from the virtual address, used to from the page table entry address. At the same time, the directory entry and tag (DI field of the virtual address) are stored in the DB, replacing the oldest entry (i.e., FIFO replacement).

When the page table entry has been read from either local or remote memory, it is checked to determine if the entry is valid (V = "1") and legal. If the entry is invalid or contains an illegal combination of privilege bits, an instruction or a data page fault trap (depending on the access type) is generated and control transferred to the kernel. Otherwise, the Node and Real Page Number (RPN) fields from the page table entry are concatenated with the intra-page byte index (BI) field from the virtual address to form the real address; the segment type (T) flag from the directory entry is used to determine which bits from the page table entry are to be used as the RPN and which bits from the virtual address are to be used as the BI. When the address has been formed, a memory access is initiated for the instruction or data word or line. At the same time, the new translation is stored in the TB. If the translation was performed for a buffer region address, the page table entry and tag (VPN field of the virtual address) are stored in the special buffer region TB entry, displacing the current contents of that entry. If the translation was performed for a user region address, the page table entry and tag replace an invalid TB entry, if one exists, or otherwise replace the oldest of the 16 user region TB entries (again, FIFO replacement).

Access Initiation [17-25]. When address translation is completed, a memory access is initiated for the instruction or data word or line. The Node field taken from the page table entry determines if the access is directed to local memory via the local memory bus or to remote memory via the IPB.

A store to a page marked interrupt-on-write is a special case of a store word access. If the store is to local memory, the MMU first determines if the CPU is disabled for external interrupts (ICR bit <5> = "0") or if there already is a message interrupt pending at the CPU (**ICR** bit <27> = "1"). If either condition holds, the store is not executed; instead, a Message Reject Trap is generated on the PU attempting to execute the store, assuming the PU is interrupt/trap enabled (PsR bit <15> = "1"), and control is transferred to the kernel. If the PU is interrupt/trap disabled, a PU Check Trap, rather than a Message Reject Trap, is generated.[9] If the CPU is enabled for external interrupts and no message

[9] Because the attempted generation of a trap while a PU is interrupt/trap disabled is assumed to be an error event. Attempted generation of an Access Privilege Violation Trap or a Page Fault Trap while interrupt/trap disabled also results in generation of a PU Check Trap.

interrupt is pending, the MMU initiates the memory store, sets the Message Interrupt Pending flag in the ICR, and stores the real byte address (without Node field) of the word being stored in the IAR. (Bits <1:0> of this address will be "00".) If there is an available PU (i.e., a PU which is interrupt/trap enabled), the Message Interrupt will be recognized and processed.

If the store is to remote memory, the MMU initiating the store — the sender — generates an IPB request in which an interrupt flag is set in the request "header", and then waits for a response to the request. The receiving MMU checks its CPU's external interrupt enabled and message interrupt pending flags; if the CPU is disabled for external interrupts or if it already has an message interrupt pending, the MMU does not execute the store but instead returns a "request rejected" response to the sender. The sending CPU then generates a Message Reject Trap on the PU initiating the store (or a PU Check Trap, if that PU is interrupt/trap disabled). If the receiving CPU is enabled for external interrupts and does not have a message interrupt pending, its MMU returns a "request accepted" response to the sending CPU, initiates the store access, sets the Message Interrupt Pending flag in the ICR, stores the real address of the store operand in the IAR, and presents a message interrupt to the receiving CPU.

3.8 Cache and TB Coherency in Antares

Whenever multiple copies of data can exist in a way such that changing the value of one copy does not change the value of all copies, there is a *coherence* problem. Coherence problems can arise when copies of a datum are distributed to multiple users under a single name, or when a single user acquires multiple copies of a single datum under different names (the *synonym* problem). In Antares, maintaining coherence of data is the responsibility of software. Coherency problems can arise in the following areas.

Cache Coherency in Single CPU Systems. There are two ways in which the Antares cache can have multiple copies of the same memory line. First, a line referenced both on an instruction fetch and on a data fetch or store (either because of code modification or because instructions and data were stored in the same line) can have one copy in the instruction cache and one copy in the data cache. While this may be somewhat inefficient, it presents a coherence problem only when code is modified. Modifying the copy of the line in the data cache does not change the copy of the line in the instruction cache; instruction fetches to that line will fetch its original contents. When code modification or "on-the-fly" generation is required, correct operation can be obtained as follows:

- assume α is the address of a line containing instructions to be modified. After making the desired changes to this line,
- execute a **FDC** (Flush Data Cache Line) instruction to force the contents of line α to be written to memory, and

- execute an **IIC** (Invalidate Instruction Cache Line) instruction to invalidate the instruction cache's copy of line α, so that an instruction fetch to line α will cause an instruction cache miss and bring in the line just flushed from the data cache.

Attention must be paid to line boundaries; the **IIC** instruction should not reside in the modified line. Also, if instructions from line α may be present in the pipeline's instruction fetch queue, the queue must be flushed (by executing a jump instruction).

Because Antares has a virtually-addressed cache, multiple copies of a line can appear in the instruction cache, or the data cache, or both, because of synonyms - multiple virtual addresses mapping to the same real address. (This implies that multiple virtual pages have been mapped to a single real page; consequently, the TB may contain multiple translations for the same real page.) When multiple copies of a line exist in the data cache, a store to a copy updates only that copy, so that a read of another copy will not return the current contents of the line. To avoid this problem, the software must be aware of synonyms; whenever a page is to be modified and that page can have synonyms by which it may be subsequently referenced, it is necessary to insure that synonym lines are deleted from the cache prior to such references. This can be done by inspecting each data cache line with the **RDTX** instruction and deleting synonym lines using an **IDC** instruction. It may sometimes be necessary to flush the TB to discard entries for synonym pages.

Cache Coherency in Multiple CPU Systems. Coherence problems can arise when a page is shared between nodes; a common instance is an IO buffer page accessed by an Antares CPU and an IO processor. In performing a write operation, it is necessary to flush all buffer page lines from the data cache (via **FDC** instructions) prior to requesting the IO processor to initiate the write.

A real memory page can be mapped into multiple address spaces on multiple CPUs, and copies of lines from that page can simultaneously exist in each CPU's cache. If one CPU modifies its copy of the line, copies of the lines in the caches of the other CPUs are not updated; it is the responsibility of software to insure that all users of a shared page access current copies of that page. The instance of the system executing on the CPU making the change must send a message to all other potential users of that page advising them to invalidate lines of that page in their caches.

TB Coherency. The TB and DB are caches for translation table entries for the currently-active address space, and also present coherency and synonym problems. Whenever a translation table entry is modified, it is necessary for the operating system to insure that any TBs/DBs which may contain a copy of that entry are flushed.[10] Situations in which this is required include the following:

[10] As noted earlier, the TB and DB must be flushed on an address space switch or termination.

- page state changes, such as changes from read-only to read-write made as part of the operating system's tracking of modified pages and changes to the OS field of directory and page table entries;
- page deallocation;
- page reallocation (as on a store to a copy-on-write page),

As a consequence of page sharing, a real memory page can be mapped into multiple address spaces on multiple CPUs (on any single CPU, that page can be mapped into both inactive address spaces and the active address space). Thus, a number of translations to a single real page can appear in TBs throughout the system. It is the responsibility of the operating system to insure that changes in the state of that real page are propagated to every translation table which may have a translation for that page and to every TB which may have a copy of the translation.

4. Interrupts and Traps

4.1 Introduction

Interrupts and traps are events which cause control to be transferred from the currently-executing program to an interrupt processing program, commonly called the *interrupt handler*, in the kernel region. This chapter describes the interrupts and traps defined in Scorpius, the steps involved in transferring control to the interrupt handler, the CPU state resulting from this transfer, considerations in analyzing the cause of an interrupt or trap, and the process by which control can be returned to the original program.

Portions of this description may be relevant only to Antares, where implementation constraints require that some analysis of interrupt and trap causes and of CPU state be done by software. Later implementations may perform a larger part of this analysis in hardware. Careful organization of the interrupt handler can ease the task of transporting the operating system from one implementation to another.

4.2 Interrupts and Traps

The distinction between interrupts and traps traditionally has been based on their source; from the viewpoint of the executing program, an interrupt is an external event unrelated to program execution, while a trap is an internal event, caused by execution of a particular instruction in the program. In Scorpius, the distinction is based on destination; traps always are processed by a specific PU, while interrupts can be processed by any available PU. Traps divide into two classes: *local exceptions* (e.g., page fault, overflow), and *inter-PU signals*. The inter-PU signals generated by **Prmpt** (Preempt) and **Res** (Restart) instructions are classified as traps because they are directed to a specific PU. Also, since these can be sent to several PUs simultaneously, local storage of interrupt identifying information is needed, just as local storage of local exception identifying information is needed.

Interrupts	Traps
• Reset • Machine Check — IPB error — hardware error • Power/Temp. • Deadlock • IO • Message • Event Counter 2 Overflow • Event Counter 1 Overflow	• PU Check • PU Restart • PU Preempt • Data Page Fault — access to invalid page/segment — access to illegal page table entry • Data Access Privilege Violation — user mode access to system page — user mode access to kernel region — store to read-only page/segment — StB/LdB to non-cached page • Message Reject • System Call • Operation Fault — illegal operation code — operation privilege violation — taken branch trap • Overflow • Instruction Page Fault — access to invalid page/segment — access to illegal page table entry • Instruction Access Privilege Violation — user mode access to system page — user mode access to kernel region

Figure 4.1. Scorpius Interrupts and Traps

Scorpius interrupts and traps are listed in Figure 4.1; a later section in this chapter provides a detailed description of each interrupt and trap. Note that several traps and interrupts have multiple causes (or subclasses); the kernel's interrupt/trap handler must examine the CPU state to determine the exact cause. The reset interrupt generated on machine power-on or via a reset switch is implementation-dependent and is described in an appendix. The discussion in this chapter excludes this interrupt.

Interrupts and traps result in transfers to different kernel entry addresses. Also, the entry address for an interrupt or a trap depends on the interrupt/trap type and on the setting of the PU Available flag in the PU Status and Control Register (PsR). Entry address selection is discussed in a later section.

From a hardware view, interrupt and trap handling divides into three phases: generation, presentation, and recognition. An interrupt or trap is *generated* when a particular event, such as a machine error, occurs; it is then *presented* to the CPU for assignment of a PU to process it. Traps always are presented to the PU on which they were generated. An interrupt or trap is *recognized* when a PU is selected to process it and a transfer of control to the interrupt handler on that PU is initiated.

There are three forms of interrupt/trap control. First, for certain interrupts and traps, the generation of the interrupt or the trap may be disabled, in which case the underlying event does not occur insofar as the CPU's interrupt handler

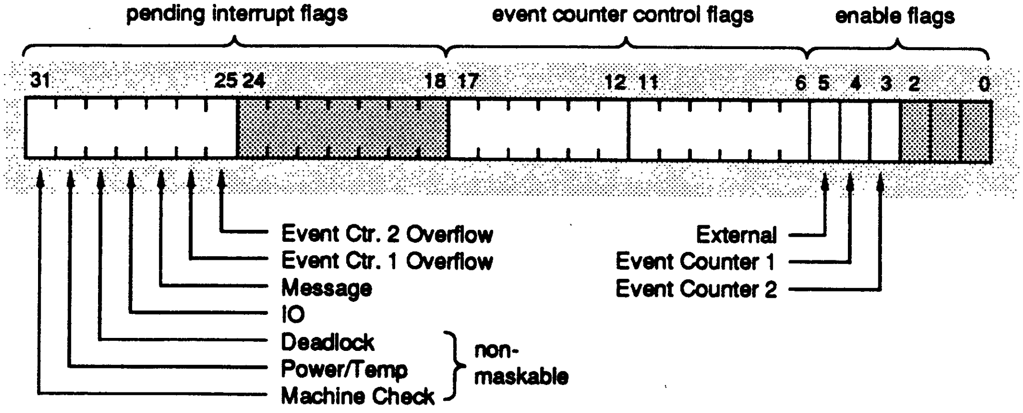


Figure 4.2(a). Interrupt Control Register (ICR)

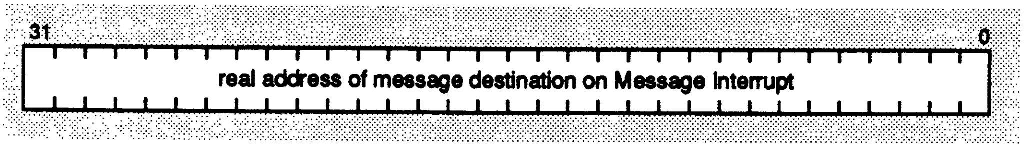


Figure 4.2(b). Interrupt Argument Register (IAR)

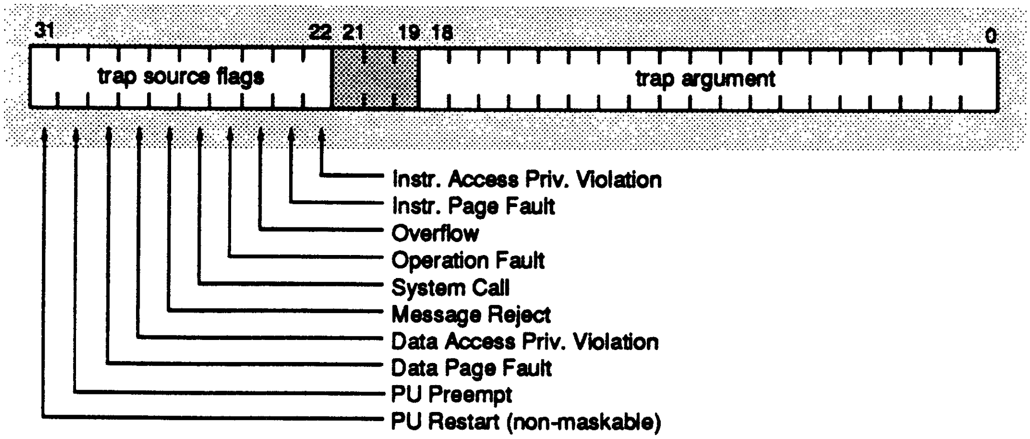


Figure 4.2(c). Trap Register (TrapR)

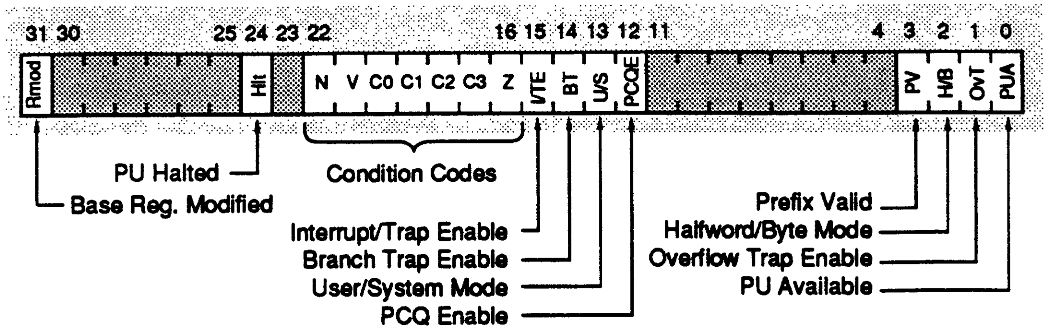


Figure 4.2.(d). PU Status and Control Register (PsR)

disabled at presentation. Third, a PU can disable recognition of interrupts and traps, in which case an interrupt (although not a trap) may be assigned to another PU. Interrupts and traps whose presentation or recognition can be disabled are called *maskable*. Interrupts and traps resulting from errors cannot be disabled; these are called *non-maskable*.

Interrupts can be processed by any PU. In recognizing an interrupt, the CPU assigns a maskable interrupt to a halted PU, if there is at least one halted PU, and otherwise to a PU for which interrupt/trap recognition is not disabled. If all four PUs have interrupt/trap recognition disabled, the interrupt continues to be presented (remains pending) until some PU enables interrupt/trap recognition and the pending interrupt can be processed. Non-maskable (error) interrupts always are presented to and are immediately recognized by PU 0. A trap always is presented to the PU on which it was generated, in the case of local exceptions, or to which it was directed, in the case of inter-PU signals.

Interrupt/trap generation, presentation, and recognition are controlled by setting flags in certain special registers. The special registers involved in interrupt and trap processing are the Interrupt Control Register (ICR), the Interrupt Argument Register (IAR), the Trap Register (TrapR), and the PU Status and Control Register (PsR). These registers are shown in Figure 4.2. The ICR and IAR are global registers, while the TrapR and PsR are local registers. The ICR, IAR, and TrapR are privileged and can be accessed only in system mode; these registers are accessed via Move to Special and Move from Special instructions. PsR flags are set and cleared via Set Mode and Clear Mode instructions, and can be inspected via the Test Mode instruction. The flags in bits <3:0> of the PsR are non-privileged, and can be accessed in either user or system mode; other PsR flags can be accessed only in system mode. A summary description of PsR fields is given in Figure 2.5.

4.3 Interrupt Generation, Presentation, and Recognition Control

Generation of Event Counter Overflow interrupts, and presentation of IO and Message interrupts, can be enabled or disabled. Generation of Event Counter

Overflow interrupts is enabled by setting the appropriate Event Counter Overflow Interrupt Enable Flags in the ICR (bit <4> for counter 1 or bit <3> for counter 2) to "1", and disabled by clearing these flags to "0". Presentation of IO and Message interrupts is enabled by setting the External Interrupt Enable Flag (ICR bit <5>) to "1" and disabled by clearing this flag to "0". To set or clear these flags, a Move From Special instruction is used to transfer the ICR contents to a general register, the appropriate logical instructions are used to operate on the enable flag bits, and a Move To Special instruction is used to update the ICR. The ICR can be accessed only in system state; also, since it is global, it should be updated only in a critical section.

If an IO or a Message interrupt arrives while external interrupt presentation is disabled (External Interrupt Enable Flag in the ICR = "0"), it is rejected. For a message interrupt, external interrupt rejection results in cancellation of the store to an interrupt-on-write page which generated the message interrupt and the generation of a Message Reject Trap on the PU (local or remote) which initiated that store. For an IO interrupt, external interrupt rejection simply causes the state of the external interrupt signal to be ignored; if external interrupt presentation is later enabled and the external interrupt signal is still asserted, the IO interrupt can be recognized. Also, an IO interrupt will be rejected if an earlier-arriving IO interrupt already is pending, and a Message interrupt will be rejected if an earlier-arriving Message interrupt already is pending. However, an IO interrupt can be accepted even if a Message interrupt is pending, and conversely.

An event counter can run — and overflow — regardless of whether or not Event Counter Overflow interrupt generation is enabled, since counter operation is controlled solely by the event counter control flags. However, if Event Counter Overflow interrupt generation is disabled (by clearing the appropriate enable flags in the ICR), an interrupt will not be generated, even if a counter overflows. Thus, a PU can disable External and Event Counter interrupts, process any pending interrupts, and then open an "enabled window", in which it is guaranteed that no (non-error) interrupt can occur, by setting its Interrupt/Trap Enable flag (Section 4.4). This is needed in message sending; a message reject is treated as an error if it occurs while the PU is interrupt/trap disabled (see the description of the Message Reject trap).

Recognition of maskable interrupts by an individual PU is disabled when the Interrupt/Trap Enable Flag (I/TE), bit <15> in the PU's PsR, is cleared to "0". A PU with this flag cleared (set) is said to be *interrupt/trap disabled (enabled)*. Maskable interrupts are the Event Counter Overflow, Message, and IO interrupts. When a maskable interrupt is presented to an interrupt/trap enabled PU and recognized, the IT/E flag is cleared — and interrupt/trap recognition disabled — as a consequence of saving the current PsR in SaveR and clearing the PsR. (Clearing the PsR also clears the User Mode, PCQ Enable, Overflow Trap Enable, and Taken Branch Trap Enable flags, among others.) A PU can, while in system mode, become interrupt/trap enabled or disabled by setting or clearing the IT/E flag (via Set/Clear Mode instructions). To dispatch a user or system task in an interrupt/trap enabled (or disabled) state, the kernel sets (or clears) bit <15> in the SaveR via a Move to Special Instruction before the return from interrupt which initiates or resumes task execution. (Typically, this flag is set to "1" when execution of a task is first initiated and remains unchanged thereafter.)

The Reset, Machine Check, Power/Temp., and Deadlock interrupts are non-maskable. When one of these interrupts is generated by or arrives at the CPU, it is immediately presented to and recognized by PU 0.

Pending Interrupt Flags. When an interrupt is recognized, the appropriate pending interrupt flag is set in the ICR; also, for a Message interrupt, an argument is stored in the IAR. The kernel can examine the pending interrupt flags to determine the cause of an interrupt. The pending interrupt flags, ICR bits <31:25>, are set to "1" only by hardware and are cleared to "0" only by software. (Other ICR fields are set and cleared only by software.) If an attempt is made to write a "1" to a pending interrupt flag, the state of the flag does not change.

At the time an interrupt is recognized, more than one pending interrupt flag may be set, either because multiple interrupts were generated in the same cycle or because multiple interrupts occurred while all PUs were disabled or while those interrupts were masked. The kernel enters a critical section, reads the ICR, decides which interrupt to process,[1] generates a mask for the pending interrupt flags field in which all bits are "1"s except for the bit corresponding to the interrupt being processed, stores the mask in its copy of the ICR, and writes that copy back to the ICR. (The kernel may, at the same time, clear interrupt enable flags if desired.) Critical section entry insures that only one kernel instance at a time modifies the ICR; writing "1"s to all flags except that for the recognized interrupt insures that the state of any flag set by hardware during this read-modify-write process is preserved.

When multiple interrupts are pending, the hardware interrupt handler insures that a separate instance of interrupt presentation occurs for each interrupt. For example, suppose all PUs initially are interrupt/trap disabled, an event counter overflow interrupt is generated and an external interrupt arrives, setting two pending interrupt flags. Assume PU 0 becomes interrupt/trap enabled; an interrupt will be presented, the kernel instance executing in PU 0 will select one of the two pending interrupts to process and clear the pending interrupt flag for that interrupt. When PU 0 or any other PU becomes interrupt/trap enabled, the other interrupt will be presented to that PU.

Interrupt recognition is serialized so that a pending interrupt flag will not be repeatedly set before being cleared. If a Message interrupt arrives to find external interrupt presentation enabled but all PUs interrupt/trap disabled, it is rejected if a Message interrupt already is pending (ICR bit <27> = "1"); otherwise, it is queued (by setting the Message Interrupt Pending flag and storing an

[1] by, for example, using a **CLZ** instruction to translate the flag into a jump table index.

State Seen by an Arriving Message Interrupt			
External Interrupt Enable Flag (ICR bit 5)	Message Interrupt Pending Flag (ICR bit 27)	PU Interrupt/Trap Enable Flag (PsR bit 15)	*Response to Interrupt*
0	don't care	don't care	reject interrupt
1	0	1 (any)	recognize interrupt: argument -> IAR, set Message Interrupt Pending flag, assign PU to process interrupt
1	0	0 (all)	queue interrupt: argument -> IAR, set Message Interrupt Pending flag
1	1	don't care	reject interrupt

Figure 4.3. Responses to an Arriving Message Interrupt

argument in the IAR). Similarly, if an IO interrupt arrives to find external interrupt presentation enabled but all PUs interrupt/trap disabled, it is rejected if an IO interrupt already is pending (ICR bit <28> = "1"); otherwise, it is queued (by setting the IO Interrupt Pending flag).

An IO or a Message interrupt also is rejected if it finds external interrupt presentation disabled upon its arrival. If external interrupt presentation is disabled while an IO or a Message interrupt is pending, or if event counter overflow interrupt generation is disabled while an event counter overflow interrupt is pending, the interrupt remains pending and will be presented and recognized when a PU becomes interrupt/trap enabled. The possible responses to an arriving Message interrupt are summarized in Figure 4.3. Responses to an IO interrupt are similar, except that no argument is stored.

Interrupt Arguments. The Interrupt Argument Register (IAR) is a privileged global register which holds the argument associated with a pending Message interrupt. The IAR is shown in Figure 4.2(b). When a Message interrupt arrives,[2] the hardware interrupt handler first examines the external interrupt mask (ICR bit <5>); if this bit is "0", the interrupt is rejected. Otherwise, the Message Interrupt Pending flag (ICR bit <27>) is examined; if this flag is "0", it is set it to "1", and the real byte address of the message destination is stored in the IAR (with the two low-order bits cleared to "0"). If the Message interrupt finds the Message Interrupt Pending Flag already set to "1", it is rejected, as discussed earlier. In processing a Message interrupt, the kernel must save the contents of the IAR before clearing the Message Interrupt Pending Flag in the ICR.

[2] or is generated; a Message interrupt can result from a store to an interrupt-on-write page in local memory as well as one in remote memory.

4.4 Trap Generation, Presentation, and Recognition Control

Generation of the Overflow trap and the Taken Branch trap is controlled by trap enable flags in the PsR. The Overflow Trap Enable flag (OVT) is PsR bit <1>; the Taken Branch Trap Enable flag (BT) is PsR bit <14>. To enable generation of one of these traps, a Set Mode instruction is used to set the appropriate enable flag to "1". To disable trap generation, a Clear Mode instruction is used to clear the enable flag to "0". A Test Mode instruction can be used to examine the current state of one of these flags. The Taken Branch Trap Enable flag can be set, cleared, or tested only in system mode; the Overflow Trap Enable flag can be set, cleared, or tested in either user or system mode. Since the PsR is a local register, critical section entry is not required on trap enable flag access.

Recognition of traps by a PU is disabled when the Interrupt/Trap Enable Flag (I/TE), bit <15> in the PU's PsR, is cleared to "0". A PU with this flag cleared (set) is said to be *interrupt/trap disabled (enabled)*. When a local exception trap is generated on or an inter-PU signal trap arrives at an interrupt/trap enabled PU, it is immediately presented and recognized. Local exceptions are the data page fault, data access privilege violation, message reject, system call, operation fault, overflow, instruction page fault, and instruction access privilege violation traps. Inter-PU signal traps are the PU restart and PU preempt traps; these are generated by execution of **Res** (Restart) and **Prmpt** (Preempt) instructions on some other PU. Upon recognizing the trap, the current PsR is saved in SaveR, Current PC and Next PC are transferred to the PCQ, PsR bits <24> and <15:0> are cleared, and control is transferred to the appropriate kernel entry address. Among the PsR bits cleared are the Interrupt/Trap Enable flag, the PCQ Enable flag, the User Mode flag, the Overflow Trap Enable, flag, and the Taken Branch Trap Enable flag. Clearing the Interrupt/Trap Enable flag disable interrupt/trap recognition for the PU; the PU can re-enable interrupt/trap recognition as described in the preceding section.

The disposition of a trap which is generated or which arrives while recognition of traps is disabled depends on the trap type. An attempt to generate a local exception trap on an interrupted/trap disabled PU is an error; a PU Check Trap is generated (instead of the local exception trap) and is immediately presented to and recognized by the PU. For example, occurrence of a page fault or a message reject while the PU is disabled will cause a PU Check Trap. The local exception causing the PU Check Trap can be determined by examination of the trap source flags, as described below. If a PU Preempt Trap arrives while a PU is interrupt/trap disabled, its presentation is deferred until the PU becomes enabled; it is then presented and recognized. If a PU Restart Trap arrives while the PU is interrupt/trap disabled, it is immediately presented to and recognized by the PU. The PU Check Trap and the PU Restart Trap are called non-maskable traps because they are immediately recognized.

Trap	Trap Argument Field TrapR Bits <18:0>
PU Check, PU Restart, or PU Preempt	Unpredictable.
Data Page Fault, Data Access Privilege Violation, or Message Reject	Virtual page number referenced on the access which generated the trap.
System Call	Bits <3:0> contain the trap number, (instruction bits <3:0>). In Antares, bits <18:12> are unpredictable, and bits <11:4> contain instruction bits <15:8>
Operation Fault: illegal operation code (Illegal operation codes are listed in the Operation Fault Trap description in Section 5.8)	Bit <18> is "0", distinguishing this case from the operation privilege violation and taken branch trap cases. Bits <17:12> are unpredictable. Bits <11:4> contain instruction bits <15:8>. If instruction bits <15:12> (TrapR bits <11:8>) are "0001", this is a 16-bit illegal opcode and bits <3:0> contain instruction bits<3:0>; otherwise, this is a 4-, 8-, or 12-bit illegal opcode and bits <3:0> contain instruction bits <7:4>.
Operation Fault: operation privilege violation	Bit <18:17> are "10", distinguishing this case from the illegal operation code and taken branch trap cases. Bits <16:0> are unpredictable.
Operation Fault: taken branch trap	Bit <18:17> are "11", distinguishing this case from the illegal operation code and operation privilege violation cases. Bits <16:0> are unpredictable.
Overflow	Unpredictable.
Instruction Page Fault or Instruction Access Privilege Violation	Unpredictable, in Antares. Later versions of the Scorpius architecture may store the virtual page number referenced on the instruction fetch generating the trap in the trap argument field.

Figure 4.4. Trap Arguments

Trap Source Flags. Bits <31:22> of the TrapR contain flags which identify the trap source (bits <21:19> are reserved for future use). For a local exception, a trap source flag is set (to "1") when the exception occurs, even if the PU is interrupt/trap disabled (and a PU Check Trap results). For inter-PU signals, a trap source flag is set when the trap is presented. There is no trap source flag for the PU Check Trap. Recognition of the PU Check Trap or the PU Restart Trap results in a transfer to the non-maskable trap vector address; no other traps result in a transfer to this address. The PU Check Trap can be distinguished from the PU Restart Trap by examining TrapR bit <31>. If this bit is "1", a PU Restart Trap was recognized; if this bit is "0", a PU Check Trap was recognized, and TrapR bits <29:22> can be examined to determine which exception resulted in the PU Check Trap (assuming the Trap Source Flag field had been cleared when the PU Check Trap was recognized).

Only one trap source flag will be set when a trap is recognized. If an inter-PU signal trap (Preempt or Restart) is presented to a PU at the same time that a local exception trap is generated on that PU, the inter-PU signal trap is recognized, and the trap is ignored. Because trap generation is non-destructive, restoring the PU to its original state usually will cause the trap to be regenerated. (There is one case in which a trap may not be regenerated; a store to an interrupt-on-write page can result in a message reject trap on one instance of execution but not on another.)

Trap source flags normally are set by hardware and cleared by software. While trap source flags can be set, as well as cleared, by software, software setting of a trap source flag is ignored by the hardware and does not cause trap recognition. When the hardware sets a trap source flag, it does not clear the remaining flags. Consequently, a PU should clear a trap source flag while disabled following recognition of the trap associated with that flag.

Trap Arguments. Certain traps are accompanied by an argument which is stored in the trap argument field of the TrapR (bits <18:0>) when the trap is recognized. The contents of this field upon trap recognition are shown in Figure 4.4. The TrapR should be read only when the PU is interrupt/trap disabled. The kernel should read and clear the trap argument field (as well as clear the trap source flag) before enabling interrupt/trap recognition.

Multiple Exception Instances. The operating system must be prepared to handle multiple instances of an exception such as a page fault or an access privilege violation. For example, initiation of a SIMD loop can result in all four PUs attempting simultaneous access to a single data page. If that page is invalid, these accesses will result in simultaneous (or nearly simultaneous) generation of a page fault trap for the same virtual page on each PU. Correct and efficient operation requires that trap processing on each PU be coordinated by the operating system; a variety of approaches are possible.

4.5 Interrupt/Trap Entry Addresses

Interrupt/trap recognition causes control to be transferred (by setting an address in Current PC) to one of eight interrupt/trap entry addresses generated as follows. Recognition of an interrupt causes a PU to transfer control to the kernel at address 0xFFC00000 + 0x200 x NE + 0x100 x PUA, where NE is "0" if the interrupt does not represent a error and is "1" otherwise, and PUA is the PU Available flag, bit <0> of the PU's PsR. The Machine Check, Power/Temp., and Deadlock interrupts are called error interrupts; these interrupts are always presented to and immediately recognized by PU 0, regardless of whether or not PU 0 is interrupt/trap enabled. If one of these interrupts is recognized while PU 0 is interrupt/trap disabled, some PU state information may be lost.

Recognition of a trap causes a PU to transfer control to the kernel at address 0xFFC00400 + 0x200 x NE + 0x100 x PUA, where NE is "0" if the trap represents a error and is "1" otherwise, and PUA is the PU Available flag from the PsR. The PU Check and PU Restart traps are called error traps. The PU Check trap is generated when a local exception occurs while the PU is interrupt/trap disabled

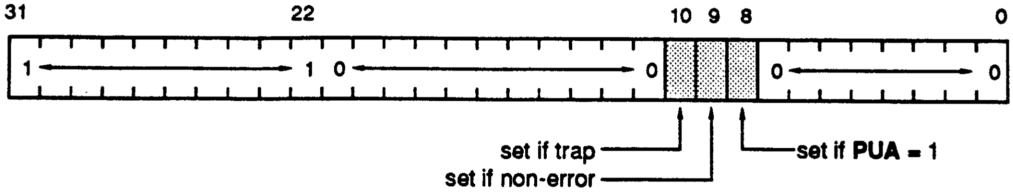


Figure 4.5. Interrupt/Trap Entry Address Formation

and is immediately recognized by the PU. The PU Restart trap is generated and immediately recognized by a PU when that PU is the target of a Restart instruction executed by another PU, usually because the PU is "hung" while interrupt/trap disabled. In both cases, some PU state information may be lost (see ***The PCQ Enable Flag*** in Section 4.6).

Separate entry addresses are provided for error and non-error interrupts and traps so that the kernel can, in the error case, bypass critical sections used in non-error trap/interrupt processing. The spacing between successive entry addresses is 256 bytes (128 instructions).

Note that interrupt and trap entry addresses can be formed by setting instruction address bits <31:22> to "1" to designate a kernel address, setting bit <8> to PUA, setting bit <9> if the interrupt or trap does not represent an error, setting bit <10> for a trap, and clearing the remaining bits, as shown in Figure 4.5.

Entry addresses for Scorpius interrupts and traps are listed below.

Interrupt/Trap	PUA	entry address
Error interrupts: Machine Check, Power/Temp. & Deadlock	"0" "1"	0xFFC00000 0xFFC00100
Non-error interrupts: IO, Message, & Event Counter Overflow	"0" "1"	0xFFC00200 0xFFC00300
Error traps: PU Check & PU Restart	"0" "1"	0xFFC00400 0xFFC00500
Non-error traps: PU Preempt, Data Page Fault, Data Access Privilege Violation, Message Reject, System Call, Operation Fault, Overflow, Instruction Page Fault, & Instruction Access Privilege Violation	"0" "1"	0xFFC00600 0xFFC00700

4.6 PU State at Interrupt/Trap Recognition

Recognition of an interrupt or a trap results in a transfer to one of the kernel addresses listed earlier. Concurrently, the contents of the two program counters, Current PC and Next PC, are moved to the PC Save Queue, PCQ[1] and PCQ[2] (assuming that the PCQE flag in the PsR was "1" at the time the interrupt or trap was recognized), and the contents of the PsR are saved in SaveR. Except for the System Call trap, the address of the instruction being executed at the time the interrupt or trap was recognized is contained in PCQ[1]. (PCQ[1] sometimes must be adjusted to this address, as discussed below.) For a System Call trap, PCQ[1] contains the address of the instruction following the the Trap instruction. For certain local exceptions, the kernel's interrupt handler must use the address in PCQ[1] to examine this instruction. On return from interrupt, execution of the interrupted program begins with the instruction whose address is contained in PCQ[1]; this instruction is called the *interrupted instruction*. This section describes the state of the machine following interrupt/trap recognition; return from interrupt processing is discussed in the next section.

The state of the machine following interrupt/trap recognition is guaranteed only in that restoring the state saved at the time of the interruption in accordance with the architectural rules for restoring state produces correct results. If the interrupted instruction is a **LdM/StM** (load/store multiple) instruction, it is generally not possible to determine from examination of the machine state how many registers have loaded or stored. On return from interrupt, execution of the **LdM/StM** instruction is repeated, not resumed.[2] If an interrupt or trap is recognized during execution of an instruction issued after issue of an asynchronous instruction (a multiply or divide instruction), the effect on machine state is as if the asynchronous instruction had completed execution prior to interrupt/trap recognition. When an overflow trap is recognized, the result causing the trap will not have been stored in the result register.

When the kernel begins execution at the interrupt/trap entry address following recognition of an interrupt or a trap, the state of the PU is as follows.

- On interrupt recognition, the pending interrupt flag in the ICR corresponding to the interrupt recognized will be set to "1'. Other pending interrupt flags in the ICR also may be set, either because the associated interrupt is pending or because the flag has not yet been cleared (see ***Pending Interrupt Flags***). If a Message Interrupt is pending, the real address of the message destination will be stored in the IAR . When an interrupt other than a Message Interrupt is recognized (and a Message Interrupt is not pending) the contents of the IAR are unpredictable.

- On recognition of a trap other than PU Check, the trap source flag in the TrapR corresponding to the trap recognized will be set to "1". All other trap source flags usually will be "0" (assuming that the kernel properly clears trap source flags prior to enabling interrupt/trap recognition). However, more than one trap source flag may be set if a nonmaskable trap (Restart or Check) is recognized while the kernel is processing a previous trap and has not yet cleared the associated trap source flag. In the case of PU Check, the trap source flag associated with the local exception whose generation while the PU was interrupt/trap

[2] Consequently, LdM/StM should not be used to transfer data from/to IO locations.

disabled caused the PU Check is set. For certain traps, a trap argument is stored in the Trap Argument Field of the TrapR (see Figure 4.4).

• If the PCQ Enable flag in the PsR was "1" at the time the interrupt or trap was recognized, the contents of the PsR have been moved to the SaveR and the contents of the Current PC and the Next PC have been moved to PCQ[1] and PCQ[2], respectively. If the PCQ Enable flag was "0", these moves do not take place and part of the PU state at interrupt/trap recognition time is lost. (The PCQ Enable flag is discussed later in this section.) In the process, PsR and PCQ flags may be set indicating that state adjustments are required before returning from interrupt.

— On recognition of a data page fault or data access privilege violation trap generated by the data access of a Load/Store Byte or a Load/Store Multiple instruction, or when an interrupt is recognized during execution of either of these instructions, the Rmod flag in the PsR (bit <31>) is set to "1" before the PsR contents are moved to SaveR. When Rmod is "1", address register adjustment is required prior to return from interrupt (see next section); when Rmod is "0", adjustment is not required.

— PCQ[1] and PCQ[2] each contain instruction address bits <31:1> in bits <31:1> and a Correction (C) flag in bit <0>. (Since Scorpius instructions always start on half-word boundaries, instruction address bit <0> is ignored.) Usually, PCQ[1] and PCQ[2] contain the addresses of the next two instruction to be executed on return from interrupt. Assuming the PCQ Enable flag was "1" when the interrupt or trap was recognized, the contents of PCQ[1] and PCQ[2] are frozen from that point up to the point at which the PCQ Enable flag (which is cleared upon interrupt/trap recognition) is set to "1" again and are unpredictable thereafter. Under certain (implementation-dependent) conditions, the C flags in PCQ[1] and/or in PCQ[2] may be set to "1" when the contents of Current PC and Next PC are moved to PCQ[1] and PCQ[2]. The C flag in PCQ[1] must be examined prior to return from interrupt to determine if the contents of PCQ[1] require adjustment (see next section). Also, the C flag in PCQ[1] must be checked when it is necessary to inspect the instruction whose execution was interrupted by interrupt/trap recognition. If the C flag in PCQ[1] is "0", no adjustment is required; PCQ[1] contains the address of the instruction being executed when the interrupt/trap was recognized. If the C flag in PCQ[1] is "1", the address of the instruction being executed at the time the interrupt/trap was recognized is (PCQ[1]) + "2". (PCQ access is discussed later in this section.)

The C flag in PCQ[2] may or may not be set on interrupt/trap recognition. However, it is never necessary to inspect this flag or to adjust the contents of PCQ[2] for correction purposes. (It is necessary to adjust the contents of PCQ[2] only when it is desired to bypass execution of the interrupted instruction on return from interrupt; see next section.)

It is not necessary to clear the C flags in either PCQ[1] or PCQ[2] prior to returning from interrupt. However, care must be taken to insure that, when the C flag in PCQ[1] is set, the contents of PCQ[1] are adjusted only once.

- PsR bits <24> and <15:0> are cleared to "0". As a result, when the kernel begins execution following interrupt/trap recognition, modes and flags have been initialized as follows.
  - — System mode is selected.
  - — Interrupt/trap recognition is disabled.
  - — Overflow and Taken Branch Trap generation are disabled.
  - — The PU Halted and PU Available flags are cleared.
  - — The PCQ Enable flag is cleared.
  - — Byte mode is selected for partial arithmetic operations.
  - — Condition codes are inherited from the user-mode PsR.

The PCQ Enable Flag. The PCQ Enable flag lets the kernel control when PsR and PC contents are to be saved in the event of an interrupt or trap. This capability is provided to facilitate operating system debugging. When the PCQ Enable flag, PsR bit <12>, is "1", recognition of an interrupt or trap causes the contents of the PsR to be transferred to the SaveR. the contents of the Current PC and Next PC to be transferred to PCQ[1] and PCQ[2], and the PCQ Enable flag to be cleared to "0". If this flag is "0" when the interrupt or trap is recognized, these transfers do not take place.[3]

The PCQ Enable flag (PCQE) is privileged. It is cleared to "0" on machine reset and on interrupt or trap recognition. It can be set or cleared (in system mode) via SetM or ClrM instructions; setting PCQE = "1" invalidates the contents of the SaveR and PCQ. PCQE, like other PsR flags, also is set or cleared when execution of a Return From Interrupt instruction pair causes the contents of the SaveR to be transferred to the PsR. The result of setting PCQE to "0" while the PU is interrupt/trap enabled (PsR bit <15> = "1") is unpredictable.

In normal operation, PCQE = "1" at all times except for two short intervals. The first interval is that between the time at which an interrupt or trap is recognized (which clears PCQE to "0") and the time at which the kernel (after saving SaveR and PCQ contents in a save area) sets PCQE to "1" again. The second interval occurs just prior to return from interrupt, when the kernel sets PCQE to "0"

[3] In Antares, when the PCQ Enable flag is "1", these transfers take place as each instruction is executed.

before restoring the PCQ from the save area.[4] Except in these intervals, PsR and PC contents will be saved if a non-maskable interrupt or trap occurs while the PU is interrupt/trap disabled . This is important in operating system debugging, since it helps, for example, find the address of the system instruction which incurred a page fault while the PU was interrupt/trap disabled and so generated a PU Check Trap. If an interrupt or a trap is recognized during an interval in which PCQE = "0", the contents of the PsR and PCs at recognition time are lost.

It may not be possible to determine, by an examination of the state of the machine, whether or not a non-maskable interrupt or trap was recognized during the interval in which PCQE = "0". Since the PsR is not saved in this case, the value of PCQE at the time the interrupt or trap was recognized cannot be determined. However, assuming the kernel code executed in this interval is correct, no local exceptions, and consequently no PU Check Trap, will be generated in this interval. While a non-maskable interrupt or a restart (inter-PU) trap can occur, losing PU state information in these cases usually is not the severe debugging handicap that losing information in the PU Check Trap case can be.

The PU Available Flag. On recognition of any particular interrupt or trap, control is transferred to one of two kernel entry addresses, depending on the setting of the PU Available (PUA) Flag in the PsR. This flag is set only by software (using a Set Mode instruction). It is cleared when the PU is a target of a Start instruction, when an interrupt or a trap is recognized, or on return from interrupt if bit <0> of SaveR is "0".

The intended interpretation of the PUA flag is that, when set, the kernel is not required to save PU state on interrupt/trap recognition or restore state prior to returning from interrupt, reducing interrupt processing overhead. (The PsR and Current and Next PCs are saved on recognition and restored on execution of a return from interrupt instruction pair. When switching address spaces, local register saving and restoring, except for the SaveR and PCQ registers, is assumed to be unnecessary if PUA is set.) To effect this, PU activities should, on completion of execution, set PUA prior to halting. From a hardware standpoint, the PUA flag is used in kernel entry address selection and may be used in selecting a PU to process an external or event counter overflow interrupt. In Antares, this PU selection considers only PU halt flags; the hardware interrupt handler will attempt to assign an interrupt to the highest-numbered halted PU. Later implementations may, in addition, consider PUA flags when selecting a PU.

PU State Saving. The steps performed by the kernel in saving state following recognition of an interrupt or trap might include the following.

- Execute a **Lock** instruction to clear the system mode semaphore; this insures that one and only one PU at a time will be attempting to access

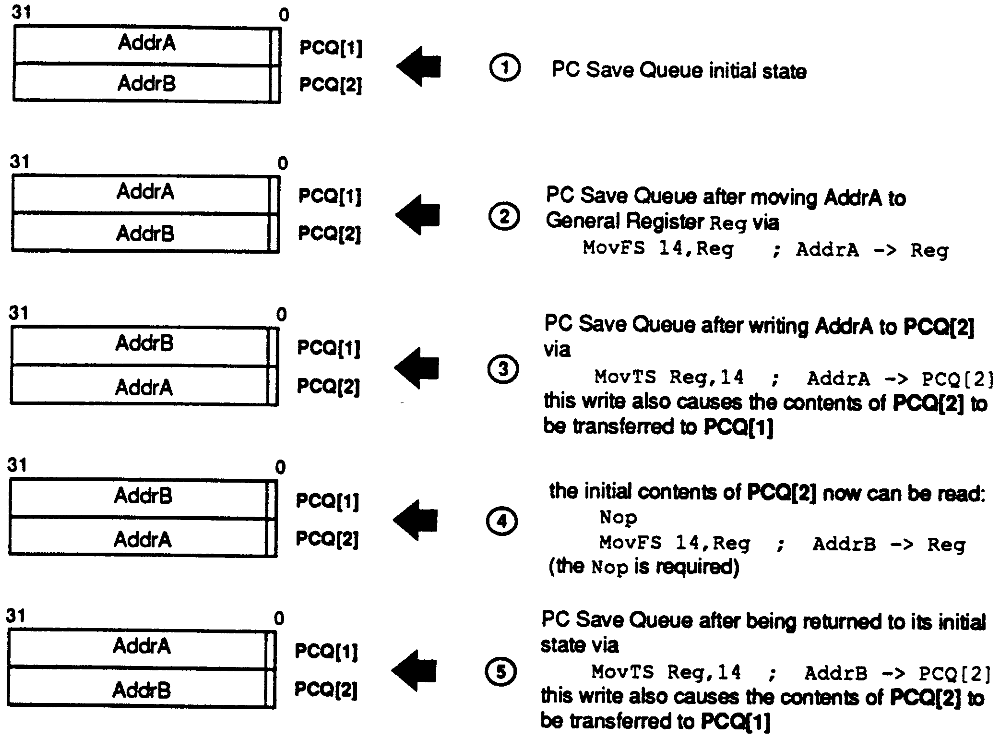
[4] At least in Antares, where the transfer of the PCs to the PCQ is done on every instruction and must be inhibited in order to restore the PCQ.

the save area pointer, Scratch Register, and, on an interrupt, the ICR and IAR.

- Save the value of general register R0 in the Scratch Register (ScR).
- Construct, in R0, the address of a pointer to a register save area, and read the pointer. (Note that prefixing cannot be used to construct this address because PfxR contents have yet to be saved.)
- Execute a Store Multiple instruction to save registers 1-15 in the save area.
- Read and save the contents of the SaveR, read and save the contents of the PCQ, and set the PCQ Enable flag to "1". (This may be skipped on some kernel service calls which, for efficiency, are processed without resetting the PCQ Enable flag — and without the corresponding saving of SaveR and PCQ contents.)
- Retrieve R0's original value from the ScR, and store it in the save area.
- If the PfxR will be used, read and save its value. Also, if a Multiply or a Divide instruction will be executed, read and save the value in ProdR or RemR.
- On an interrupt, read the ICR and select an interrupt pending flag. If the selected interrupt is a Message Interrupt, read the message address from the IAR. Clear the interrupt pending flag (as described in ***Pending Interrupt Flags***). Note that the IAR should be read before the interrupt pending flag is cleared.
- Execute an **Unlk** instruction to release (set) the system mode semaphore flag.
- On a trap, read the trap source flag from the TrapR and clear it, and, if appropriate, read the trap argument from TrapR and clear the trap argument field (TrapR bits <18:0>), before enabling interrupt/trap recognition.

On an address space switch, it may be necessary to save Event Counters 1 and 2, together with the event counter control flags from the ICR (if event counting was active), and the DSA and ASN fields from the IdR (if not recorded elsewhere).

PC Save Queue Access. The PCQ is read when saving state prior to setting the PC Queue Enable flag or prior to an address space switch, or whenever the interrupted instruction must be examined, and is written when restoring state or performing an address adjustment. (When the Rmod flag in the SaveR is set, the kernel must determine if the interrupted instruction is an **LdM/StM** or an **LdB/StB** and adjust the base register accordingly prior to return from interrupt. The address of the interrupted instruction is obtained by reading PCQ[1] using a Move From Special instruction and adjusting the address read if the C flag in bit <31> is set.)



① PC Save Queue initial state

② PC Save Queue after moving AddrA to General Register `Reg` via

```
    MovFS 14,Reg    ; AddrA -> Reg
```

③ PC Save Queue after writing AddrA to **PCQ[2]** via

```
    MovTS Reg,14   ;  AddrA -> PCQ[2]
```

this write also causes the contents of **PCQ[2]** to be transferred to **PCQ[1]**

④ the initial contents of **PCQ[2]** now can be read:

```
    Nop
    MovFS 14,Reg   ;  AddrB -> Reg
```

(the `Nop` is required)

⑤ PC Save Queue after being returned to its initial state via

```
    MovTS Reg,14   ;  AddrB -> PCQ[2]
```

this write also causes the contents of **PCQ[2]** to be transferred to **PCQ[1]**

Figure 4.6. Accessing **PCQ** Contents in Antares

In Antares, the PC Save Queue, PCQ[1] and PCQ[2], is a FIFO register pair in which the contents of PCQ[2] are moved to PCQ[1] whenever PCQ[2] is written. The contents of PCQ[1] (only) can be read via a Move From Special instruction; PCQ[2] (only) can be written via a Move To Special instruction. To read the contents of the PCQ while preserving its initial state, the procedure shown in Figure 4.6 can be used.

4.7 Return From Interrupt

Returning control to an interrupted program following interrupt/trap processing (*return*), initiating program execution in a newly-created address space (*startup*), or reinitiating program execution after an address space switch (*switch*) is effected via a sequence of operations called *return from interrupt.* Return involves only a single PU, while an address space startup or switch requires the coordinated action of all four PUs. When an instance of the kernel executing in one PU decides that execution in the currently-active address space is to be suspended in preparation for a startup or switch, it will cause the other PUs to trap (via a Preempt instruction) so that they can save local state. The four PUs must decide which PU is responsible for saving global state and performing other serial operations. In all cases, the actual transfer of control from the kernel to the program whose execution is being initiated or resumed is effected by executing a pair of Return From Interrupt instructions. These instructions transfer the contents of the SaveR to the PsR, transfer the contents of the PCQ to the Current PC and Next PC, and cause execution to continue at the address in the Current PC.

This section describes the operations in a return, startup, or switch. An address space startup or a swap usually requires that execution of the currently-active address space be suspended and its state saved. Some implementations may require additional steps; in Antares, it is necessary to flush the TB and caches on a startup or a switch (see Chapter 3).

Return. Returning to the interrupted program after processing an interrupt or trap involves the following steps.

1. If the Correction flag in PCQ[1] is "1", the address in PCQ[1] must be corrected by adding 2 (unless this correction was made during interrupt/trap analysis). Use the procedure illustrated in Figure 4.6 to read and write PCQ[1]. If the PCQ and SaveR contents were saved, the PCQ and SaveR accesses required in this and subsequent steps are made to the save area.

2. If it is desired to bypass reexecution of the interrupted instruction on return from interrupt, increment both PCQ[1] and PCQ[2] by 2. One case in which this is required is in exiting from trap processing for and simulation of a non-implemented instruction. Note that this is <u>not</u> required in exiting trap processing for a System Call instruction.

3. If the Rmod flag of the interrupted program (SaveR bit <31>) is"1", adjustment of the address register of a Load/Store Multiple or a Load/Store Byte instruction is required. (Note that moving the contents of SaveR to a general register via a Move From Special instruction will set the N condition code if the Rmod flag is "1".) The adjustment procedure is as follows.

   - Get the instruction address from PCQ[1], save bit <1> (i.e., the halfword index), and use this address to read the word containing the interrupted instruction. Use the halfword index to select the appropriate halfword.

   - Extract the operation code field. For Load/Store Multiple and Load Store Byte, this is instruction halfword bits <15:8>.

   - Extract the RegA field, bits <3:0>, of the instruction halfword. This is the address register number; subtract it from the starting address of the register save area to obtain a pointer *p* to the value to be corrected.[5]

   - Examine the operation code. If the instruction is a Load/Store Byte instruction, the address register must be decremented before

[5] Assuming that the general registers were stored beginning at the first word of the save area using a Store Multiple instruction (which decrements).

returning. Read memory location *p*, subtract 1, and store the result in location *p*.

- If the instruction is a Load/Store Multiple instruction, extract the RegB field, bits <7:4>, of the instruction halfword. This is the number of registers loaded or stored.

- If the instruction is a Load Multiple instruction, the address register must be decremented by the RegB field value, multiplied by 4, before returning. Read memory location *p*, subtract 4 x RegB, and store the result at location *p*.

- If the instruction is a Store Multiple instruction, the address register must be incremented by the RegB field value, multiplied by 4, before returning. Read memory location *p*, add 4 x RegB, and store the result at location *p*.

- Clear the Rmod flag in SaveR.

4. Restore the ProdR, RemR, and PfxR register contents from the register save area, using a load instruction to read the value stored in the appropriate save area memory location into a general register and a Move To Special instruction to move that value to the special register.

5. If the PCQ Enable flag was set and the SaveR and PC contents saved on entry, clear the PCQ Enable flag and restore the SaveR and PCQ contents from the save area.

5. Restore the general registers by executing a Load Multiple instruction.

6. Execute a pair of Return From Interrupt (**RtI**) instructions; these must be executed "back-to-back" (machine operation otherwise is unpredictable). The first **RtI** instruction moves the contents of PCQ[1] to the Current PC. The second **RtI** moves the contents of PCQ[2] to the Next PC, restores PsR from SaveR, and (unless the Halt flag in the PsR is "1") causes execution to continue at the address in the Current PC.

The PU must be interrupt/trap disabled when the **RtI** instruction pair is executed. (Typically, the kernel will disable interrupt/trap recognition while performing the entire return from interrupt sequence.) When register save areas are dynamically allocated and deallocated on a global basis,[6] the kernel must adjust a free area pointer before restoring general registers; this (and save area allocation) must be done in a critical section controlled by **Lock** and **Unlk** instructions.

Switch. An address space switch is initiated by a single PU as the result of an event such as a System Call trap suspending or terminating execution in the

[6] As opposed to a scheme using a single fixed save area for each PU. In a fixed save area scheme, PU save area contents are moved to and from address space save areas on switches.

currently-active address space or an external (timer) interrupt signalling quantum end. The initiating PU causes the other PUs to trap by executing a Preempt instruction. Local (PU) state saving can be done independently by each PU, while global state saving is done by a single controlling PU. Assuming the initiating and controlling PU is PU 0, the basic steps involved in an address space switch are as follows.

1. PU 0 disables presentation of external interrupts and generation of event counter overflow interrupts. It then checks to see if there are any interrupts pending and, if so, enables interrupt/trap recognition and processes any pending interrupts before proceeding.
2. PU 0 executes a Preempt instruction with PUs 1-3 as targets, advising each target PU (via a memory-based or register-based communication mechanism) that a switch is to take place, and then waits for PUs 1-3 to halt. (Careful coordination of kernel activities is required here.)
3. PUs 1-3 save their local state and halt. (PU 0 saved its local state prior to processing the interrupt or trap causing the switch.)
4. PU 0 saves global state and performs any necessary housekeeping associated with an address space switch. In Antares, the data cache must be flushed, and the instruction cache and Translation Buffer invalidated. The procedure for this is described in Chapter 3, and the cache control instruction used are described in Chapter 6.
5. PU 0 obtains the DSA | ASN for the address space to be reactivated and stores it in the IdR, and restores global state.
6. PU 0 obtains the local save area addresses for PUs 1-3, sends these to the appropriate PUs, and returns these PUs to execution via Start or Resume instructions.
7. PU 0 enables presentation of external interrupts and, if required, enables event counter overflow interrupt generation.
8. PUs 0-3 execute the steps described under ***Return***.

Details of this procedure will depend on the kernel design and on the hardware implementation.

Startup. To initiate execution in a new address space, the kernel first suspends execution in the currently-active address space as described above. It then creates a page table for the new address space; the address of the first word of the page table directory defines the DSA | ASN field for the address space. Typically, the page table includes an entry for several shared pages, one of which contains a startup task; the kernel starts address space execution by having one PU "return" to the startup task's entry address while the remaining PUs "return" to halt state. Assuming PU 0 is the controlling PU and PUs 1-3 are halted, the final steps in this process, after global state has been initialized, might be as follows.

1. PU 0 sends a "dispatch halted PU" message to PUs 1-3.

2. PUs 1-3 each set SaveR bits <24> (Halt flag), <15> (Interrupt/Trap Enable flag), <13> (User Mode flag), and <0> (PU Available flag). All other SaveR flags are set to "0". No other local registers need be initialized.[7]

3. PUs 1-3 each execute an **RtI** instruction pair, causing them to halt in user mode.

4, PU 0 sets SaveR bits <15> (Interrupt/Trap Enable flag) and <13> (User Mode flag). All other SaveR flags are set to "0". PCQ[1] is set to the startup task's entry address, and PCQ[2] is set to that address + 2.

5. PU 0 enables external interrupt presentation and, if appropriate, event counter overflow interrupt generation, and executes an **RtI** instruction pair to initiate startup task execution.

The startup task, which begins execution in PU 0 in this example, can activate other PUs via a Start instruction. In addition to activating these PUs, execution of a Start instruction effectively causes its target PUs to flush the contents of their instruction queues. Note that an attempt to activate a PU using a Resume instruction would cause it to initiate an instruction fetch at the address contained in PCQ[1] at the time of return.

4.8 Interrupt/Trap Summary

This section provides a summary description of interrupts and traps. This summary includes, for each interrupt and trap, the interrupt pending flag or trap source flag bit position, the argument (value, if any, stored in the IAR or TrapR), the cause(s), generation or presentation and recognition controls, and notes on analysis.

[7] Although PUs 1-3 will not execute instructions if dispatched in halt state,, they will, in Antares, issue instruction fetches for the addresses loaded into Current PC and Next PC from the PCQ. Unless the PCQ is explicitly set with addresses valid in the new address space, these fetches can cause useless cache and TB misses (although, because the PU is halted, they cannot cause exceptions). To avoid this, the PCQ for all four PUs should be set with the entry address and entry address + 2 (or some other valid address) prior to return from interrupt. (Note that by adopting the convention that the instruction at the initial entry point always is a "Wait 1" instruction, all four PUs could be dispatched in run state.)

Interrupts

Name:	**Machine Check**
Cause(s):	• Inter-Processor Bus (IPB) error. • Internally-detected hardware error.
Pending Flag:	ICR Bit <31>
Argument:	None
Entry Addr.:	`0xFFC00000` + `0x100` x PUA
Controls:	None (non-maskable)
Notes:	This interrupt always is presented to and recognized by PU 0.

Name:	**Power/Temp**
Cause(s):	Power failure or internal temperature over limit
Pending Flag:	ICR Bit <30>
Argument:	None
Entry Addr.:	`0xFFC00000` + `0x100` x PUA
Controls:	None (non-maskable)
Notes:	The Power/Temp is implementation-dependent; voltage or temperature sensing may or may not be provided in a particular implementation. This interrupt always is presented to and recognized by PU 0.

Interrupts (continued)

Name:	**Deadlock**
Cause(s):	Hardware deadlock detection.
Pending Flag:	ICR Bit <29>
Argument:	None
Entry Addr.:	`0xFFC00000 + 0x100` x PUA
Controls:	None (non-maskable)
Notes:	This interrupt is generated when the state of each of the four PUs, as reflected in the Global Status Register (GSR), is either halt or wait (see discussion of PU states in Chapter 5), and remains unchanged for a fixed time. (This is the time required to propagate state changes from the PUs to the GSR., and is implementation-dependent.)

Name:	**IO**
Cause(s):	IO interrupt
Pending Flag:	ICR Bit <28>
Argument:	None.
Entry Addr.:	`0xFFC00200 + 0x100` x PUA
Controls:	Presentation of this interrupt is disabled for the CPU when the External Interrupt Enable flag, ICR bit <5>, is "0". Recognition of this interrupt is disabled for a PU when the Interrupt/Trap Enable flag in the PsR of that PU is "0".
Notes:	If an IO interrupt arrives while external interrupts are disabled, it is rejected. Rejection of an IO interrupt simply causes the IO interrupt signal to be ignored; if external interrupt recognition is later enabled and the IO interrupt signal remains asserted, it can be recognized at that time. An IO interrupt is rejected, even if external interrupts are enabled, if an earlier-arriving IO interrupt still is pending (ICR bit <28> = "1") when it is presented. (See ***Pending Interrupt Flags***.) This interrupt is directed to a PU in accordance with the PU selection rules for the implementation.

Interrupts (continued)

Name: **Message**

Cause(s): Interrupt-on-write access to CPU's local memory

Pending Flag: **ICR** Bit <27>

Argument: On recognition of a message interrupt, the real (byte) address of the interrupt-on-write access (i.e., the operand address of the store instruction initiating that access) is contained in the IAR.

Entry Addr.: 0xFFC00200 + 0x100 x PUA

Controls: Presentation of this interrupt is disabled for the CPU when the External Interrupt Enable flag, ICR bit <5>, is "0". Recognition of this interrupt is disabled for a PU when the Interrupt/Trap Enable flag in the PsR of that PU is "0".

Notes: If a message interrupt arrives while external interrupts are disabled, it is rejected. Rejection of a message interrupt causes a Message Reject trap to be generated on the PU attempting to store to an interrupt-on-write page. A message interrupt is rejected, even if external interrupts are enabled, if an earlier-arriving message interrupt still is pending (ICR bit <27> = "1") when it is presented. (See ***Pending Interrupt Flags***.) This interrupt is directed to a PU in accordance with the PU selection rules for the implementation.

Name: **Event Counter Overflow**

Cause(s): Overflow of Event Counter 1 or Event Counter 2.

Pending Flag: ICR Bit <26> (Event Counter 1)
ICR Bit <25> (Event Counter 2)

Argument: None

Entry Addr.: 0xFFC00200 + 0x100 x PUA

Controls: Generation of an overflow interrupt for Event Counter 1 is enabled by setting the Event Counter 1 Overflow Interrupt Enable flag, ICR bit <4> to "1", and disabled by setting this flag to "0". Generation of an overflow interrupt for Event Counter 2 is enabled by setting the Event Counter 2 Overflow Interrupt Enable flag, ICR bit <3> to "1", and disabled by setting this flag to "0". Recognition of this interrupt is disabled for a PU when the Interrupt/Trap Enable flag in the PsR of that PU is "0".

Notes: This interrupt is directed to a PU in accordance with the PU selection rules for the implementation. If an event counter overflow interrupt is generated for an event counter while a counter overflow interrupt is still pending for that counter, the newly generated interrupt simply is discarded. (This should occur only if the pending interrupt was ignored by the kernel.) Event counters are discussed in Chapter 7.

Traps

Name: **PU Check**

Cause(s): Attempt to generate a local exception trap on a PU while that PU is interrupt/trap disabled (PsR bit <15> = "0")

Source Flag: There is no TrapR source flag for the PU Check trap itself; the trap source flag corresponding to the local exception which resulted in the PU Check trap will be set.

Argument: None

Entry Addr.: `0xFFC00400 + 0x100` x PUA

Controls: None (non-maskable)

Notes: A PU Check trap is generated when a data page fault, data access privilege violation, message reject, system call, operation fault, overflow, instruction page fault, or instruction access privilege violation occurs while a PU is interrupted/trap disabled. The TrapR flag for the local exception is set to "1" and control transferred to the non-maskable trap entry address. Control also is transferred to this address on a PU Restart trap. The two cases generally can be distinguished by examining the PU Restart trap source flag in TrapR. Note that two trap source flags can be set at the time a PU Check trap is recognized if the PU Check trap occurred while the PU was processing an earlier trap and had not yet cleared the source flag for that trap. When a PU Check trap is recognized, the contents of the PsR and Current and Next PCs are not saved (see Section 4.6).

Name: **PU Restart**

Cause(s): Execution of a PU Restart (**Res**) instruction by another PU; this trap is generated, presented to, and recognized by each PU specified as a target of the **Res** instruction.

Source Flag: TrapR bit <31>

Argument: None

Entry Addr.: `0xFFC00400 + 0x100` x PUA

Controls: None (non-maskable)

Notes: Control is transferred to this entry address on a PU Check trap as well as on a PU Restart trap. The two cases generally can be distinguished by examining the PU Restart trap source flag in TrapR, which will be set to "1" on a PU Restart trap. If the PU is interrupted/trap disabled when a PU Check trap is recognized, the contents of the PsR and Current and Next PCs are not saved (see Section 4.6). (This is the expected case; the **Res** instruction is provided to interrupt a PU which is "hung" while interrupt/trap disabled.) The Restart instruction is discussed in Section 5.3.

Traps (continued)

Name: **PU Preempt**

Cause(s): Execution of a PU Preempt (**Prmpt**) instruction by another PU; this trap is generated and presented to each PU specified as a target of the **Prmpt** instruction, and recognized when the target PU becomes interrupt/trap enabled.

Source Flag: TrapR bit <30>

Argument: None

Entry Addr.: `0xFFC00600 + 0x100` x PUA

Controls: Recognition of this trap is disabled when the Interrupt/Trap Enable flag in the PsR is cleared to "0".

Notes: This is the only maskable trap, in the sense that its recognition may be deferred. Execution of the **Prmpt** instruction does not complete until each target PU has become interrupt/trap enabled and recognized the Preempt trap. In Antares, all target PUs must simultaneously be enabled before preemption takes place. (See Section 5.3).

Name: **Data Page Fault**

Cause(s): Attempted operand load or store access to

- a segment whose directory entry is invalid (V flag = "0"),
- a page whose page table entry is invalid (V flag = "0"), or
- a page whose page table entry flags are set to an illegal combination: read-only and interrupt-on-write (RO="1" & IW="1") or cacheable and interrupt-on-write (NC="0" & IW="1").

Source Flag: TrapR bit <29>

Argument: The virtual page number from the address of the operand to which access was attempted is contained in TrapR bits <18:0>

Entry Addr.: `0xFFC00600 + 0x100` x PUA

Controls: None (non-maskable)

Notes: If a data page fault is encountered while the PU is interrupt/trap disabled, a PU Check trap occurs.

Traps (continued)

Name: **Data Access Privilege Violation**

Cause(s):

- Attempted operand load or store access in user mode (PsR bit <13> = "1") to a system page (page whose page table block entry has its S flag set to "1") or to the kernel region (i.e., operand address in the range 0XFFC00000 to 0XFFFFFFFF).
- Attempted operand store access to a read-only page (page whose page table entry has its RO flag set to "1").
- Attempted operand access by a load byte or store byte instruction to a non-cached page (page whose page table entry has its NC flag set to "1").

Source Flag: TrapR bit <28>

Argument: The virtual page number from the address of the operand to which access was attempted is contained in TrapR bits <18:0>

Entry Addr.: 0xFFC00600 + 0x100 x PUA

Controls: None (non-maskable)

Notes: If a data access privilege violation is encountered while the PU is interrupt/trap disabled, a PU Check trap occurs. Analysis of this trap may require examination of the operation code of the interrupted instruction as well as of the directory entry and page table entry associated with the address of the operand to which access was attempted. Accessing the interrupted instruction is described in Section 4.6.

Name: **Message Reject**

Cause(s): A store access to a local or remote page marked "interrupt-on-write" was rejected either because external interrupt presentation was disabled (ICR bit <5> = "0") or because a message interrupt already was pending (ICR bit <27> = "1').

Source Flag: TrapR bit <27>

Argument: The virtual page number of the "interrupt-on-write" page to which access was attempted is contained in TrapR bits <18:0>

Entry Addr.: 0xFFC00600 + 0x100 x PUA

Controls: None (non-maskable)

Notes: If a message reject is encountered while the PU is interrupt/trap disabled, a PU Check trap occurs. Consequently, the kernel must open an "enabled window" to send a message using the interrupt-on-write mechanism. If the kernel desires to become interrupt/trap enabled while insuring that no interrupts occur while it is enabled, it must disable external interrupt presentation and event counter overflow interrupt generation and process all pending interrupts before setting its interrupt/trap enable flag. The processing of a store access to an "interrupt-on-write" page is described in Section 3.7.

Traps (continued)

Name: **System Call**

Cause(s): Execution of a **Trap** instruction

Source Flag: TrapR bit <26>

Argument: Bits <3:0> of the TrapR contain the trap number field from the instruction (instruction bits <3:0>). The contents of TrapR bits <18:4> are unpredictable.

Entry Addr.: `0xFFC00600` + `0x100` x PUA

Controls: None

Notes: When a System Call trap is presented to a PU, PCQ[1] contains the address of the instruction following the Trap instruction. An attempt to execute a **Trap** instruction while the PU is interrupt/trap disabled results in a PU Check Trap.

Name: **Operation Fault**

Cause(s):

- Attempted execution of an illegal (undefined) operation code
- Operation privilege violation
  - — attempted user-mode execution of a privileged instruction
  - — attempted user-mode access to PsR bits <15:8>
  - — attempted user-mode access to a privileged special register (special registers 0 through 4 or 8 through 14)
  - — attempted user-mode access to a non-existent special register (special registers 7 or 15)[8]
- Execution of a taken branch or a jump instruction while taken branch trap generation is enabled.

Source Flag: TrapR bit <25>

Argument: The three causes of an operation fault can be distinguished by examining the two high-order bits of the trap argument field, TrapR bits <18:17>, as shown in Figure 4.7.

- For an Operation Fault trap resulting from attempted execution of an illegal operation code, TrapR bit <18> is "0", bits <17:12> are unpredictable, and bits <11:4> contain instruction bits <15:8>. The contents of TrapR bits <3:0> depend on the operation code length, as follows:
  - — if instruction bits <15:12> (TrapR bits <11:8>) are "0001", the operation code is a 16-bit code, and TrapR bits <3:0> contain instruction bits <3:0>.

[8] The result of an attempted system-mode access to a non-existent special register is undefined.

Traps (continued)

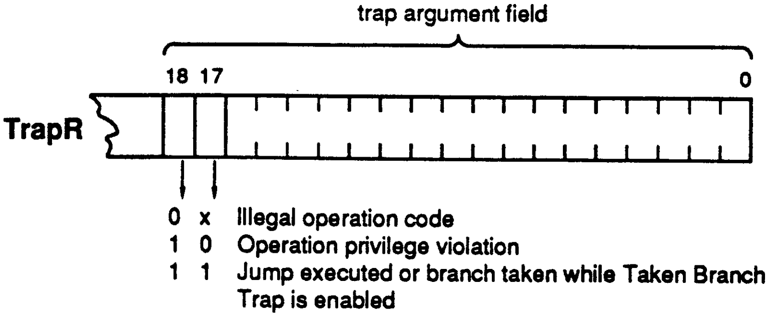

trap argument field
18 17
0
TrapR
0 x Illegal operation code
1 0 Operation privilege violation
1 1 Jump executed or branch taken while Taken Branch Trap is enabled

Figure 4.7. Distinguishing Operation Fault Causes

— if instruction bits <15:12> (TrapR bits <11:8>) are other than "0001", the operation code is a 4-, 8-, or 12-bit code and TrapR bits <3:0> contain instruction bits <7:4>.

• For an Operation Fault trap resulting from attempted execution of an privileged operation while in user mode, TrapR bits <18:17> are "10", and bits <16:0> are unpredictable.

• For an Operation Fault trap resulting from execution of a taken branch or a of a jump instruction while the Taken Branch Trap is enabled, TrapR bits <18:17> are "11", and bits <16:0> are unpredictable.

Entry Addr.: `0xFFC00600 + 0x100` x PUA

Controls: Generation of a trap on a taken branch or a jump instruction is enabled by setting the Taken Branch Trap Enable flag, PsR bit <14>, to "1" and disabled by setting this flag to "0". Illegal operation code or operation privilege violation traps are non-maskable.

Notes: Illegal operation codes are codes which are undefined or for which the corresponding instruction is not implemented on a particular Scorpius CPU and so must be emulated. (See Appendix A.)

When an Operation Fault trap results from an attempt to execute an illegal operation code, a maximum of 12 instruction bits are stored in TrapR, even in the case of a 16-bit operation code. However, since all 16-bit illegal opcodes have instruction bits <15:12> equal to "0001" and instruction bits <7:4> (not stored in TrapR) equal to "0000", this case can be identified by examining the 12 bits which are stored in TrapR.

When the Taken Branch Trap Enable flag (PsR bit <14>) is set to "1", a taken branch trap is generated on execution of a branch (**Bcc**) or jump (**Jmp**, **JmpL**, **JmpR**) instruction. On recognition of this trap, the branch or jump instruction will not have been executed, and its address is available in PCQ[1]. In Antares, a taken branch trap is not generated on execution of the other instructions which cause a transfer of control: System Call, (**Trap**), Start (**Strt**), or Return from Interrupt (**RtI**).

Later versions of the architecture may extend the Taken Branch Trap to include the **Trap** instruction; this will mean that, unlike Antares, two trap source flags could simultaneously be set at trap recognition time. One reason for inhibiting taken branch trap generation on **RtI** execution is the requirement that a PU be interrupt/trap disabled when executing an **RtI** instruction.

Detection of a operation fault while the PU is interrupted/trap disabled results in a PU Check trap.

Name: **Overflow**

Cause(s):

- Execution of an **Add**, **AddI**, **AddC**, **Neg**, **Sub**, **SubI**, or **SubC** instruction which sets the V condition code flag to "1"
- Attempt to divide by zero or, for **Div**, **DivE**, and **DivUE**, an attempt to divide which would produce a quotient which would not fit in the result register.

Source Flag: TrapR bit <24>

Argument: None

Entry Addr.: 0xFFC00600 + 0x100 x PUA

Controls: Generation of an Overflow Trap caused by setting the V condition code flag can be enabled by setting the Overflow Trap Enable flag (PsR bit <1>) to "1", and disabled by setting this flag to "0". Generation of an overflow trap caused by dividing by zero or by attempting to produce a quotient which would not fit in the result register cannot be disabled.[9]

Notes: Detection of overflow while the PU is interrupted/trap disabled results in a PU Check trap. To determine the trap cause, the operation code of the interrupted instruction must be examined (Section 4.6). When an arithmetic operation results in Overflow Trap recognition , no result is stored for that operation.

While division is asynchronous, the trap caused by divide overflow is precise; upon recognition, the divide instruction is the interrupted instruction (Section 4.6), and no instruction fetched after the divide will have been executed.

[9] CPU architectures frequently define divide by zero to be non-maskable, in the sense that the fixed-point overflow trap enable flag has no effect on generation and recognition of a divide-by-zero trap. Some architectures treat divide overflow in the same way as divide-by-zero, while others permit divide overflow to be masked. By defining both to be non-maskable, Scorpius effectively leaves the choice to software. The kernel, on recognizing an Overflow Trap caused by attempted execution of a divide instruction, can choose to ignore the exception or can choose to process it.

Traps (continued)

Name: **Instruction Page Fault**

Cause(s): Attempted instruction fetch access to

- a segment whose directory entry is invalid (V flag = "0"),
- a page whose page table entry is invalid (V flag = "0"), or
- a page whose page table entry flags are set to an illegal combination: read-only and interrupt-on-write (RO="1" & IW="1") or cacheable and interrupt-on-write (NC="0" & IW="1").

Source Flag: TrapR bit <23>

Argument: None. (Future implementations may store the instruction address associated with the fault in TrapR).

Entry Addr.: 0xFFC00600 + 0x100 x PUA

Controls: None (non-maskable)

Notes: The address associated with the instruction page fault can be obtained from PCQ[1]. If the C bit in PCQ[1] is set to "1", this address requires adjustment (see Section 4.6). If an instruction page fault occurs while the PU is interrupt/trap disabled, a PU Check trap results.

Name: **Instruction Access Privilege Violation**

Cause(s): Attempt to fetch an instruction in user mode from a system page (page whose page table entry has its S flag set to 1) or from the kernel region (i.e., instruction fetch address in the range 0XFFC00000 to 0XFFFFFFFF).

Source Flag: TrapR bit <22>

Argument: None. (Future implementations may store the instruction address associated with the violation in TrapR).

Entry Addr.: 0xFFC00600 + 0x100 x PUA

Controls: None (non-maskable)

Notes: The address of the instruction causing the violation can be obtained from PCQ[1]. If the C bit in PCQ[1] is set to "1", this address requires adjustment (see Section 4.6). When the violation is caused by a sequential instruction fetch (i.e., a "page-crosser"), the address in PCQ[1] is the address of the system page or kernel region instruction. When the violation is caused by a branch or jump from a user page to a system page or to the kernel region, PCQ[1] contains the address of the branch or jump instruction.[9] If an instruction access privilege violation occurs while the PU is interrupt/trap disabled, a PU Check trap results.

4.9 Interrupt/Trap Processing in Antares

This section describes the hardware processing of interrupts and traps in Antares from generation through recognition. (To be added.)

[9] In Antares, when an instruction access privilege violation is caused by a taken branch or a jump to a system page or the kernel region, the address of the <u>target</u> instruction is stored in PCQ[1]; the address of the branch or jump instruction is lost.

5. Inter-PU Communication and Coordination

5.1 Introduction

Instructions for the initiation and coordination of concurrently-executing activities on Scorpius PUs can be divided into three classes: broadcast, inter-PU trap, and semaphore[1]. Using broadcast instructions, one PU can send data values and activity starting addresses to one or more other PUs, and wait for other PUs to complete activity execution. Inter-PU trap instructions are a broadcast instruction variant and permit a PU executing in system mode to cause one or more other PUs to trap, as when preparing for a task switch. Semaphore instructions clear and set a semaphore in the Global Status Register. If a PU attempts to clear a semaphore which already is cleared, its execution is blocked until the semaphore is set. Semaphore instructions are used to control critical regions in programs. (A critical region is a code sequence, such as an enqueue operation, which should be executed by only one PU at a time.)

This chapter describes broadcast, inter-PU trap, and semaphore instructions and their use, discusses the PU states which may result from execution of these instructions, and describes the CPU's deadlock detection mechanism. Summary descriptions of these instructions and instruction formats are given in Chapter 8.

5.2 Broadcast Instructions

The PU Mask Field. Start, Resume, and Send instructions permit a PU to send an instruction address or a data value to other PUs in a single operation.

[1] Concurrently-executing instruction sequences can correspond to any of several program entities: expressions, statements, functions, etc. Whatever their correspondence, these sequences are referred to as *activities* in this discussion.

The PUs receiving the address or data value are called the *targets* of the instruction, and are specified by a 4-bit PU Mask field in the instruction. This field has the form $b_3b_2b_1b_0$, where b_i is "0" if PU i is a target of the instruction and "1" otherwise. In these three instructions, the PU Mask bit corresponding to the PU issuing the instruction is ignored (a PU cannot send itself an address or a data value). The Wait instruction also uses the PU Mask field; if the PU Mask bit corresponding to the PU issuing the Wait instruction is "0", the instruction performs a halt operation. (The inter-PU trap instructions Preempt and Restart use the PU Mask field to specify which PUs are to be preempted or restarted.). The result of issuing a broadcast instruction with no target PUs specified (PU Mask = 1111B) is unpredictable.

The Wait Instruction. This instruction has the form

Wait PUMask

and performs two different operations: halt and synchronize.

Halt Operation. When a Wait instruction is issued with the PU Mask bit corresponding to the issuing PU set to "0" (i.e., when the PU specifies itself as a target), the remaining bits of the PU Mask are ignored, and PU execution is halted. (A Wait instruction with PU Mask = 0000_2 will cause any PU on which it is issued to halt.) The Hlt flag (PsR bit <24>) is set to "1" and instruction issue is halted with the Current PC containing the address of the instruction immediately following the Wait instruction. PU execution can be recontinued in one of the following ways (in addition to the transfer to an interrupt/trap entry address resulting from recognition of an interrupt or an inter-PU trap).

- Execution by some other PU of a Resume instruction specifying the halted PU as a target; execution resumes at the address in Current PC.
- Execution by some other PU of a Start instruction specifying the halted PU as a target; execution continues at the address contained the in the register specified by the Start instruction.
- Return from interrupt with the Hlt flag cleared to "0"; execution continues at the address in Current PC.

In Antares, the hardware interrupt router attempts to assign an external or a message interrupt to the highest-number halted PU. When the state of a PU (other than PC and PsR contents) is not required for execution of subsequent activities, the PU should set the PUA flag (PsR bit <0>) via a Set Mode instruction prior to halting. This eliminates the need to save and restore registers if the halted PU is assigned to process an interrupt. (Later implementations of the architecture may use the PUA flag, as well as the Hlt flag, in interrupt routing.)

Synchronize Operation. When a Wait instruction is issued with the PU Mask bit corresponding to the issuing PU set to "1", the issuing PU suspends execution until all of its target PUs have halted in the same mode (user or system) as the PU issuing the Wait instruction. When all target PUs have halted, the Wait instruction completes and execution continues on the PU executing the Wait instruction.

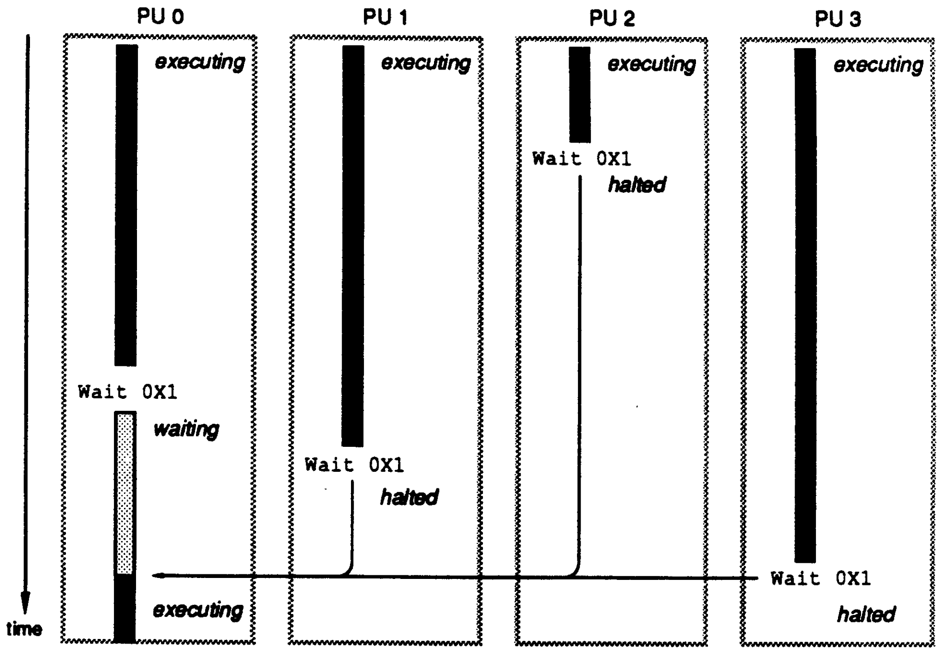


Figure 5.1. SIMD Mode Termination Via **Wait**

Combining both halt and synchronize operations in the Wait instruction lets this instruction be used to terminate SIMD execution. For example, Figure 5.1 illustrates an interval of execution which begins with all four PUs active, executing the same code. As PUs complete their current activity, they all execute the same instruction, **Wait** 0X1 (0X1 = 0001B). On PUs 1-3, this instruction performs a halt operation, since the PU Mask bits corresponding to these PUs are set to "0". On PU 0, this instruction performs a synchronization operation, since PU Mask bit 0 = "1". PU 0 waits until PUs 1-3 have halted, and then continues execution. At this point, PU 0 is assured that all other PUs have completed their activities, including storing of results if necessary, and halted. PU 0 can now perform any serial steps required by the computation, send new data to PUs 1-3 through memory or via Send instructions, and re-initiate execution on these PUs via a Start or Resume instruction.

SIMD mode execution frequently results from a loop which can be "unwound" across two or more PUs. Depending on dependencies within the loop, synchronization via Wait instruction execution may be required on every iteration or only at the end.

Address Broadcasting. A PU returns one or more halted PUs to execution via the broadcast instructions

RsM PUMask

Strt @RegB, PUMask

The Resume (**RsM**) instruction causes each target PU, if halted, to resume execution at the address in its Current PC. All the target PUs may or may not be halted at the time the Resume instruction is issued. Halted PUs resume execution immediately; other PUs resume immediately after halting.[2] Execution of the Resume instruction does not complete until all target PUs have halted and resumed execution. On resuming execution, each target PU clears the PUA and Hlt flags in its PsR. Note that if a PU sets PUA prior to halting, its state (other than PC and PsR contents) is unpredictable when its execution is resumed (or started).

When the issuing PU is in user mode, all target PUs must halt in user mode before the Resume instruction completes. When the issuing PU is in system mode, the Resume instruction will resume execution of target PUs halted in either system or user mode. This lets a system activity signal a user activity that it can execute concurrently with the system activity.

The Start (**Strt**) instruction broadcasts the address in general register RegB to each halted target PU, and causes the target PU to start execution at that address. All the target PUs may or may not be halted at the time the Start instruction is issued. Halted PUs start execution immediately; other PUs start immediately after halting.[2] Execution of the Start instruction does not complete until all target PUs have halted, received their new starting address, and started execution. On resuming execution, each target PU clears the PUA and Hlt flags in its PsR. Unlike Resume, the Start instruction requires all its targets to have halted in the same mode as the issuing PU before it completes.

A PU cannot effect a transfer of control by specifying itself as the target of a Start instruction, since the PU Mask bit corresponding to the PU issuing the Start instruction is ignored.

Data Broadcasting. A PU can broadcast a data value (or data address) contained in one of its general registers to one or more other PUs via the Send instruction:

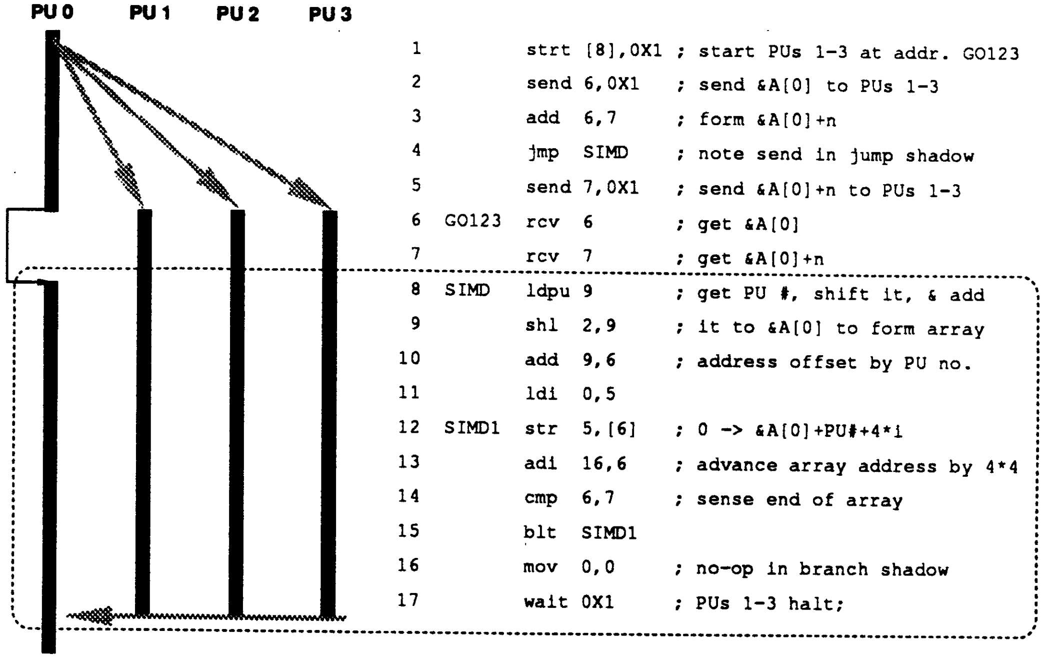
Send (RegB), PUMask

Each target of the Send instruction must execute a Receive instruction to store the broadcasted value in one of its general registers:

Rcv RegA

[2] In Antares, a synchronization operation is implicit in Resume instruction execution; all target PUs must simultaneously be halted in the appropriate mode before their execution is resumed and the Resume instruction completes. The Start and Send instructions operate similarly. Start, like Resume, requires that all target PUs must simultaneously be halted in the appropriate mode before the new starting address is broadcast, target PU execution resumed and Start instruction instruction execution completed. Send requires that all target PUs be receiving (i.e., executing unsatisfied Receive instructions) in the appropriate mode before the data value is broadcast and the Send instruction completes execution.

This synchronization is a characteristic of the Antares implementation, not the Scorpius architecture, and may not occur in all implementations. (Note that synchronization eliminates the need to keep track of which PUs have been resumed, started, or received data.) If synchronization is required prior to Resume, Start, or Send instruction execution, it should be explicitly programmed using a Wait instruction.



```
        strt [8],0X1 ; start PUs 1-3 at addr. GO123
        send 6,0X1   ; send &A[0] to PUs 1-3
        add  6,7     ; form &A[0]+n
        jmp  SIMD    ; note send in jump shadow
        send 7,0X1   ; send &A[0]+n to PUs 1-3
GO123   rcv  6       ; get &A[0]
        rcv  7       ; get &A[0]+n
SIMD    ldpu 9       ; get PU #, shift it, & add
        shl  2,9     ; it to &A[0] to form array
        add  9,6     ; address offset by PU no.
        ldi  0,5
SIMD1   str  5,[6]   ; 0 -> &A[0]+PU#+4*i
        adi  16,6    ; advance array address by 4*4
        cmp  6,7     ; sense end of array
        blt  SIMD1
        mov  0,0     ; no-op in branch shadow
        wait 0X1     ; PUs 1-3 halt;
```

Figure 5.2. SIMD Execution Using **Strt**, **Send**, **Rcv**, and **Wait**

The Send instruction sends the contents of general register RegB to each target PU. Execution of the Send instruction does not complete until each target PU has executed a Receive instruction in the same mode as the sender to store the broadcasted value in the target's general register RegA.[2] If a target initiates execution of a Receive instruction before execution of the corresponding Send, the target waits for the data value to be broadcast by the Send instruction before execution of the Receive completes. If a target initiates execution of a Receive after execution of the corresponding Send has been initiated, the broadcasted value is stored and the Resume completed without delay.

A PU cannot transfer data from register RegB to register RegA by specifying itself as the target of a Send instruction, since the PU Mask bit corresponding to the PU issuing the Send instruction is ignored.

As a simple example of the use of Start, Send, Receive, and Halt instructions, Figure 5.2 shows, in somewhat simplified form, the SIMD code which might be generated by the C statement:

```
for (i=0; i<n; i++) A[i]=0;
```

The compiler "unwinds" this loop across the four PUs in a SIMD loop in which PU i clears array elements A[i], A[i+4], A[i+8], ... It is assumed that, initially, PU 0 is executing, PUs 1-3 are halted, register 6 contains a pointer to the start of the array (&A[0]), register 7 contains the array's dimension (n), and register 8 contains the instruction address GO123. The bars at the left of the code sequence show which PUs are executing which lines of code. Instructions are shown in Pyxis assembler format. (The actual compiler-generated code is more complex because (e.g.) &a[0]+n must be checked for overflow.)

The sequence begins with PU 0 broadcasting the address GO123 to PUs 1 -3; this is the address at which these three PUs will perform their initialization for the SIMD loop. PU 0 then sends the starting address of the array and the ending address plus one to the other PUs (lines 2, 3, and 5), and jumps to the beginning of the SIMD code section. (In actual compiler-generated code, the addition of the base address and loop limit would be followed by an overflow test.) Note that PU 0's second Send instruction is executed in the shadow of the jump.

On being activated by the Start instruction, PUs 1-3 transfer control to GO123, and begin their execution by storing the array starting address and ending address plus 1. Thereafter, beginning at line 8, all four PUs execute the same instruction sequence (circled by a dashed line in Figure 5.2). Each PU loads its number, shifted left 2 places to form a word offset, and adds it to the array starting address. In the first iteration of the loop, then, PU 0 clears A[0], PU 1 clears A[1], etc. At the end of an iteration, each PU advances its array by 4*4 (PU offset times word increment), so that on the second iteration PU 0 clears A[4], PU 1 clears A[5], and so on. When each PU has advanced its array address beyond the end of the array, it executes a **Wait** 0X1 instruction. This halts PUs 1-3; PU 0 waits until all three PUs are halted and then continues executing.

5.3 Inter-PU Traps

There are two instructions which cause one or more other PUs to generate traps; Preempt (**Prmpt**) and Restart (**Res**). These instructions use the PU Mask field to specify which PUs are to generate traps, and so resemble broadcast instructions. The two instructions are similar in form:

Prmpt	PUMask
Res	PUMask

Both instructions are privileged and must be executed in system mode. The PU Mask field bit corresponding to the PU issuing the Preempt or Restart instruction is ignored. Issuing a Preempt or Resume instruction with no target PUs specified (PU Mask = 1111B) produces unpredictable results.

Preempt. The Preempt instruction causes each target PU to generate a PU Preempt trap. If a target PU is interrupt/trap enabled (PsR bit <15> = "1"), it immediately recognizes the trap; the PU Preempt Trap flag is set in the Trap Register, PsR and PC contents are saved in the SaveR and PCQ registers, the PU mode is set to system, the interrupt/trap enabled flag is cleared, and control is transferred to the appropriate kernel entry address (see Chapter 4). If a target PU is interrupt/trap disabled, recognition of this trap is deferred until the PU becomes interrupt/trap enabled. (This is the only trap which can be deferred.) Execution of the Preempt instruction does not complete until each target PU has recognized the trap.[3]

[3] Antares waits until all target PUs are simultaneously interrupt/trap enabled before trap generation and recognition is effected.

While it is possible to cause multiple PUs to to generate traps via a single Preempt instruction, the PU issuing the instruction will incur a delay (which may or may not matter) if any of the target PUs is interrupt/trap disabled. To minimize this delay, the operating system may elect to preempt one PU at a time, using the information in the Global Status Register (Figure 5.3) to determine if a prospective target is enabled or disabled. (While the target's state may change between the time it was examined and the time at which the Preempt instruction is issued, so that a delay occurs anyway, the average delay can be reduced in this way.)

The operating system uses the Preempt instruction primarily to interrupt execution of other PUs when initiating an address space switch; a message area in memory is used to advise preempted PUs of the reason for preemption. The operating system also may preempt PUs in preparation for parallel execution of some system task. (However, the trap overhead must be considered when specifying what tasks should be executed in parallel). It is expected that the operating system will perform preemption in a critical section, so that one and only one PU at any instant will be attempting to preempt other PUs. If PU A and PU B both attempt to preempt PU C, either PU A or PU B, depending on implementation, will succeed first, and the other will succeed when PU C becomes interrupt/trap enabled once again. However, if PU A and PU B simultaneously attempt to preempt one another, the result is unpredictable.

Restart. The Restart instruction causes each target PU to generate and immediately recognize a PU restart trap, regardless of whether or not the target PU is interrupt/trap disabled. The PU Restart Trap flag is set in the Trap Register, the PU mode is set to "system", the interrupt/trap enabled flag is cleared, and control is transferred to the appropriate kernel entry address. If the PU is interrupt/trap enabled when the PU Restart trap is recognized, the contents of the PsR and PC are saved in the SaveR and PCQ registers. If the PU is interrupt/trap disabled when the PU Restart trap is recognized, the contents of the PsR and PC are not saved, and the contents of the SaveR and PCQ registers upon transfer to the kernel interrupt/trap entry address are unpredictable.

The Restart instruction is used to restart execution after a fatal error has been detected. A fatal error may be detected by hardware (and reported via a non-maskable interrupt or by the PU Check Trap) or by software (as when the kernel decides that a program is in an infinite loop). In such cases, execution in the currently-active address space is terminated; it usually is desirable to save as much state as possible for subsequent error analysis. A hardware-detected error generates either a non-maskable interrupt (Machine Check, Power/Temp. or Deadlock), which always is presented to and recognized by PU 0, or a PU Check Trap, which is presented to the PU causing the PU Check. If PU 0 was interrupt/trap disabled at the time a non-maskable interrupt is recognized and its PCQ Enable flag is cleared, its PsR and PC contents are lost. A PU Check Trap is generated when a trap (other than a PU Preempt Trap) is generated on a PU which is interrupt/trap disabled. The PU's PsR and PC contents are lost if its PCQ Enable flag is cleared when the trap occurs.

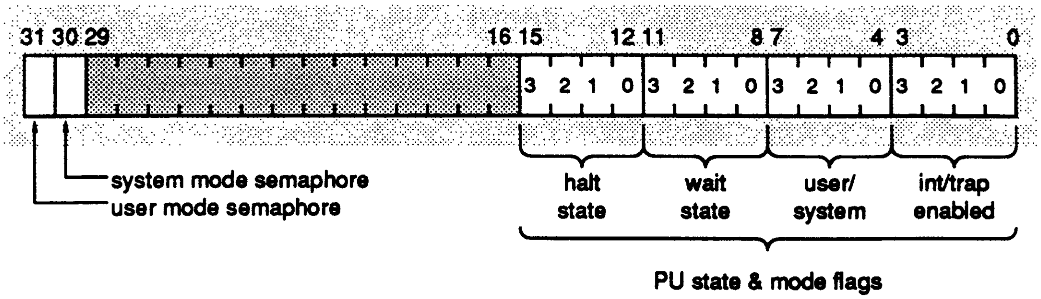


Figure 5.3. Global Status Register (**GSR**)

When an interrupt or a trap indicating a fatal error is recognized, the PU on which it is recognized begins the process of terminating execution in the current address space. If other PUs are interrupt/trap enabled when the error occurs, this PU can interrupt their execution using the Preempt instruction. In this case, these PUs can save their PsR and PC contents to aid in error analysis. If any PU is interrupt/trap disabled, it can be interrupted via the Restart instruction; in this case, however, the current PU's PsR and PC contents can be saved only if the PU has its PCQ Enable flag set. See ***The PCQ Enable Flag*** in Section 4.6.

5.4 Semaphore Instructions

Lock and Unlock. In addition to using broadcast instructions, PUs can coordinate their activities by means of semaphore instructions. The Global Status Register (Figure 5.3) contains two semaphores, a user mode semaphore (bit <31>) and a system mode semaphore (bit <30>). Semaphore operations are performed by Lock and Unlock instructions, whose form is simply

Lock

Unlk

If the PU issuing a Lock or Unlock instruction is in user mode (PsR bit <13> = "1"), these instructions operate on the user mode semaphore. GSR bit <31>. If the PU is in system mode, these instructions operate on the system mode semaphore, GSR bit <30>.

The Lock instruction examines the appropriate (user mode or system mode) semaphore; if the semaphore is set (semaphore bit = "1"), it is cleared (*locked*) and Lock instruction execution completes. If the semaphore initially is cleared, Lock instruction execution is blocked until the semaphore becomes set; the semaphore then is cleared and Lock instruction execution completes.

The Unlock instruction unconditionally sets the appropriate (user mode or system mode) semaphore. The PU unlocking a semaphore may or may not be the PU which originally locked it.

Service Order. Multiple PUs may attempt to lock an already-locked semaphore, in which case execution on these PUs are blocked until the semaphore is unlocked. On execution of an Unlock instruction, one of the blocked PUs is selected and its Lock instruction allowed to complete, locking the semaphore again; the remaining PU wait for a subsequent unlocking. The order in which waiting PUs are granted a semaphore upon its unlocking is implementation-dependent. For correct operation, code should not rely on the order in which semaphore lock requests are serviced.

Antares uses a grouping scheme to provide fair service to semaphore requestors. If a semaphore is initially unlocked with no waiting requests, and then is locked, Antares records the number of each PU which, during that locked interval, attempts to lock the semaphore. These PUs comprise a group, or batch. When the semaphore is unlocked, it is granted next to the lowest-numbered PU in this group; no other PU is allowed to join the group until all PUs in the group have been granted the semaphore. Thus, all requests arriving during the first locked interval are processed before any arriving during later lock intervals. For example, suppose a semaphore is unlocked with no waiting requests when PU 0 executes a Lock instruction. Before PU 0 unlocks the semaphore, PU 3 and later PU 2 generate requests (i.e., initiate Lock instructions). When PU 0 unlocks the semaphore, PUs 2 and 3 are marked as comprising the group to receive service, and the semaphore is granted to PU 2. Suppose PU 1 and later PU 0 generate requests before PU 2 unlocks the semaphore; their requests are ignored until PU 2 has unlocked the semaphore and PU 3 locked it. At that point, all the PUs in the original group have received service, and PUs 1 and 0 now can join the group.

Applications. Semaphore instructions are used to serialize execution of critical regions and control access to data structures. For example, when a PU begins saving state following recognition of an interrupt or a trap, it saves the contents of one general register in the global Scratch Register (the **ScR**, special register 10) and uses that general register to establish the memory address at which general registers are to be stored. Since several PUs simultaneously can be attempting to save state and there is only one Scratch Register, general register saving is done in a critical section such as that outlined below.

```
lock               ;  begin critical section
movts 0,10         ;  (R0) -> ScR
pfxi  ...
ldi   ...,0        ;  save area address -> R0
stm   15,[0]       ;  store registers 1-15
movfs 10,1         ;  retrieve original (R0)
unlk               ;  free ScR
.......  ...       ;  adjust address & store (R0)
```

The operating system will require more than the single lock per mode provided by the architecture. Operating system locks can be implemented in various ways, using Lock and Unlock instructions to control software lock access. Spin locks (which require a waiting PU to repeatedly read a memory location until the contents of the location change state) cause cache access contention and should be avoided.

If necessary, additional semaphores can be added without great difficulty by adding semaphore bits in the GSR and defining instructions to operate on these bits. For example, if another pair (user mode and system mode) of semaphores is added, the current Lock instruction would become **Lock1** and a second Lock instruction, **Lock2**, would be added to the instruction set. Unlock instructions would be handled in the same way.

5.5 PU States & Deadlock Detection

PU States. The Global Status Register (Figure 5.3) contains, in addition to semaphores, four flags for each PU: a Halt flag, a Wait flag, a Mode flag, and an Enabled flag. The Halt flag is a copy of the PU's Halt flag, PsR bit <24>, the Mode flag is a copy of the PU's User/System Mode flag, PsR bit <13>, and the Enabled flag is a copy of the PU's Interrupt/Trap Enabled flag, PsR bit <15>. The Wait flag is set to "1" whenever a PU begins waiting as the result of issuing a broadcast or semaphore instruction. GSR bit positions of these flags for the four PUs are as follows.

	PU 0	PU 1	PU 2	PU 3
Halt flag	12	13	14	15
Wait flag	8	9	10	11
Mode flag	4	5	6	7
Enabled flag	0	1	2	3

From an execution viewpoint, a PU has three states: run, wait, and halt. These states are reflected by the Halt and Wait flags in the GSR, as follows.

Halt	Wait	PU state
"0"	"0"	run
"0"	"1"	wait
"1"	"0"	halt
"1"	"1"	invalid

These states are described below.

run state. A PU is in run state while it is executing instructions, even though instruction execution may be temporarily delayed (stalled) because of resource conflicts or cache line misses. (An event counter can be used to count the number of cycles a PU is in run state or, equivalently, is active in either system mode or user mode, or in both modes; see Chapter 7.)

halt state. A PU is in halt state when the Halt flag in its PsR is set, either because the PU executed a Wait instruction with its own number specified in the PU Mask field or because the kernel set the Halt flag in the PU's SaveR before executing an **RtI** pair. A PU exits halt state when it is the target of a Start or a Resume instruction or when an interrupt or an inter-PU trap is recognized.[4]

[4] Antares requires that all targets of a Start or Resume instruction be halted simultaneously before the start or resume operation is performed. In Antares, then, a target PU effectively remains in halt state until the Start or Resume completes execution.

wait state. A PU is in wait state when execution of an instruction is suspended waiting for an activating signal from some other PU or waiting for a semaphore to become unlocked. The activating signal may represent (a) an information transfer or (b) a change in the state of other PUs. An information transfer either is an instruction address value sent by a Start instruction (or, implicitly, by a Resume instruction) or a data value sent by a Send instruction. A PU enters wait state for one of the following reasons.

- *halt wait*. It initiates execution of a Start or a Resume instruction and all target PUs are not in halt state. It returns to run state when all target PUs have halted and then had the start address broadcast to them or received the resume signal.

- *receive wait*. It initiates execution of a Send instruction and all target PUs are not in wait state attempting to receive data. It returns to run state when the data value has been received by each target PU.

- *send wait*. It initiates execution of a Receive instruction and some other PU is not already attempting to send data to this PU. It returns to run state upon receipt of the data value.[5]

- *join wait*. It initiates execution of of a wait instruction and all target PUs are not in halt state. It returns to run state when all target PUs are, jointly, in halt state.

- *semaphore wait*. It initiates execution of a Lock instruction and the semaphore flag already is locked. It returns to run state when execution of a Unlock instruction by some other PU releases the semaphore and the semaphore is then granted to the waiting PU.

- *preempt wait*. It initiates execution of a Preempt instruction and all target PUs are not interrupt/trap enabled. It returns to run state when all target PUs have been recognized the PU Preempt trap.[6]

State Change Delay. The PU state flags in the GSR are copies of local flags, and may be time-delayed in reflecting state changes. This delay, which represents the time required to propagate state changes from a PU to the GSR is implementation-dependent, and typically will be on the order of 1 or 2 cycles.

Deadlock Detection. A deadlock occurs when all four PUs are in halt or wait state (i.e., when at least one PU is not in run state) for some period of time. This period depends on the state change delay described above. The GSR is monitored by hardware which, upon detecting deadlock, generates a Deadlock

[5] Antares requires that all targets of a Send be simultaneously receiving before the data value is broadcast.

[6] Antares requires that all targets of a Preempt be simultaneously interrupt/trap enabled before the preempt operation takes place.

interrupt. This interrupt is non-maskable and is presented to and immediately recognized by PU 0.

To isolate the program error which caused the deadlock, it is useful to determine the address of the current instruction for each of the four PUs. This is easily done for PUs which were interrupt/trap enabled when the Deadlock interrupt was recognized. An interrupt/trap enabled PU can be activated via a Preempt instruction, and that PU's PCQ[1] register provides the address of the instruction which the PU was attempting to execute when the deadlock occurred. If current instruction addresses can be obtained for all four PUs, it usually is straightforward to determine why the deadlock occurred. If PU 0 is interrupt/trap disabled when the Deadlock interrupt is recognized, the PsR and PC contents are not saved, making analysis more difficult. Similarly, if any of the other PUs are interrupt/trap disabled, PU 0 will have to activate those PUs via a Restart instruction, and their PsR and PC contents will be lost. Before dealing with the other PUs, PU 0 should save the contents of the GSR, so that it can be at least determined whether a PU was in halt state or wait state.

6. Cache Control Operations

6.1 Introduction

The Scorpius instruction and data caches are architecturally visible at both system and user levels. Instructions are provided to control certain aspects of cache operation in order to insure correct operation and also to improve performance. This chapter provides a brief introduction to cache organization, using the Antares cache as an example, and then describes the various cache control instructions.

While caches are architecturally visible, substantial differences among implementations are possible; only the line size of 64 bytes must be maintained. It is possible that an implementation might have a single composite cache holding both instructions and data (although unlikely, for performance reasons). The semantics of certain instructions may differ in different implementations, particularly for those instructions used to flush part or all of a cache on an address space switch or termination; the operating system may require model-dependent code to perform certain of its functions. However, at the user level, code which operates correctly on a Scorpius CPU with separate instruction and data caches is guaranteed to operate correctly on all implementations of the Scorpius architecture.

6.2 Cache Organization

A cache is a buffer placed in the data path between a processor and memory. It is constructed of faster (and more expensive) components than memory, and its capacity is small compared to that of memory. When a processor reads a word not in the cache (a cache miss), a block of words, called a *line*, is transferred from memory to the cache, and the requested word transferred from the cache to the processor. While a line may be as small as one word, it almost always is larger for two reasons. First, a larger line size often permits the overhead (memory access time, etc.) involved in the memory-

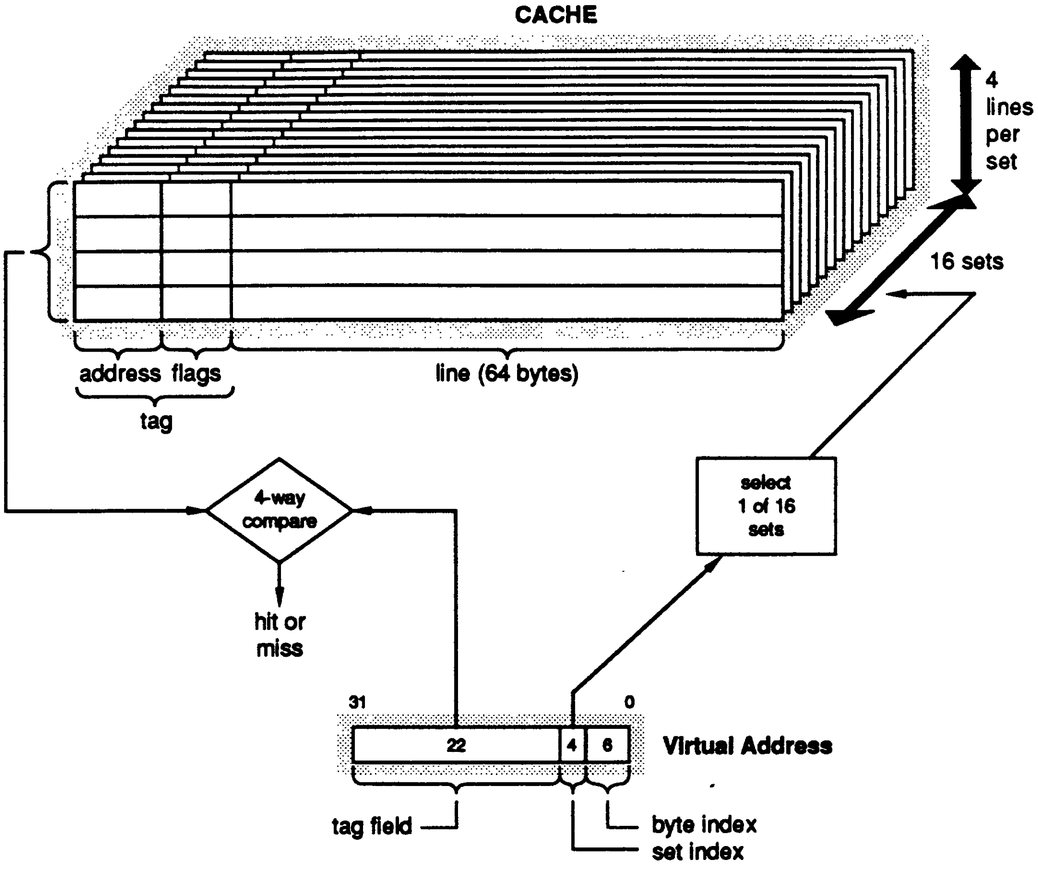


Figure 6.1. Cache Address Mapping in Antares

cache transfer to be amortized over a number of words. Second, programs usually evidence *spatial locality;* words near the word currently being read are more likely to be read in the immediate future than are words in more distant locations. Most programs also evidence *temporal locality;* words read by the most recently executed instructions are more likely to be read again in the future than are words not read by those instructions. Since the time required to access a word in the cache is much less than the time required to access a word in memory, a combination of spatial and temporal locality can substantially improve performance.[1] Generally, the spatial and temporal localities of code are greater than those of data. Sometimes data can have considerable spatial locality but little temporal locality, as in certain vector operations. When neither spatial locality nor temporal locality exists, the cache can degrade performance; Scorpius permits the cache to be bypassed by declaring a page or pages to be non-cacheable (see Chapter 3).

[1] The programmer, through careful design of program and data structures, can enhance both temporal and spatial locality.

As long as the words of a cache line are only read by the processor, the contents of that line's location in the cache are identical to the contents of that line in memory. However, as soon as a processor writes to one of the words in that line, the cache and memory copies differ. This is the *coherence* problem, which arises whenever multiple copies of data can exist in a way such that changing the value of one copy does not change the value of all copies. Some system designs attempt to ease the cache coherence problem by using a *store-through*, or *write-through*, cache; whenever the processor writes a word (i.e., executes a store instruction), the word is written to the cache, if the line in which the word is located is in the cache, and also to memory. Other system designs, including Antares, use a *store-to*, *copy-back*, or *write-back*, cache; processor writes are directed only to the cache, and a line containing modified words is, in most case, written to memory only when its cache space is need for other data. Store-through designs can generate substantially more memory traffic than store-to designs, with consequent impact on performance. Since memory is not kept current with the cache, store-to designs require that coherence be maintained by other means. The Scorpius architecture assumes that coherence, when necessary, is maintained by software, and provides supporting instructions.

From the viewpoint of the cache, memory is divided into lines of 16 words which start on line boundaries; i.e., the low-order six bits of the address of the first word of a line are zero. The Antares instruction and data caches are identical (except that a PU cannot store into the instruction cache); each has a capacity of 64 lines (4K bytes). The cache (instruction or data) is organized into 16 sets of 4 line locations (Figure 6.1). Any memory line maps into one and only one set, but can be stored in any one of the 4 line locations in that set. Associated with each cache line location is a *tag* comprising address and flag fields. The address field, or address tag, contains bits <31:10> of the virtual address of the memory line stored in the cache line location; the flag field contains a valid bit, a modified (or dirty) bit, a system/user bit, a read-only bit, and least-recently-used (LRU) bits. The valid bit can be cleared by certain cache control instructions (and also is cleared on system powerup). The modified bit is not used by the instruction cache; it is set by the data cache when a store instruction modifies a word or a byte of the line. The system/user and read-only bits are inherited from the page table entry of the page in which the line is located. The LRU bits record the relative recency of reference of each of the four lines in a set, and are used to select which of the four is to be replaced when a new memory line must be read into that cache line set. The Antares cache is called a *four-way set associative* cache because an address is associated with each line of a set so that any memory line mapping to a set can be stored in any of the set's four line locations.

Note that no address space number is stored with a cache line in Antares. Consequently, it is necessary to flush the cache on an address space switch or termination. (For the relatively small caches of Antares, this has little performance impact, since all cache lines usually would be replaced anyway.)

Figure 6.1 shows how a cache line location is selected on a cache instruction or data access. The low-order six bits of the instruction or data address are used to select the desired byte, halfword, or word from the line, once the correct line has been found. Bits <9:6> of the virtual address are used to select one of the sixteen sets of lines. Once a set has been selected, the address tags of each of the four line locations in that set are compared, simultaneously, with bits <31:10> of the virtual address. If the address tag associated with one of the four line locations matches the tag field bits of the instruction or data address, a *cache hit* has occurred; the instruction or data unit is read from or written to the cache, and the LRU bits are updated. If none of the address tags match the tag bits of the address, a *cache miss* occurs.

In processing a miss, the cache control hardware selects one of the lines from the set selected by the instruction or data address for replacement. In Antares, a line marked invalid is selected if one exists; otherwise, the least-recently-used line (referenced by a load or store instruction) is selected, using the LRU bits in the flag field associated with the line's location. If the line selected for replacement is not modified, the missing line can be read directly into that line location. If that line is modified, it must be written before it can be replaced. In Antares, when a modified line must be replaced, it is written to a Write Buffer, the missing line is *moved in* from memory to the vacated cache line location, and the modified line then is written — *moved out* — to memory. When a line has been moved in, the address tag and flags of the cache line location are updated.

Cache designs for other Scorpius implementations may differ from the Antares cache design in every aspect except line size. Future implementations can be expected to have larger instruction and data caches which are not necessarily equal in size. A larger virtual-addressed cache may incorporate address space numbers in its tag to eliminate the need to flush the cache on an address space switch. The number of lines per set (sometimes called the *degree of associativity*) may change, and line replacement algorithms other than LRU may be used.[2] It also is possible that future implementations may incorporate a real-addressed cache rather than a virtual-addressed cache, which eliminates the *synonym* problem.[3] This problem arises when one real page is mapped to more than one virtual page. When this occurs, a virtual-addressed cache can have multiple copies of the same line; modifying one copy does not modify the others. In Scorpius, the resolution of the synonym problem is the responsibility of software (see Section 3.8).

[2] A cache whose degree of associativity is 1 (any given memory line maps into one and only one cache line location) is called a *direct-mapped* cache.

[3] However, a real-addressed cache requires address translation to be done prior to (or at least parallel with) cache address tag comparison. In Antares, all four PUs can simultaneously access the data cache (and instruction cache as well); it is easier to support simultaneous address tag comparisons for four accesses than it is to support four simultaneous address translations.

6.3 Cache Line Control

Cache line control instructions permit both system and user to force data cache lines to be written to memory, mark data cache lines unmodified or invalid, mark instruction cache lines invalid, and create data cache lines. All cache line control instructions specify their operand —a memory line — via a byte address in a general register; with the exception of the Invalidate Instruction Cache line instruction, this address can be prefixed. If the Prefix Valid flag is "0", the address in the specified general register are used as the operand address. If the Prefix Valid flag is "1", the contents of the Prefix Register are shifted left two places and added to the address in the general register to form the operand address. (The prefix is a signed word displacement from the register address.) Bits <31:6> of the operand address identify the memory line; bits <5:0> are ignored.

The Create Data Cache line instruction **CDC** creates a memory line in the data cache without causing the line to be moved in from memory if missing. The data cache set into which the memory line maps is examined; if the specified line already is in that set, **CDC** completes execution. If the line is not found in that set, the line location of the least recently used line in the set is selected, and the line presently in that location is written to memory if it is modified. The address tag at that line location is set to bits <31:10> of the address of the line to be created, the system/user flag bit is set to the value of the system/user flag bit in the page table entry for the page in which the line resides, and the remaining flag bits are set so as to mark the line valid and unmodified.

In executing a **CDC** instruction, the TB/TLB is searched to determine if a translation is present for the page in which the line to be created is located. If a translation is not present, a TB/TLB miss occurs and the page table entry for that page is located and read. (Section 3.7 describes TB miss processing.) If either the page table entry or its associated directory entry is invalid, a data page fault trap is generated, and the page address is stored in the Trap Register. **CDC** is the only cache control instruction which can cause a page fault.

The **CDC** instruction is used to eliminate the overhead of a movein when a memory line is to be completely rewritten. Because a cache line is created without clearing the line, the contents of the newly-created line are those of the line it replaced. This creates a security exposure; programs which deal with secure data should, before terminating, insure the cache no longer contains that data. One possibility is for the operating system to provide an model-dependent "erase" service function. This function, which would be aware of the degree of associativity, would simply read empty lines into the appropriate set. The simplest solution is to place secure data in a non-cacheable page.

Four instructions are provided to dispose of data cache lines: Flush, Invalidate, Update, and Validate Data Cache line. These differ in whether or not they cause the specified line to be written to memory if modified and in whether the final state of the line is invalid or valid and unmodified. The functions of these instructions are summarized in the following table.

instruction	mnemonic	write line to memory if modified?	final state of cache line
Flush Data Cache line	**FDC**	Y	invalid
Invalidate Data Cache line	**IDC**	N	invalid
Update Data Cache line	**UDC**	Y	unmodified
Validate Data Cache line	**VDC**	N	unmodified

No operation is performed by these instructions if the specified line is not in the cache.

FDC (or **UDC**, depending on the final line state desired) can be used to insure that lines of an output buffer are written to memory before an IO operation is performed. **VDC** can be used to save unnecessary moveouts by marking "scratch" lines unmodified. In Antares, **CDC** and **FDC** are used together to flush the data cache's Write Buffer; see Section 3.6.

The Invalidate Instruction Cache line instruction **IIC** marks the specified instruction cache line invalid; it performs no operation if the specified line is not in the cache. **IIC** and **FDC** can be used to control instruction cache/data cache synonyms. One case in which such synonyms occur is in interpretive execution. In this case, the interpreter builds a line of code in the data cache, executes a **FDC** instruction to write the code line from the data cache to memory, executes an **IIC** instruction to insure that any previous copy of the line which might be present in the instruction cache is invalidated, and then transfers control to the code line, causing the newly updated memory line to be moved into the instruction cache (see Section 3.8). In Antares, when an instruction cache line is invalidated, the instruction queues of the pipelines of one or more PUs may contain instructions from that line, depending on the line's location and recency of use. In this situation, it is necessary to flush the instruction queue of any suspect PU. A PU's instruction queue is flushed when the PU executes a Jump instruction, is the target of a Start (but not a Resume) instruction, or is dispatched on return from interrupt.

In deallocating a page, the **IIC** and **FDC** instructions can be used to invalidate the lines of that page in the instruction and data caches.[4] Since a 8KB page comprises 128 lines, this requires 128 executions of both the **IIC** and **FDC** instructions if the page contains both instructions and data. The code required can be parallelized. In Antares, it may be faster to invalidate the entire instruction cache using the Invalidate Instruction Cache (**IICA**) instruction described in Section 6.5. The choice depends on the expected reuse of instruction cache lines (i.e., the number of lines which will have to be moved back in again if the entire instruction cache is invalidated).

[4] Subsequent versions of the architecture may provide flush/invalidate page instructions.

6.4 Prefetching

When a missing line is accessed, execution of the accessing instruction is delayed until the line has been moved in. When it is known that a memory line will be used in the near future, and that line is not likely to be already in the cache, some or all of the cache miss delay can be avoided by prefetching that line.[5]

Scorpius provides a Prefetch Data Cache line (**PDC**) instruction which can be used to prefetch lines into the data cache in advance of their use. **PDC** specifies its operand— a memory line — via a byte address in a general register. This address can be prefixed. If the Prefix Valid flag is "0", the address in the specified general register is used as the operand address. If the Prefix Valid flag is "1", the contents of the Prefix Register are shifted left two places and added to the address in the general register to form the operand address. (The prefix provides a signed word displacement from the register address.) Bits <31:6> of the operand address specify the memory line, and bits <5:0> are ignored

If the memory line specified by the operand address is not already in the data cache, **PDC** causes a memory request to be initiated for that line and completes execution without waiting for the line to be moved in.[6] A cache line is selected for replacement in the same way as a normal (*demand*) miss. If the specified line is in the cache, no operation is performed.

6.5 Cache Invalidation in Antares

All Scorpius implementations require some means of invalidating part or all of a cache's contents when certain events occur. These events include deallocation of a page from an address space, address space termination, and, for caches like those of Antares whose contents are not distinguished by address space number, address space switching. Different implementations may provide different instructions or different instruction operation, requiring model-dependent code on the part of the operating system. This section describes Antares cache invalidation instructions and operation. These events also require invalidation of the Translation Buffer, as discussed in Section 3.6.

[5] The time required to process a miss depends on the implementation, and includes delays resulting from cache and memory bus conflicts as well as memory access and transfer times. In Antares, the *nominal* movein time — the time required to process a miss in the absence of conflict delays — is 15 cycles. Antares begins the movein with the word accessed on the miss, and that word is forwarded to the accessing PU before the entire line is moved in. This reduces the delay for the accessed word to 8 cycles (ignoring conflicts); however, a subsequent access to another word of the line must wait for the movein to complete. Insofar as scheduling of prefetch instructions on Antares is concerned, it is desirable to initiate prefetch of a line at least 15 cycles before the line is used; however, any lead time in prefetching a line can help performance in many instances.

[6] In Antares, if a cache miss occurs on an access of a subsequently-issued instruction and the memory transfer for the prefetch request has not yet been initiated, the prefetch request is discarded.

Instruction Cache. All lines in the Antares instruction cache belong to the user space portion of the currently-active address space (the address space whose number is contained in the ASN field of the Id Register) or to the kernel, which is mapped into every address space. The Invalidate Instruction Cache (**IICA**) instruction marks all instruction cache lines, both user space and kernel space, invalid. When the operating system deallocates a user space page, it must invalidate all lines of that page which are in the instruction cache. There are two alternatives: an **IICA** instruction can be used to invalidate all the lines in the instruction cache, or 128 **IIC** instructions can be used to invalidate individual lines of the page, as discussed in Section 6.3.

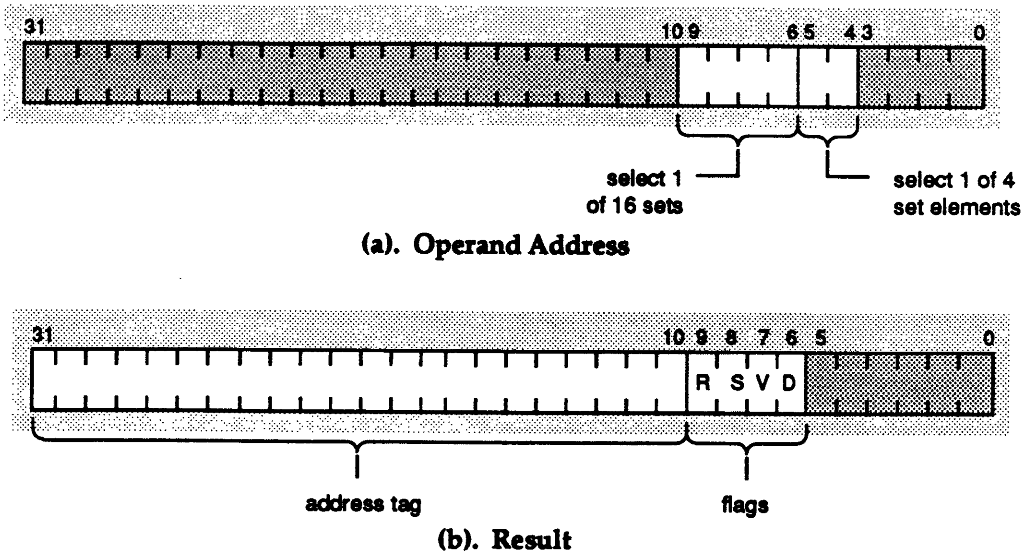
On an address space switch or termination, the **IICA** instruction is used to invalidate the instruction cache Also, the instruction queues of the pipelines of all the PUs may contain instructions fetched from the old address space, and these instructions must be flushed from the queues. The following procedure can be used.

1. Assume PU 0 is controlling the address switch or termination and PUs 1-3 are halted. To flush the instruction queues of PUs 1-3, their execution must be reinitiated after the new address space has been established either by a **Start** instruction or by being dispatched to a new address (i.e., by setting a new address in the PCQ prior to returning from interrupt).
2. PU 0 then executes an **IICA** instruction to invalidate the instruction cache, followed immediately by a **Jmp** instruction to flush its instruction queue.

While the **IICA** instruction is non-privileged in Antares, it may become privileged in later implementations. Also, the cache invalidation process is implementation-dependent. Consequently, instruction cache invalidation should be done only by the operating system, and the invalidation function should be isolated for ease in updating to new models.

Data Cache. Data cache invalidation differs somewhat from instruction cache invalidation because data cache lines can be modified, and modified lines must be written to memory before their cache line location can be marked invalid. It is not possible to invalidate the data cache with a single instruction, as can be done with the instruction cache; however, it is possible to examine the tag associated with each data cache line using the Read Data Tag by Index instruction.

From the view of the Read Data Tag by Index (**RDTX**) instruction, the data cache is an array of 64 lines indexed 0-63. The **RDTX** instruction specifies a line index as its operand address, and receives as its result the tag of the corresponding cache line location. The line index is contained in bits <9:4> of a general register; as shown in Figure 6.2(a), bits <9:6> specify one of the sixteen sets, while bits <5:4> select one of the four lines in the selected set. The result is returned in a second general register, as shown in Figure 6.2(b). The result register contains the address tag (bits 31:10> of the virtual address of the line



(a). Operand Address

(b). Result

Figure 6.2. **RTDX** Instruction Operand and Result

stored in the specified location) in bits <31:10> together with the read-only, system/user, valid, and modified flag bits from the tag in bits <9:6>. (The LRU bits are not returned.)

The flags are interpreted as follows.

R read-only flag (result register bit <9>). This flag is inherited from the page table entry for the page in which the cache line is located; it is "1" if the page is read-only and "0" otherwise.

S system/user flag (result register bit <8>). This flag also is inherited from the page table entry for the page in which the cache line is located; it is "1" if the page can be accessed only system mode and "0" if the page can be accessed in both system and user modes.

V valid flag (result register bit <7>). This flag is "1" if the contents of the cache line location are valid and "0" otherwise.

D dirty flag (result register bit <6>). This flag is "1" if the line in this location has been modified since being moved in or since being marked unmodified by a **UDC** or **VDC** instruction.

The settings of the R, S, and D flags are valid only if V = "1", so this flag should be examined first. If the line in the specified location is valid, the address of that line is formed by concatenating the address tag from the result register with the set number from the operand address register and setting the low-order six bits of the address to "0". (The last is not necessary if the address is to be used as the operand address of a cache line control instruction, since these instruction ignore the low-order six bits of an address)

Invalidation of the data cache lines of a single page can be effected by executing 128 **FDC** instructions, as discussed earlier. It also can be done by reading the tags of the 64 lines in the data cache using **RDTX**, verifying that the corresponding line is valid, determining if the address tag falls with the page, and, if it does, forming the line address from the address tag and set number and executing an **FDC** instruction to dispose of the line. The first method is more efficient and is independent of cache size. However, performance in any case is determined primarily by the number of modified lines which have to be written to memory.

To invalidate the data cache, the tag of each cache line location is read via an **RDTX** instruction and checked for validity. If the line in a location is valid, an **FDC** instruction is executed to write the line to memory (if necessary) and mark the location invalid. In performing this process, the kernel can check the virtual address and avoid invalidating its own lines. In Antares, it is not necessary to check the validity of a cache line location's contents prior to executing the **FDC** instruction; the **FDC** instruction should not carry out any operation when its operand address maps to and generates a tag match with a cache line location which is marked invalid. However, this check is a safety measure and is required if the virtual address from the tag is to be checked against kernel region addresses.

Future implementations of the Scorpius architecture may have virtually-addressed caches in which ASNs are incorporated in cache line tags. With this extension, a cache access results in a "hit" only if the virtual address associated with the access matches the virtual address field of a cache line tag and the ASN field from that tag matches the ASN field of the **IdR** (i.e., the line "belongs" to the currently-active address space). In such implementations, it is not necessary to flush the cache on a task switch, because all cache lines are uniquely identified. However, it is still necessary to be able to invalidate cache lines on address space termination so that the address space number can be safely reused, so **RDTX** (or a similar, possibly background, mechanism) still is required. It now is necessary to distinguish between lines with the same virtual address but different ASNs. **RDTX** instruction compatibility between Antares and future implementations with larger caches which incorporate ASNs in cache line tags can be achieved if the **RDTX** definition is extended so that the valid flag, as returned by **RDTX**, is "1" only if (a) the valid bit in the cache line tag is "1" and (b) the ASN field in the tag matches the ASN field in the **IdR**. The kernel's invalidation code then can use the valid flag to bypass invalidation of lines not belonging to the currently-active address space. However, since **FDC** should operate only on valid lines belonging to the currently active address space, it also should be possible to simply read the tag for each line location, form the address of the line in that location (without regard to validity), and flush and invalidate the line via an **FDC** instruction.

7. Measurement Facilities

7.1 Introduction

Scorpius provides two event counters which can be used to count cache moveins and moveouts, Translation Buffer (TB) misses, total cycles, active cycles for a PU, and instructions executed by a PU. These counts can be collected in user mode, system mode, or in both modes. Using these counts, the programmer can, for example, measure the level of parallelism realized by an application, or the number of instructions and cache moveins and moveouts required to accomplish some task. These measurements provide a means of evaluting the effect on performance of application program and operating system changes. It is expected that the operating system will provide system services for selecting events to be counted, initializing counters, and reporting counts.

This chapter describes the event counters and their controls, and briefly discusses the collection and interpretation of measurement data.

7.2 Event Counters and Their Controls

Event Counters 1 and 2 are special registers 8 and 9. These are 32-bit global, privileged, registers whose contents can be read or written via Move From Special and Move To Special instructions. By setting flags in the EvCtlr1 and EvCtlr2 fields of the Interrupt Control Register (ICR), these counters can be used to count various events.

Counts are 32-bit unsigned quantities; an event counter can count approximately 4.3 billion events before overflowing. When an event counter overflows, an Event Counter Overflow interrupt is generated if the Event Counter Overflow Interrupt Enable flag for that counter in the ICR is set to "1". If the Overflow Interrupt Enable for the counter is "0", the overflow simply is ignored. In either case, the counter "wraps around" and continues to count. Upon recognition of the interrupt, a pending interrupt flag indicates that an

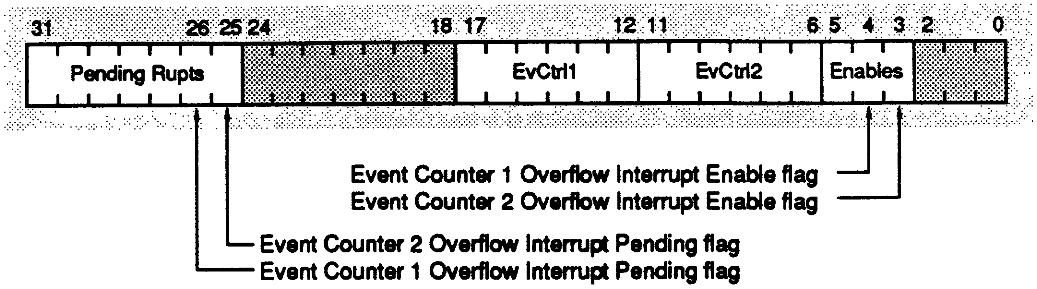
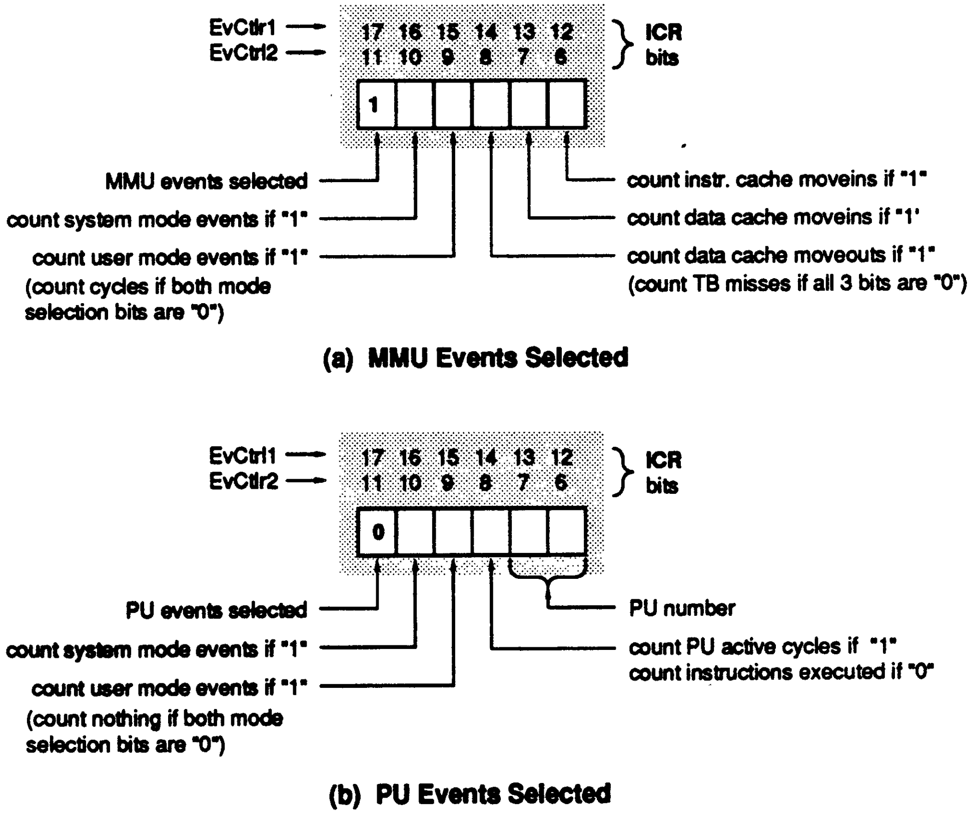

31
26 25 24
18 17
12 11
6 5 4 3 2
0
Pending Rupts
EvCtrl1
EvCtrl2
Enables
Event Counter 1 Overflow Interrupt Enable flag
Event Counter 2 Overflow Interrupt Enable flag
Event Counter 2 Overflow Interrupt Pending flag
Event Counter 1 Overflow Interrupt Pending flag

Figure 7.1. Event Counter Control Fields & Interrupt Enable/Pending Flags in the ICR.


EvCtlr1
EvCtrl2
17 16 15 14 13 12
11 10 9 8 7 6
ICR bits
1
MMU events selected
count system mode events if "1"
count user mode events if "1"
(count cycles if both mode selection bits are "0")
count instr. cache moveins if "1"
count data cache moveins if "1'
count data cache moveouts if "1"
(count TB misses if all 3 bits are "0")
EvCtrl1
EvCtlr2
17 16 15 14 13 12
11 10 9 8 7 6
ICR bits
0
PU events selected
count system mode events if "1"
count user mode events if "1"
(count nothing if both mode selection bits are "0")
PU number
count PU active cycles if "1"
count instructions executed if "0"

(a) MMU Events Selected

(b) PU Events Selected

Figure 7.2. Event Counter Controls

event counter overflow has occurred and identifies the overflowing counter. The Event Counter Overflow Interrupt Enable flags for counters 1 and 2 are ICR bits <4> and <3>, respectively. The Event Counter Overflow Interrupt Pending flags are ICR bits <26> and <25>, respectively. These flags, together with EvCtlr field positions, are shown in Figure 7.1. Interrupt generation, presentation, and recognition are discussed in Section 4.3. In most cases, event counter overflow interrupt processing simply is a matter of incrementing a software counter, and most measurements will be made without counter overflow occurring. However, it is possible to use an event counter as an interval timer of sorts, so that an interrupt is generated after, for example, a specified number of instructions have been executed, as discussed later in this chapter.

The events to be counted by a particular counter are specified by setting the appropriate bits in the corresponding EvCtlr field in the ICR. The EvCtlr1 field, ICR bits <17:12>, controls Event Counter 1, and the EvCtlr2 field, ICR bits <11:6>, controls Event Counter 2. The ICR is a privileged register which can be accessed only in system mode. To set or clear EvCtlr field bits or the Event Counter Overflow Interrupt Enable flags, the operating system uses the following procedure.

1. If the PU is not already interrupt/trap disabled, interrupt/trap recognition is disabled by clearing the Interrupt/Trap Enable Flag in the PsR to "0".

2. A critical section is entered, using the Lock instruction, to insure that only one PU is attempting ICR access. (Disabling interrupt/trap recognition insures that the PU is not interrupted while in this critical section.)

3. The contents of the ICR are moved into a general register via a Move From Special instruction.

4. The desired EvCtlr field values or Event Counter Overflow Interrupt Enable flags are set using field manipulation instructions.

5. The Pending Rupts field, bits <31:25>, is set to all "1's" to insure that the currrent state of the pending interrupt flags will not be changed when the ICR is written. The pending interrupt flags, ICR bits <31:25>, are set to "1" only by hardware and cleared to "0" only by software. Attempting to write a "1" to a pending interrupt flag leaves the state of the flag unchanged.[1]

6. The contents of the general register are written to the ICR via a Move To Special instruction.

7. The critical section is exited.

[1] See ***Pending Interrupt Flags*** in Section 4.3.

Events which can be counted and the EvCtlr field values which select these events are shown in Figure 7.2. Events can be classified as MMU events or PU events: the high-order bit of the EvCtlr field, ICR bit <17> for EvCtlr1 or ICR bit <11> for EvCtlr2, selects the class. If this bit is "1", MMU events are counted: if "0", PU events are counted. The next two bits of the EvCtlr field, ICR bits <16:15> for EvCtlr1 or ICR bits <11:10> for EvCtlr2, select the mode in which events are to be counted"; "01" specifies user mode only, "10" specifies system mode only, and "11" specifies both system and user modes. If these two bits are "00", cycles are counted if the MMU event class was selected, and nothing is counted if the PU event class was selected. Thus, setting the high-order three bits of the EvCtlr field to "0" turns the counter off. Any other setting will cause the counter to run and eventually overflow; however, an Event Counter Overflow interrupt will be generated only the Event Counter Overflow Interrupt Enable flag for that counter is set to "1".

The interpretation of the high-order three bits of the EvCtlr field is summarized in the following table.

high-order 3 bits of EvCtlr	interpretation
"000"	counter off
"001"	count PU events in user mode
"010"	count PU events in system mode
"011"	count PU events in user & system modes
"100"	count time in cycles
"101"	count MMU events in user mode
"110"	count MMU events in system mode
"111"	count MMU events in user & system modes

In the MMU event class, countable events are instruction cache moveins, data cache moveins, data cache moveouts, singly or in sum, and Translation Buffer misses. Moveins are all lines read from memory (including prefetched lines); moveouts are all lines written to memory, including modified lines replaced on a movein and lines written to memory via a cache control instruction. When the high-order bit of the EvCtlr field is "1", the interpretation of the low-order three bits is as follows.

low-order 3 bits of EvCtlr	interpretation in MMU Event class
"000"	count TB misses
"001"	count instr. cache moveins
"010"	count data cache moveins
"011"	count both instr. & data cache moveins
"100"	count data cache moveouts
"101"	count data cache moveouts & instr. cache moveins
"110"	count data cache moveins & moveouts
"111"	count instr. & data cache moveins & moveouts

There is no provision for counting accesses to non-cached pages. In Antares, a TB miss results in either one or two one-word memory reads, depending on whether or not the directory entry is found in the Directory Buffer (see Section 3.7).

Events in the MMU event class are global, and reflect the activity of the CPU as a whole. Events in the PU event class (high-order bit of the EvCtlr field = "0") are local to the PU specified by the low-order two bits of the EvCtlr field. The remaining bit of the low-order three bits selects one of the two countable events in this class: PU active cycles or PU instructions executed.[2] PU active cycles are the cycles a PU spends in run state. A PU is in run state while executing instructions, even if instruction execution is temporarily delayed because of (e.g.) cache misses or register interlock delays. See Section 5.5 for a description of PU states.

On reset, the contents of the event counters are undefined and remain undefined until explicitly written; EvCtlr field and Event Counter Overflow Interrupt Enable flags are cleared to "0", so the counters initially are off with overflow interrupts disabled.

7.3 The Measurement Process

The primary use of the Scorpius measurement facilities is in improving the performance of application and system programs. For purposes of this discussion, it is assumed that the key measure of program performance is its CPU execution time. Other measures, such as number of instructions executed, number of cache moveins and moveouts, and level of parallelism may suggest ways in which performance can be improved. Generally, there is no absolute standard with which measurements can be compared, although experience with any given implementation tends to suggest, for example, when miss rates are "too high". The performance improvement process typically involves running the program and collecting measurements, examining the measurements to obtain some idea of where improvements might be made, revising the program, rerunning the program and collecting new measurements, and comparing the two sets of measurements to see if the anticipated performance improvement was realized and, if not, to gain some idea why not.

Since there are only two counters, collecting all available measurements for a particular program requires that the program be executed several times. This makes it desirable that program execution be repeatable, in these sense that repeated runs produce essentially identical measurement results. Not all programs are repeatable. In particular, run-to-run differences in IO device behavior, task switching, and other environmental aspects can affect cache

[2] Antares counts a folded instruction pair as two instructions, and counts **LdM/StM** as single instructions. When counting instructions, Antares counts by two's (i.e., increments the counter by two after two instruction have been executed, rather than incrementing it by one after one instruction has been executed), leaving the least significant bit of the counter unchanged from its initial value.

misses and consequently program execution time.[3] Some experimentation may be necessary to determine if this problem exists; if it does, it may be necessary to construct a test environment in which to carry out performance evaluation.

The initial set of runs of a user-mode-only application program might collect the following counts.

run 1: CPU time in cycles (**T_c**) and moveins plus moveouts (**M + W**)

run 2: for PU 0, instructions executed (**N_0**) and active cycles (**A_0**)

run 3: for PU 1, instructions executed (**N_1**) and active cycles (**A_1**)

run 4: for PU 2, instructions executed (**N_2**) and active cycles (**A_2**)

run 5: for PU 3, instructions executed (**N_3**) and active cycles (**A_3**)

The following measurements can be computed from these counts.

N = total instructions executed = $N_0 + N_1 + N_2 + N_3$

A = total PU active cycles = $A_0 + A_1 + A_2 + A_3$

P = average number of active PUs = A/T_c

U_B = approximate memory path utilization
= [(**M** + **W**) x (mean line transfer time in cycles)]/**T_c**

I = mean instruction execution time in cycles = **A/N**

Note that $T_c = N \times I/P$; program execution time equals the total number of instructions executed multiplied by the mean instruction execution time, divided by the average number of active PUs. **I** includes cache miss processing time, and is a function of **U_B**, and the objective of the performance improvement process is to reduce **T_c** .

The two most important measurements are **P**, the average number of active PUs, and **U_B**, the approximate memory path utilization. The term "memory path" is used to describe the serial part of the path via which lines are transferred between the cache and memory; this includes, but is not limited to, the memory bus and memory itself. The mean line transfer time is the average number of cycles this serial path is busy when a line is moved in or moved out. This time is implementation-dependent and, because of overlap situations, usually will have to be estimated. Also, line transfer times for CPU ↔ local memory and CPU ↔ remote memory transfers may differ. As defined above, **U_B** ignores word transfers for non-cached pages and for directory and page table entries read on TB misses. For many applications, however, this approximate value of **U_B** is adequate. The minimum value of **U_B** is 0 (i.e., no misses occurred); the maximum value may be 1 but often is less because a gap may be required between successive transfers (e.g., for bus pre-charge).

[3] It is assumed that the kernel associates measurements with address spaces and, on an address space switch, saves and restores the event counters, event counter controls, and counter overflow interrupt enable flags as necessary.

If $\mathbf{U_B}$ is close to its maximum, then performance is memory-path-limited.[4] There is no point in trying to improve parallelism or "tune" code in this case (except that other implementations may not be memory path-limited). The only way to improve performance is to reduce the number of moveins and moveouts. The next step, then, is to make two more runs and collect the following counts.

run 6: instr. cache moveins ($\mathbf{M_i}$) and data cache moveins ($\mathbf{M_d}$)

run 7: data cache moveouts (**W**) and TB misses ($\mathbf{M_{TB}}$)

This decomposition of memory traffic provides a starting point for performance improvement efforts. Program reorganization (e.g., moving infrequently-used or error-handling code out of the "mainstream") can reduce instruction cache moveins. Data structure redesign (e.g., using a hash table instead of a linked list) and reorganization (e.g., grouping modified variables) can reduce data cache moveins and moveouts.

A high Translation Buffer miss rate — on the same order of magnitude as the cache miss rate — sometimes indicates an access ordering problem, such as a program accessing elements of a large array in an order contrary to that array's storage form. This problem usually results in high data cache miss rates as well. Certain TB/TLB designs may evidence high miss rates because of thrashing, as when a loop touching three particular pages executes on a CPU with a 2-way set associative TB/TLB. (The Antares TB is fully associative.)

P, the average number of active PUs, is a measure of the level of parallelism realized by the application. A PU is inactive when it is halted, when it is waiting for a broadcast instruction to be executed by some other PU, or when it is waiting for a semaphore. Improvements in **P** can sometimes be realized by simple coding changes, but at other times can be achieved only by various levels of algorithm redesign. Compiler reference manuals may suggest programming techniques which enhance the compiler's ability to generate parallel code. Note that as program changes improve **P**, $\mathbf{T_c}$ decreases, so that even though **M + W** is unchanged, $\mathbf{U_B}$ increases.

When $\mathbf{U_B}$ and **P** cannot be further improved, some additional performance may be gained by attempting to reduce **I**, the mean instruction execution time. (It is assumed that **N**, the number of instructions executed, has been reduced as much as possible.) Factors which effect **I** include cache miss delays, branch and prefix instruction folding (at least in Antares), general, product, and remainder register interlock delays, and the relative frequency of multi-cycle instructions such as Load Multiple. Reordering instructions can increase the number of folded instructions and reduce or even eliminate register interlock delays.

[4] Being memory-path-limited is not necessarily an undesirable state; in the end, performance is limited by some resource or other. For maximum performance, use of the limiting resource must be optimized.

It is expected that the operating system function which provides measurement services will clear event counters as part of count initialization, so that event counter overflow rarely will occur in most measurement experiments. (Nevertheless, the operating system must anticipate overflow.) However, an event counter can be used as a form of interval timer by initializing to something other than a zero value and performing a measurement operation when the event counter overflow interrupt occurs. For example, the time spent by an application in various states and modes can be measured by sampling. Initialize a counter to a value $(2^{32}-1) - r$, where r is a random variate with mean **R** cycles, and set the counter's controls to count total cycles. On each overflow interrupt, save bits <15:4> of the Global Status Register (GSR) and set the counter once again to $(2^{32}-1) - r$, where r is a new random variate. On completion of execution, $\mathbf{T_c}$ /**R** samples will have been collected. From the average values of the GSR flags, estimates of the proportions of time spent in halt, wait, and run state, and in user and system mode, can be computed for each PU and for the CPU as a whole, and the average number of busy PUs can be estimated.

8. Instructions

8.1 Introduction

This chapter specifies the format and operation of the 81 instructions which compose the Scorpius instruction set. For description purposes, the instruction set is divided, by function, into the eight groups shown in the instruction set summary of Figure 2.11, repeated here as Figure 8.1.

Instruction descriptions comprise an instruction operation description, a description of condition code changes which can occur as the result of execution of the instruction, a list of exceptions which may be caused by execution of the instruction, and the format of the instruction. Supplementary notes are appended when necessary.

Instruction operation descriptions begin with an operation description statement; for example, "**LdB @RegA → RegB**" is the operation description statement for the Load Byte instruction. The notation and operators used in these operation statements are defined in Figures 8.2 and 8.3.

An exception is an event such as a page fault or an arithmetic overflow which can be caused by execution of an instruction and which can result in generation of a trap (see Chapter 4). The exceptions listed for an instruction do not include instruction page faults or instruction access privilege violations; these two exceptions result from an instruction's place of residence, and not from its execution.

Instruction formats show the length of each instruction field and give the instruction's operation code in binary. While all Scorpius instructions are 16 bits in length, instructions are tightly encoded, with operation codes varying in length, and there are more than a dozen instruction formats. Appendix A shows the various instruction formats and lists instructions by operation code length.

MNEMONIC	OPERATION	MNEMONIC	OPERATION
	LOAD, STORE, AND MOVE		*ARITHMETIC*
LdI	Load Immediate	**Add/Sub**	Add/Subtract
LdR/StR	Load/Store Word (Register)	**AddC/SubC**	Add/Subtract with Carry
LdRD/StRD	Load/Store Word (Base + Disp.)	**AddI/SubI**	Add/Subtract Immediate
LdB/StB	Load/Store Byte	**AddP/SubP**	Add/Subtract Partial
LdM/StM	Load/Store Multiple	**CLZ**	Count Leading Zeroes
Lcc	Load Condition	**Div**	Divide
LdCP	Load Carry Partial	**DivE**	Divide Extended
LdPC	Load Program Counter	**DivU**	Divide Unsigned
LdPU	Load PU Number	**DivUE**	Divide Unsigned Extended
Mov	Move Register	**Mul**	Multiply
MovFS	Move From Special	**MulU**	Multiply Unsigned
MovTS	Move To Special	**MulP**	Multiply Partial
	BRANCH, COMPARE, & JUMP	**MulPU**	Multiply Partial Unsigned
Bcc	Branch Relative on Condition	**Neg**	Negate
Cmp	Compare Register		*BROADCAST & SEMAPHORE*
CmpI	Compare Immediate	**Rcv**	Receive
CmpP	Compare Partial	**Rsm**	Resume PUs
Jmp	Jump Relative	**Send**	Send
JmpL	Jump and Link	**Strt**	Start PUs
JmpR	Jump Register	**Wait**	Wait PUs (or Halt)
TstF	Test Field	**Lock**	Lock Semaphore
TstM	Test Mode	**Unlk**	Unlock Semaphore
	LOGICAL & SHIFT		*CACHE CONTROL*
And	And	**CDC**	Create Data Cache line
AndC	And Complement	**FDC**	Flush Data Cache line
Not	Not	**IDC**	Invalidate Data Cache line
Or	Or	**IIC**	Invalidate Instruction Cache line
XOr	Exclusive Or	**IICA**	Invalidate Instruction Cache
Dsh	Shift Double	**PDC**	Prefetch Data Cache line
ShL	Shift Left	**RDTX**	Read Data Tag by Index
ShR	Shift Right	**UDC**	Update Data Cache line
	FIELD MANIPULATION	**VDC**	Validate Data Cache line
ClrF	Clear Field		*CONTROL & MISCELLANEOUS*
Dep	Deposit	**ClrM**	Clear Mode
ExtS	Extract Signed	**Prmpt**	Preempt PUs
ExtU	Extract Unsigned	**Res**	Restart PUs
Ins	Insert	**RtI**	Return from Interrupt
Msk	Define Field	**SetM**	Set Mode
PfxI	Prefix Immediate	**Trap**	System Call
SetF	Set Field		

Figure 8.1. Scorpius Instruction Set Summary

RegA	value contained in the **RegA** field of an instruction (bits <3:0>); the register specified by this value is called register **RegA**[1]
RegB	value contained in the **RegB** field of an instruction (bits <7:4>); the register specified by this value is called register **RegB**[1]
Base	value contained in the **Base** field of an instruction (bits <1:0>); the register specified by this value is called register **Base**[1]
(**R**)	indicates the contents of register **R** used as an operand; (**RegA**) is read "... the contents of register number **RegA**..." or "... the value in register **RegA** ..."
@R	indicates the contents of register **R** used as an address; e.g., **@Base** is read "... the address contained in register number **Base**..."
(**R**)<H/B>	indicates either the four bytes or two half words, depending on the value of the H/B mode flag in the PsR, contained in register **R**
(**R**)<m:n>	bits m through n, inclusive, of register **R**
(**PfxR**)	contents of or value in the Prefix Register. In instruction descriptions, this value is assumed to have its least significant bit at Prefix Register bit <2> (i.e., the fixed 0's in Prefix Register bits <1:0> are ignored).
PV	Prefix Valid flag (PsR bit <3>)
<pos,len>	implicit operands of field manipulation instructions, which operate on fields described by the position of their least significant bit and their length. **pos** is the value contained in PfxR bits <11:7> and specifies the position of the least significant bit; **len** is one plus the value contained in PfxR bits <6:2> and specifies the field length.
Disp	value contained in the displacement field of relative branch and jump instructions, and load/store base plus displacement instructions
Imm	value contained in the immediate field of the add, compare, load, and subtract immediate instructions

Figure 8.2. Notation Used in Instruction Operation Descriptions

Values in immediate and displacement fields of certain Scorpius instructions (**AddI**, **LdI**, and **LdRD/StRD**) are incremented prior to their use. This increases the maximum value which can be encoded. For example, Add Immediate (**AddI**) has an 8-bit immediate field which can hold values 0 - 255. However, the effective immediate value used by this instruction is (without prefixing) the value in this field, **Imm**, plus one, and so the *effective* immediate range is 1 - 256. In these cases, the Pyxis assembler encodes immediate and displacement fields by subtracting 1. For **AddI**, Pyxis generates the instruction's immediate field value as **Imm** = (*IMMEDIATE* - 1) modulo 256, where *IMMEDIATE* is the value specified in the assembler statement's immediate field. When used

[1] In instruction description operation statements, "register" is implicit: **Mov** (**RegA**) → **RegB** is read "move the contents of register (number) **RegA** to register (number) **RegB**".

A + *B*	add *A* and *B*
A – *B*	subtract *B* from *A*
A **B*	multiply *A* (multiplicand) by *B* (multiplier)
A /*B*	divide *A* (dividend) by *B* (divisor)
~*A*	one's complement *A*
A & *B*	bitwise AND *A* and *B*
A | *B*	bitwise OR *A* and *B*
A ^ *B*	bitwise EXCLUSIVE OR *A* and *B*
A <<*B*	shift *A* left *B* bit positions
A >>*B*	shift *A* right *B* bit positions
A → *B*	store the value specified by *A* in the register or memory location specified by *B*; e.g., (**RegB**) -> @**RegA** is read "store the contents of register **RegB** in memory at the address contained in register **RegA**".
A || *B*	concatenate *A* and *B*; *A* occupies the most significant bits of the result
[]	indicate precedence: in "[D=+1]<<2", 1 is added to D and the sum then is shifted left 2 places

Figure 8.3. Operators Used in Instruction Operation Descriptions

without prefixing, the range of *IMMEDIATE* is assumed to be 1 through 256; with prefixing, the range of *IMMEDIATE* is assumed to be 0 through 255, with 0 encoded as 0xFF. Pyxis handles the immediate for **LdI** and the displacements for **LdRD/StRD** in the same way. For Subtract Immediate (**SubI**), the effective immediate is 16 - **Imm**, and Pyxis encodes the **SubI** statement's immediate value by subtracting it from 16: **Imm** = 16 - *IMMEDIATE* . Similarly, the effective shift amount, *AMOUNT*, for Shift Left (**ShL**) is 31 – **Amt**, where **Amt** is the value stored in the shift left instruction's immediate field. Pyxis encodes the immediate field value by subtracting it from 31: **Amt** = 31 – *AMOUNT*.

8.2 Load, Store, and Move Instructions

This instruction group comprises instructions which load or store words (**LdR/StR** and **LdRD/StRD**), load or store bytes (**LdB/StB**), load or store multiple words (**LdM/StM**), load a constant into a register (**LdI**), load a specified state value into a register (**Lcc**, **LdCP**, **LdPC**, **LdPU**), or move values between two general registers (**Mov**) or between a general register and a special register (**MovFS** and **MovTS**). The descriptions of these instructions appear in the order in which the instructions in this group are listed in Figure 8.1. Descriptions for symmetrical instructions (load/store, move from/to) are combined.

LOAD IMMEDIATE *LdI*

LdI	**Imm** + 1 → **RegA**	if **PV** = "0"
	(**PfxR**)<<8 + **Imm** +1 → **RegA**	if **PV** = "1"

Load the effective immediate value into register **RegA**. If the Prefix Valid flag (**PV**) is "0", the effective immediate value is **Imm** + 1. This value is stored in bits <7:0> of register **RegA**; bits <31:8> are cleared to "0's". If the Prefix Valid flag is "1", the effective immediate is formed by shifting the contents of the **PfxR** left 8 places and adding **Imm**, and then adding 1 to the result; clear the Prefix Valid flag to "0".

Condition Codes: unchanged

Exceptions: none

Notes: To load an immediate value of 0 without affecting condition codes, use **Lcc** with **cc** = false.

Format:

15 12	11 4	3 0
0110	Imm	RegA
	i_7 i_6 i_5 i_4 i_3 i_2 i_1 i_0	

LOAD/STORE REGISTER — *LdR/StR*

LdR	**@RegA** → **RegB**	if **PV** = "0"
	@RegA + (**PfxR**)<<2 → **RegB**	if **PV** = "1"
StR	(**RegB**) → **@RegA**	if **PV** = "0"
	(**RegB**) → **@RegA**+ (**PfxR**)<<2	if **PV** = "1"

Load the addressed word from memory into register **RegB** (**LdR**) or store the word in register **RegB** in memory (**StR**). If the Prefix Valid flag is "0", the memory address of the word to be loaded or stored is contained in register **RegA**. If the Prefix Valid flag is "1", the memory address is formed by shifting the contents of the **PfxR** left 2 places and adding the result to the address in register **RegA**; clear the Prefix Valid flag. (The Prefix Register provides a signed word displacement from the register **RegA** address.)

Condition Codes: unchanged

Exceptions: data page fault; data access privilege violation

Format:

	15 — 8	7 — 4	3 — 0
LdR	00010101	RegB	RegA
StR	00000101	RegB	RegA

LOAD/STORE REGISTER + DISPLACEMENT *LdRD/StRD*

LdRD	**@Base** + [**Disp** + 1]<<2 → **RegB**	if **PV** = "0"
	@Base + [(**PfxR**)<<6 + **Disp** + 1]<<2 → **RegB**	if **PV** = "1"
StRD	(**RegB**) → **@Base** +[**Disp** + 1]<<2	if **PV** = "0"
	(**RegB**) → **@Base** + [(**PfxR**)<<6 + **Disp** + 1]<<2	if **PV** = "1"

Form the effective byte displacement and add it to the byte address in register **Base** to form the effective operand address. Load the addressed memory word into register **RegB** (**LdRD**) or store the word in register **RegB** in memory (**StRD**). If the Prefix Valid flag is "0", the effective byte displacement is formed by shifting **Disp** + 1 left 2 places. (**Disp** + 1 is the effective *word* displacement.) If the Prefix Valid flag is "1", the effective byte displacement is formed by shifting the contents of the **PfxR** left 6 places and adding **Disp**, adding 1, and shifting the result left 2 places; the Prefix Valid flag is cleared. ((**PfxR**)<<6 + **Disp** + 1 is the effective *word* displacement).

Condition Codes: unchanged

Exceptions: data page fault; data access privilege violation

Format:

	15 12	11 8	7 4	3 2	1 0
LdRD	1001	Disp	RegB	Disp	Base
StRD	1000	Disp	RegB	Disp	Base
		d_5 d_4 d_3 d_2		d_1 d_0	

LOAD/STORE BYTE *LdB/StB*

LdB **@RegA → RegB**

StB **(RegB) → @RegA**

LdB loads the memory byte addressed by register **RegA** into bits <7:0> of register **RegB** and clears bits <31:8> of register **RegB** to "0's". **StB** stores the rightmost byte of **RegB** (bits <7:0>) in the memory byte addressed by **RegB**. Both instructions increment the address in **RegA** by one.

Condition Codes: unchanged

Exceptions: data page fault; data access privilege violation

Notes: An attempted operand access to a non-cached page by **LdB** or **StB** causes a data access privilege violation.

If attempted execution of this instruction results in generation of a Data Page Fault or Data Access Privilege Violation trap, the Rmod flag in the PsR (bit <31>) is set to "1", indicating that adjustment of the contents of the address register of the interrupted instruction is required before execution of the instruction is reattempted on return from interrupt. For **LdB/StB**, this adjustment is effected by decrementing the contents of register RegA by one (see Sections 4.6 and 4.7).

Format:

	15 8	7 4	3 0
LdB	00010110	**RegB**	**RegA**
StB	00000110	**RegB**	**RegA**

LOAD/STORE MULTIPLE — *LdM/StM*

LdM **@RegA** → 1...**RegB**

StM **(RegB...1)** → **@RegA**

LdM loads registers 1 through **RegB** from memory. On **LdM** initiation, register **RegA** contains address of the first word to be loaded (into register 1); on completion, the address in register **RegA** is incremented by 4 x **RegB** (i.e., by the number of bytes loaded). **StM** stores the contents of registers **RegB** through 1 to memory. On **StM** initiation, register **RegA** contains the address plus 4 of the first word to be stored (the address plus 4 of the word into which the contents of register **RegB** are to be stored); on completion, the address in register **RegA** is decremented by 4 x **RegB** (ands so contains the address of the location in which register 1 was stored).

Condition Codes: unchanged

Exceptions: data page fault; data access privilege violation

Notes: **LdM/StM** operation is undefined if **RegA** is in the range 1 – **RegB** (i.e., if the address register is one of the registers loaded/stored), or if **RegB** is 0.

It is not required that incrementing or decrementing of the address in register **RegA** be performed as each register is loaded or stored; incrementing or decrementing can be done at any implementation-convenient point prior to instruction completion. Consequently, **LdM** and **StM** are preemptive-repeat instructions; if an interrupt or a trap is recognized during execution of either instruction, the instruction is restarted, not resumed, on return from interrupt. Since this can result in repeating load or store accesses to memory, these instructions should not be used certain operations, such as data transfers from and to IO locations.

When an interrupt or a trap is recognized during execution of a **LdM/StM** instruction, the Rmod flag in the PsR (bit <31>) may be set to "1", indicating that it is necessary to adjust the address in register **RegA** to its initial value before execution of the instruction is reattempted on return from interrupt. For **LdM**, this adjustment is effected by subtracting 4 x **RegB** from the contents of register **RegA**. For **StM**, this adjustment is effected by adding 4 x **RegB** to the contents of register **RegA**. (See Sections 4.6 and 4.7.)

Format:

	15 8	7 4	3 0
LdM	00010111	RegB	RegA
StM	00000111	RegB	RegA

LOAD CONDITION — *Lcc*

Lcc   RegB

If the current settings of the condition codes correspond to the relation defined by the **cc** field of the instruction, set bit <0> of register **RegB** to "1"; otherwise, clear this bit to "0". In either case, clear bits <31:1> to "0's". The correspondence between the **cc** portion of the instruction mnemonic, the **cc** field of the instruction (instruction bits <3:0>), and the condition code settings examined by this instruction are tabulated below.

cc field		*condition code settings*	*interpretation*
encoding	*mnemonic*		
0	F	*not applicable*	false
1	OV	V = 1	overflow
2	LO	C = 0	lower than
3	LT	N = 1	less than
4	*none*	*not applicable*	*undefined*
5	EQ	Z = 1	equal
6	LS	C = 0 | Z = 1	less than or same
7	LE	N = 0 | Z = 1	less than or equal
8	*none*	*not applicable*	*undefined*
9	NV	V = 0	no overflow
10	HS	C = 1	higher than or same
11	GE	N = 0	greater than or equal
12	*none*	*not applicable*	*undefined*
13	NE	Z = 0	not equal
14	HI	C = 1 & Z = 0	higher than
15	GT	N = 0 & Z = 0	greater than

A **cc** field value of "0x0" (instruction mnemonic **LF**) causes register **RegB** to be cleared to "0's" regardless of the settings of the condition codes. This provides a means of loading an immediate value of "0" into a register without affecting condition codes (**LdI**, without prefixing, loads values in the range 1 - 256.)

Condition Codes: unchanged

Exceptions: none

Notes: The results are unpredictable if a **cc** field value of 4, 8, or 12 is used.

Format:

15 8	7 4	3 0
00010001	RegB	cc
		b_3 b_2 b_1 b_0

LOAD CARRY PARTIAL — *LdCP*

LdCP (C0...C3) → RegA

Sign extend the carry condition codes according to the mode specified by the Halfword/Byte (H/B) mode flag in the PsR (bit <2>), and store the result in register **RegA**. In byte mode (H/B = "0"), each byte's carry code, C0, C1, C2, and C3, is sign extended into 8 bits. In halfword mode (H/B = "1"), each halfword carry code, C0 and C2, is sign extended into 16 bits Figure 2.16 shows the relationship between carry condition codes and the byte and halfword of a word.

Condition Codes: unchanged

Exceptions: none

Format:

15 4	3 0
0 0 0 0 0 0 0 0 0 0 1 0	RegA

LOAD PROGRAM COUNTER — *LdPC*

LdPC (PC) + 2 → RegA

LdPC loads its memory address plus 2 into register RegA.

Condition Codes: unchanged

Exceptions: none

Notes: The sequence Jmp-LdPC can be used to synthesize PC-relative procedure calls.

Format:

15 4	3 0
0 0 0 1 0 0 0 0 0 1 1 0	RegA

LOAD PU NUMBER *LdPU*

LdPU PU# → RegA

Load the number (0 - 3) of the PU executing the instruction into register **RegA**.

Condition Codes: unchanged

Exceptions: none

Notes: This is the only means provided for a PU to determine its identity.

Format:

15 4	3 0
0 0 0 0 0 0 0 0 0 0 1 1	RegA

MOVE REGISTER *Mov*

Mov (RegA) → RegB

Move the contents of register **RegA** to register **RegB**; register **RegA**'s contents remain unchanged.

Condition Codes: unchanged

Exceptions: none

Notes: A move with **RegA = RegB** has no effect and can be used as a no operation instruction.

Format:

15 8	7 4	3 0
0 0 0 0 1 0 1 1	RegB	RegA

MOVE FROM/TO SPECIAL — *MovFS/MovTS*

MovFS **(RegA) → RegB**

MovFS **(RegB) → RegA**

MovFS moves the contents of special register **RegA** to general register **RegB**. **MovTS** moves the contents of general register **RegB** to special register **RegA**. The scope (L - local, G - global), type (P - privileged, N - non-privileged) and number of each special register is shown below.

0	Id Register (**IdR**)	G/P
1	Interrupt Argument Register (**IAR**)	G/P
2	Test Register (**TestR**)	G/P
3	Global Status Register (**GSR**)	G/P
4	Product Register (**ProdR**)	L/N
5	Remainder Register (**RemR**)	L/N
6	Prefix Register (**PfxR**)	L/N
7	*reserved*	
8	Event Counter 1 (**EvR1**)	G/P
9	Event Counter 2 (**EvR2**)	G/P
10	Scratch Register (**ScR**)	G/P
11	Interrupt Control Register (**ICR**)	G/P
12	Status Save Register (**SaveR**)	L/P
13	Trap Register (**TrapR**)	L/P
14	PC Save Queue (**PCQ[1] & PCQ[2]**)	L/P
15	*reserved*	

Condition Codes: For **MovFS**, condition codes are set as follows:

N— if special register **RegB** is local, N is set to the value of bit <31> of that register; if special register **RegB** is global, N is unchanged.

Z— if special register **RegB** is local, Z is set to "1" if the contents of that register are zero and to "0" otherwise; if special register **RegB** is global, Z is unchanged.

V— if special register **RegB** is local, V is cleared to "0"; if special register **RegB** is global, V is unchanged.

C— unchanged

For **MovTS**, condition codes remain unchanged.

Exceptions: operation fault (on access to a privileged or reserved special register while in user mode)

Notes: The PU must be interrupt/trap disabled (PsR bit <15> = "0") when reading or writing the PCQ, SaveR, or TrapR,or the result of the move is unpredictable. External interrupts must be disabled (ICR bit <5> = "0") when reading the IAR, or the results are unpredictable. The result of an attempted access to a reserved special register is undefined. When reading special registers with reserved fields, the values returned in the bit positions corresponding to those fields are undefined. The GSR cannot be written. Bits <31:25> of the ICR (the pending interrupt flags) can be cleared to "0" only, not set to "1", via a **MovTS** instruction. See the Special Registers subsection in Section 2.3.

The Antares PCQ is a FIFO register pair in which only PCQ[1] can be directly read and only PCQ[2] can be directly written; see Section 4.6.

In Antares, writing special register IdR invalidates all entries in the Translation Buffer and associated Directory Buffer (see Section 3.6).

In Antares, if a **MovFS** instruction is immediately preceded by an instruction which causes the contents of the special register accessed by the **MovFS** instruction to be modified, the result of the move is unpredictable. The result of a **MovFS** instruction reading the GSR is unpredictable if the preceding instruction was a **SetM** or **ClrM** instruction changing the state of the Interrupt/Trap Enable flag (PsR bit 15>) or the User/System Mode flag (PsR bit 13>).

Format:

	15 8	7 4	3 0
LdM	00010111	RegB	RegA
StM	00000111	RegB	RegA

8.3 Branch, Compare, and Jump Instructions

This group comprises the conditional branch instruction (**Bcc**), the compare instructions (**Cmp**, **CmpI**), the unconditional jump instructions (**Jmp**, **JmpR**, **JmpL**), and the two test instructions, Test Field (**TstF**) and Test Mode (**TstM**).

Scorpius has delayed branches and jumps; the sequential instruction following a branch instruction always is executed, regardless of whether or not the branch is taken. Similarly, the sequential instruction following a jump instruction always is executed.

Future implementations of Scorpius may extend prefixing to include relative branch (**Bcc**) and jump (**Jmp**) instruction displacements. The displacements of these instructions cannot be extended by prefixing in Antares. However, on Antares, execution of a Bcc or Jmp instruction causes the Prefix Valid flag to be cleared so that forward compatibility can be realized. See the ***Branch Displacement Prefixing*** subsection in Section 2.4.

BRANCH ON CONDITION — Bcc

Bcc **@PC + Disp<<1**

If the current settings of the condition codes correspond to the relation defined by the **cc** field of the instruction, transfer control to the target address after executing the next sequential instruction; otherwise, continue execution. In either case, clear the Prefix Valid flag. The target address is formed by shifting the value in the instruction's displacement field, **Disp**, left one and adding it to the address in Current PC (i.e., the address of the **Bcc** instruction). **Disp** is an 8-bit signed value which provides an effective displacement of -256 to +255 instruction locations (halfwords). The correspondence between the **cc** portion of the instruction mnemonic, the **cc** field of the instruction (instruction bits <3:0>), and the condition code settings examined by this instruction are tabulated below.

cc field		*condition code settings*	*interpretation*
encoding	*mnemonic*		
0*	*none*	*not applicable*	false
1	OV	V = 1	overflow
2	LO	C = 0	lower than
3	LT	N = 1	less than
4*	*none*	*not applicable*	*undefined*
5	EQ	Z = 1	equal
6	LS	C = 0 | Z = 1	less than or same
7	LE	N = 0 | Z = 1	less than or equal
8*	*none*	*not applicable*	*undefined*
9	NV	V = 0	no overflow
10	HS	C = 1	higher than or same
11	GE	N = 0	greater than or equal
12*	*none*	*not applicable*	*undefined*
13	NE	Z = 0	not equal
14	HI	C = 1 & Z = 0	higher than
15	GT	N = 0 & Z = 0	greater than

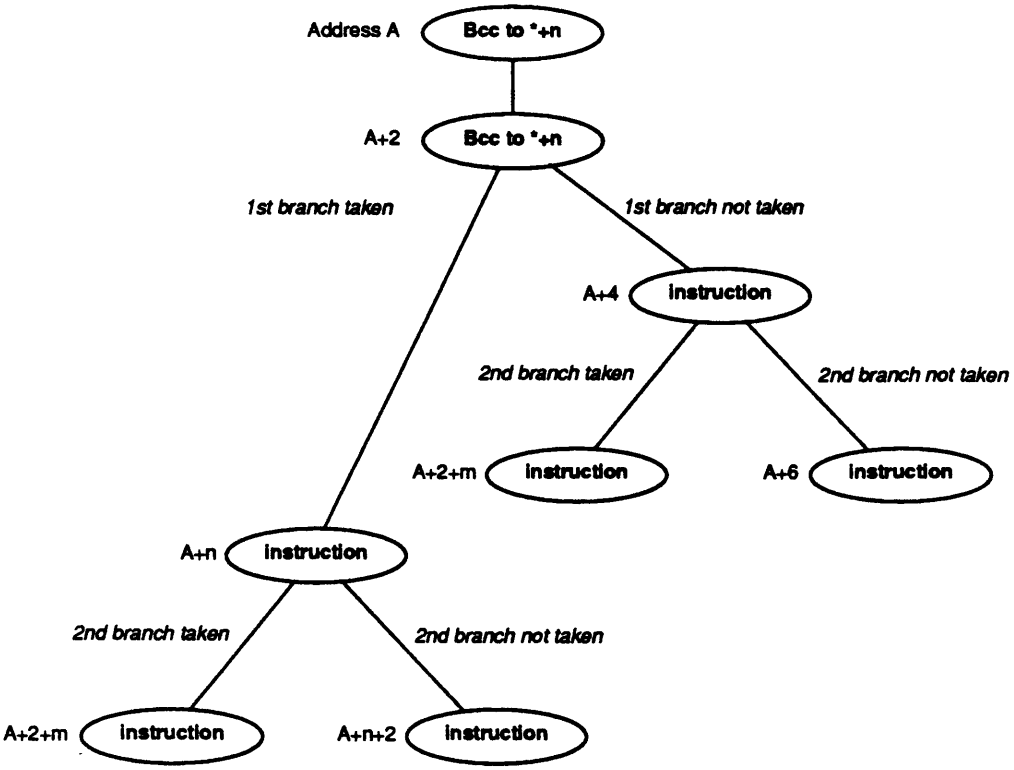
*invalid

Condition Codes: unchanged

Exceptions: taken branch trap

Notes: Both **Bcc** and **Jmp** (Jump Relative) have the same three-bit operation code, "110B"; they are are distinguished by instruction bits <1:0>, which always are "00" for **Jmp**. Note that none of the valid **cc** field values for **Bcc** have bits <1:0> = "00".

The sequential instruction following a **Bcc** instruction (the branch shadow instruction) may be any Scorpius instruction, including a branch or jump. The diagram below shows the execution sequences which can result when the shadow instruction of a **Bcc** is itself a **Bcc**. (This diagram can be applied to various **Bcc** and jump instruction combinations by removing "not taken" paths for the latter.)



Format:

15 13	12 4	3 0
110	Displacement	cc
	d7 d6 d5 d4 d3 d2 d1 d0 sgn	

COMPARE *Cmp*

Cmp (**RegB**) - (**RegA**)

Compare the contents of register **RegB** with the contents of register **RegA** and set the condition codes according to the true arithmetic result of (**RegB**) - (**RegA**).

Condition Codes: N— set to "1" if (**RegB**) is arithmetically less than (**RegA**); cleared to "0" otherwise.

Z— set to "1" if (**RegB**) equals (**RegA**); cleared to "0" otherwise

V— cleared to "0"

C— if (**RegB**), treated as unsigned, is equal to or greater than (**RegA**), C0 is set to "1"; otherwise, C0 is cleared to "0". C1, C2, and C3 are cleared to "0".

Exceptions: none

Notes: The N condition code value resulting from **Cmp** will differ from the N condition code setting resulting from **Sub**, using the same operands, when an overflow condition exists.

Format:

15 8	7 4	3 0
00011000	RegB	RegA

COMPARE IMMEDIATE *CmpI*

Cmp **Imm** - (**RegA**) if **PV** = "0"
[(**PfxR**)<<8 + **Imm**] - (**RegA**) if **PV** = "1"

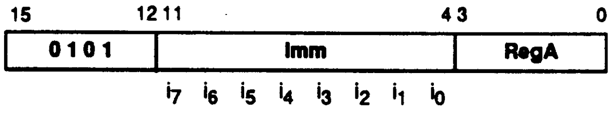
Compare the effective immediate value with the contents of register **RegA** and set the condition codes according to the true arithmetic result of the effective immediate - (**RegA**). If the Prefix Valid flag is "0", the effective immediate is **Imm**, the value stored in the instruction's immediate field. If the Prefix Valid flag is "0", the effective immediate is formed by shifting the contents of the Prefix Register (PfxR bits <31:2>) left 8 places and adding **Imm**; the Prefix Valid flag is cleared to "0". The range of **Imm** is 0 - 255.

Condition Codes:
N— set to "1" if (**RegA**) is arithmetically greater than the effective immediate; cleared to "0" otherwise.
Z— set to "1" if (**RegA**) equals the effective immediate; cleared to "0" otherwise
V— cleared to "0"
C— if the effective immediate, treated as unsigned, is equal to or greater than (**RegA**), C0 is set to "1"; otherwise, C0 is cleared to "0". C1, C2, and C3 are cleared to "0".

Exceptions: none

Notes: The effective immediate range for **CmpI** differs from that of the other two instructions of this format, **AddI** and **LdI**, which increment **Imm** prior to use and so have an effective immediate range range of 1 - 256.

Format:



COMPARE PARTIAL CmpP

Cmp (**RegB**)<H/B> - (**RegA**)<H/B>

Compare the bytes or halfwords, as determined by the H/B mode flag (PsR bit <2>), in register **RegB** with the corresponding bytes or halfwords in register **RegA**. Each byte or halfword in a register is treated as an independent operand (i.e., inter-byte or inter-halfword carries are forced to "1'). Set the condition codes according to the true arithmetic result of (**RegB**)<H/B> - (**RegA**)<H/B>.

Condition Codes: byte mode (H/B = "0")

N— set to "1" if the true arithmetic result of the comparison of the most significant bytes of registers **RegA** and **RegB** is negative; cleared to "0" otherwise. (Byte 0 — bits <31-24> — is the most significant byte.)

Z— set to "1" if the result of any byte comparison is zero (i.e., equal); cleared to "0" otherwise

V— cleared to "0"

C— Ci, i = 0, 1, 2, 3, is set to "1" if register **RegB** byte i, treated as unsigned, is equal to or greater than register **RegA** byte i, and cleared to "0" otherwise.

halfword mode (H/B = "1")

N— set to "1" if the true arithmetic result of the comparison of the most significant halfwords of registers **RegA** and **RegB** is negative; cleared to "0" otherwise. (Halfword 0 — bits <31-16> — is the most significant halfword.)

Z— set to "1" if the result of any halfword comparison is zero (i.e., equal); cleared to "0" otherwise

V— cleared to "0"

C— C0 is set to "1" if halfword 0 of register **RegB**, treated as unsigned, is greater than halfword 0 of register **RegA**, and cleared to "0" otherwise. C2 is set to "1" if halfword 1 of register **RegB**, treated as unsigned, is greater than halfword 1 of register **RegA**, and cleared to "0" otherwise. C1 and C3 are cleared to "0".

Exceptions: none

Format:

15 8	7 4	3 0
00011001	RegB	RegA

JUMP RELATIVE *Jmp*

Jmp @PC + **Disp**<<1

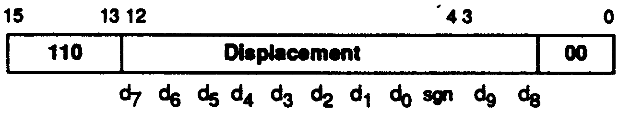
Transfer control to the target address after executing the next sequential instruction; otherwise, continue execution. In either case, clear the Prefix Valid flag. The target address is formed by shifting the value in the instruction's displacement field, **Disp**, left one bit position and adding it to the address in Current PC (i.e., the address of the **Jmp** instruction). **Disp** is an 11-bit signed value which provides an effective displacement of -1024 to +1023 instruction locations (halfwords).

Condition Codes: unchanged

Exceptions: taken branch trap

Notes: Both **Bcc** and **Jmp** (Jump Relative) have the same three-bit operation code, "110B"; they are are distinguished by instruction bits <1:0>, which always are "00" for **Jmp**. Note the position (below) of the two high-order bits of **Disp**.

Format:


15
13 12
4 3
0
110
Displacement
00
d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 sgn d_9 d_8

JUMP AND LINK *JmpL*

JmpL @RegA; (Next PC) + 2 → Register 4

Transfer control to the instruction address in register **RegA** after executing the next sequential instruction. Store the return address in general register 4. The return address (the address of the instruction following the shadow instruction) is formed by adding 2 to the contents of Next PC.

Condition Codes: unchanged

Exceptions: taken branch trap

Format:

15 4	3 0
0 0 0 0 0 0 0 0 1 0 1 1	RegA

JUMP REGISTER *JmpR*

JmpR @RegA

Transfer control to the instruction address in register **RegA** after executing the next sequential instruction.

Condition Codes: unchanged

Exceptions: taken branch trap

Notes: This instruction can be used to return from a subroutine.

Format:

15 4	3 0
0 0 0 0 0 0 0 0 1 0 1 0	RegA

TEST FIELD *TstF*

TstF (RegA) <pos,len>

Test the field in register **RegA** defined by the Prefix Register for a zero or a negative value, set the condition codes accordingly, and clear the Prefix Valid flag. The position of the least significant bit of the field, **pos**, is contained in Prefix register bits <11:7>. The length of the field, **len**, minus one, is contained in Prefix Register bits <6:2>.

Condition Codes: N— set to "1" if the most significant bit of the field is "1"; cleared to "0" otherwise.

Z— set to "1" if all bits of the field are "0"; cleared to "0" otherwise.

V— cleared to "0"

C— unchanged

Exceptions: taken branch trap

Format:

15 4	3 0
0 0 0 0 0 0 0 0 0 0 0 1	RegA

TEST MODE *TstM*

TstM No.

Set the N condition code to the value of PsR bit <**No.**>. PsR bits are listed below.

0	PU Available flag
1	Overflow Trap Enable flag
2	Halfword/Byte Mode flag
3	Prefix Valid flag
4-11	reserved
12	PCQ Enable flag
13	User/System Mode flag
14	Taken Branch Trap Enable flag
15	PU Interrupt/Trap Enable flag

Bits <15:8> are privileged and can be tested only in system mode.

Condition Codes: N— set to value of specified PsR bit
Z— unpredictable
V— cleared to "0"
C— unchanged

Exceptions: operation fault (on attempted access to bits <15:8> while in user mode)

Notes: The result of attempting to test a reserved PsR bit is unpredictable.

PsR flags are set and cleared by Set Mode (**SetM**) and Clear Mode (**ClrM**) instructions or via the transfer of SaveR contents to the PsR on execution of a Return from Interrupt (**RtI**) instruction. (**SetM**, **ClrM** and **RtI** are described in Section 8.9). In Antares, executing a **TstM** instruction immediately after an instruction which modifies the PsR (**ClrM**, **SetM**, **RtI**) may not return the modified value. Also, a correct value may not be returned when testing the Prefix Valid flag immediately following an instruction that modifies that flag.

Format:

0 0 0 1 0 0 0 0 0 0 0 1	No.
	b_3 b_2 b_1 b_0

8.4 Logical and Shift Instructions

Logical instructions include **And**, **AndC** (And Complement), **Not**, **Or**, and **Xor**, which are register-register operations; there are no register-immediate logical operations. Shift instructions are **Dsh** (Shift Double), **ShL**, and **ShR**), which perform logical shift operations. The shift amounts for **ShL** and **ShR** are obtained from immediate fields of those instructions; the shift amount for **Dsh** is obtained from the Prefix Register. Shift operations also can be effected using certain of the field manipulation instructions described in the next section; these permit using a shift amount determined at execution time. Also, **ExtS** (Extract Signed) can be used to effect an arithmetic right shift.

AND *And*

And **(RegB)** & **(RegA)** → **RegB**

AND the contents of register **RegB** with the contents of register **RegA** and store the result in register **RegB**.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Format:

15 8	7 4	3 0
00001010	RegB	RegA

AND COMPLEMENT *AndC*

And **(RegB) & ~(RegA) → RegB**

AND the contents of register **RegB** with the one's complement of the contents of register **RegA** and store the result in register **RegB**.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Format:

15 8	7 4	3 0
00011010	RegB	RegA

NOT *Not*

Not **~(RegA) → RegB**

One's complement the contents of register **RegA** and store the result in register **RegB**.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Format:

15 8	7 4	3 0
00011011	RegB	RegA

OR *Or*

Or (RegB) & (RegA) → RegB

OR the contents of register **RegB** with the contents of register **RegA** and store the result in register **RegB**.

Condition Codes:

- N— set to "1" if the result is negative; cleared to "0" otherwise
- Z— set to "1" if the result equals zero; cleared to "0" otherwise
- V— cleared to "0"
- C— unchanged

Exceptions: none

Format:

15 8	7 4	3 0
00001000	RegB	RegA

EXCLUSIVE OR *Xor*

XOr (RegB) ^ (RegA) → RegB

EXCLUSIVE OR the contents of register **RegB** with the contents of register **RegA** and store the result in register **RegB**.

Condition Codes:

- N— set to "1" if the result is negative; cleared to "0" otherwise
- Z— set to "1" if the result equals zero; cleared to "0" otherwise
- V— cleared to "0"
- C— unchanged

Exceptions: none

Format:

15 8	7 4	3 0
00001001	RegB	RegA

SHIFT DOUBLE *Dsh*

Dsh [(**RegB**) | | (**RegA**)] >> <**pos**> → **RegA**

Shift the double word formed by concatenating the contents of registers **RegB** and **RegA** right **pos** places and store the least significant 32 bits of the result in register **RegA**; clear the Prefix Valid flag. The shift amount, **pos**, is contained in Prefix Register bits <11:7>.

Condition Codes:

N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Notes: If **RegA** = **RegB**, the effect is to rotate the contents of register **RegA** right **pos** places.

Format:

15 8	7 4	3 0
00111111	RegB	RegA

SHIFT LEFT — *ShL*

ShL **(RegB)** << [31 - **Amt**] → **RegB**

Shift the contents of register **RegB** left 31 - **Amt** places, inserting "0's" in the vacated bit positions on the right (i.e., the least significant bits). The result is undefined if **Amt** equals 11111B.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

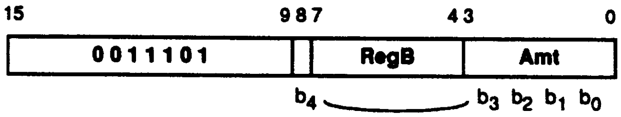
Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Format:



SHIFT RIGHT — *ShR*

ShR (RegA) >> Amt → RegA

Shift the contents of register **RegA** right **Amt** places, inserting "0's" in the vacated bit positions on the left (i.e., the most significant bits). The result is undefined if **Amt** is zero.

Condition Codes: N— cleared to "0"
Z— set to "1" if the result equals zero; cleared to "0" otherwise
V— cleared to "0"
C— unchanged

Exceptions: none

Notes: Extract Signed (**ExtS**) can be used to effect an arithmetic (sign extended) right shift.

Format:

15 9	8 4	3 0
0010110	Amt	RegA
	b_4 b_3 b_2 b_1 b_0	

8.5 Field Manipulation Instructions

The Clear/Set Field (**ClrF/SetF**), Deposit (**Dep**), Extract (**ExtS/ExtU**), and Insert (**Ins**) instructions, and the Test Field (**TstF**) instruction described in Section 8.3, operate on a bit field in a general register using a field description contained in the Prefix Register. Prefix Register bits <6:2> specify the field length, **len**, minus one, while bits <11:7> specify the right-most (low-order) bit position of the field, **pos**. A field description can be loaded into the Prefix Register via a Prefix Immediate (**PfxI**) instruction or a Define Field (**Msk**) instruction; the latter permits the field position to be specified by the contents of a register. Also, a field description can be constructed in a general register and moved to the Prefix Register via a Move To Special instruction. The move instruction does not set the Prefix Valid flag;, which can be set via a Set Mode instruction.

CLEAR/SET FIELD — *ClrF/SetF*

ClrF (**RegA**) **<pos,len>**

SetF (**RegA**) **<pos,len>**

ClrF clears the field in register **RegA** defined by the Prefix Register to "0's"; **SetF** sets the field to "1's". The Prefix Valid flag is cleared. Register **RegA** bits outside the field are not affected. The field length, **len**, minus one, is contained in Prefix Register bits <6:2>. The position of the rightmost (least significant) bit of the field, **pos**, is contained in Prefix Register bits <11:7>.

Condition Codes: unchanged

Exceptions: none

Format:

	15 … 4	3 … 0
ClrF	0 0 0 1 0 0 0 0 0 1 0 0	**RegA**
SetF	0 0 0 0 0 0 0 0 0 1 0 0	**RegA**

DEPOSIT *Dep*

Dep **(RegB)<len> → RegA<pos>**

Extract a right-justified field of length **len** (i.e., bits <**len**−1:0>) from register **RegB** and deposit it in register **RegA**, with its least significant bit at bit position **pos**. Clear the bits of register **RegA** outside the field and clear the Prefix Valid flag. The field length, **len**, minus one, is contained in Prefix Register bits <6:2>. The position of the rightmost (least significant) bit of the field, **pos**, is contained in Prefix Register bits <11:7>.

If **pos** + **len** ≥ 32, the contents of register **RegB**, left-shifted by **pos**, are stored into register **RegA**

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Notes: To extract a right-justified field from a source register and insert it in a destination register without clearing destination register bits outside the field, use the Insert (**Ins**) instruction.

Format:

15 8	7 4	3 0
00101001	RegB	RegA

EXTRACT SIGNED/UNSIGNED *ExtS/ExtU*

ExtS **(RegA)<pos,len> → RegB**

ExtU **(RegA)<pos,len> → RegB**

Extract a field of length **len**, whose least significant bit is at bit position **pos**, from register **RegA** and store it, right justified , in register **RegB**; clear the Prefix Valid flag. For **ExtS**, the most significant bit of the field is taken as the sign and the field is sign-extended in register **RegB** (i.e., bits <31:**len**> of register **RegB** are set to the value of the most significant bit of the field. For **ExtU**, the bits of register **RegB** outside the field (bits <31:**len**>) are cleared to "0's". The field length, **len**, minus one, is contained in Prefix Register bits <6:2>. The position of the rightmost (least significant) bit of the field, **pos**, is contained in Prefix Register bits <11:7>.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Notes: If **pos** + **len** ≥ 32, **ExtS** is equivalent to an arithmetic right shift of **pos** places; **ExtU** is equivalent to a logical right shift of **pos** places.

Format:

	15 8	7 4	3 0
ExtS	00101110	RegB	RegA
ExtU	00101111	RegB	RegA

INSERT *Ins*

Ins **(RegB)<len> → RegA<pos>**

Extract a right-justified field of length **len** (i.e., bits **<len**–1:0>) from register **RegB** and deposit it in register **RegA**, with its least significant bit at bit position **pos**; clear the Prefix Valid flag. Bits of register **RegA** outside the field are not affected. The field length, **len**, minus one, is contained in Prefix Register bits <6:2>. The position of the rightmost (least significant) bit of the field, **pos**, is contained in Prefix Register bits <11:7>.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise
Z— set to "1" if the result equals zero; cleared to "0" otherwise
V— cleared to "0"
C— unchanged

Exceptions: none

Notes: To extract a right-justified field from a source register and insert it in a destination register with destination register bits outside the field cleared, use the Deposit (**Dep**) instruction.

Format:

15 8	7 4	3 0
00111001	RegB	RegA

DEFINE FIELD *Msk*

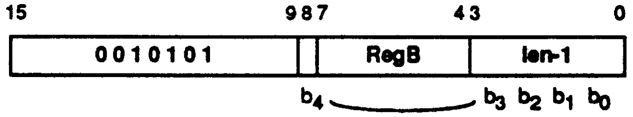
Msk **(RegB), len– 1**

Generate a field definition comprising a field length **len** and least significant (rightmost) bit position **pos**, store the definition in the Prefix Register, and set the Prefix Valid flag. The field length is in the range 1 - 32; the length minus 1 is specified by the **Msk** instruction's immediate field and stored in Prefix Register bits <6:2>. The least significant five bits of register **RegA** specify **pos**, and are stored in Prefix Register bits <11:7>. The most significant bit of the field is at bit position **pos** + **len** - 1. (If **pos** + **len** 1 ≥ 31, the most significant bit is at bit position 31.)

Condition Codes: unchanged

Exceptions: none

Format:



PREFIX IMMEDIATE *PfxI*

PfxI **Imm** → **PfxR** if **PV** = "0"

Imm + (**PfxR**)<<12 → **PfxR** if **PV** = "1"

If the Prefix Valid flag is "0", store **Imm** into the Prefix Register, sign-extended, and set the Prefix Valid flag. The high-order bit of **Imm** (instruction bit 11) is taken as the sign bit. If the Prefix Valid flag is "1", shift the contents of the Prefix Register left 12 places and store **Imm** in the vacated bit positions. Leave the Prefix Valid flag set.

Bits <1:0> of the Prefix Register are unused. The immediate field value **Imm** of the **PfxI** instruction is stored in Prefix Register bits <13:2>. When using **PfxI** to store a field description in the Prefix Register, instruction bits <4:0> specify **len** – 1, while instruction bits <8:5> specify **pos**.

Condition Codes: unchanged

Exceptions: none

Notes: In Antares, the result of executing a **PfxI** instruction immediately after a **MovTS** instruction storing into the Prefix Register is unpredictable.

Format:

15 12	11 0
0 1 1 1	Immediate
	i_{11} i_{10} i_9 i_8 i_7 i_6 i_5 i_4 i_3 i_2 i_1 i_0

8.6 Arithmetic Instructions

Arithmetic instructions include 32-bit signed integer add, and subtract instructions (**Add/Sub** and **AddC/SubC**), 32-bit signed and unsigned integer multiply and divide instructions (**Div/DivU** and **Mul/MulU**), 64-bit signed and unsigned integer divide instruction (**DivE/DivUE**), as well as add, subtract, and multiply instructions which operate on all four bytes or both halfwords of their operand registers (**AddP/SubP** and **MulP/MulPU**). Other arithmetic instructions include add and subtract immediate (**AddI/SubI**) and Negate (**Neg**).

AddP/SubP, and **MulP/MulPU** are called multi-gauge (or partial) arithmetic instructions, and are provided primarily to speed graphics operations.. The operand size — byte or halfword — for these instructions is specified by the Halfword/Byte mode flag in the PsR. Multigauge arithmetic instruction operands are treated as independent quantities (i.e., there are no carries between bytes or halfwords), and the specified arithmetic operation is carried out simultaneously on all four byte or on both halfwords (see Section 2.7).

ADD/SUBTRACT *Add/Sub*

Add **(RegB) + (RegA) → RegB**

Sub **(RegB) - (RegA) → RegB**

Add the contents of registers **RegB** and **RegA** (**Add**), or subtract the contents of register **RegA** from the contents of register **RegB** (**Sub**). Store the result in register **RegB**.

Condition Codes:

N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— set to "1" if an overflow is generated; cleared to "0" otherwise

C— C0 is set to "1" if a carry is generated and cleared to "0" otherwise; C1, C2, and C3 are cleared to "0". For **Sub**, C0 = "*not* borrow", and is set if **(RegB)** ≥ **(RegA)**, with **(RegA)** treated as unsigned.

Exceptions: arithmetic overflow

Format:

	15 8	7 4	3 0
Add	00001100	RegB	RegA
Sub	00011100	RegB	RegA

ADD/SUBTRACT WITH CARRY *AddC/SubC*

AddC **(RegB) + (RegA) + CO → RegB**

SubC **(RegB) – (RegA) + [CO – 1] → RegB**

AddC adds the value of the CO condition code (carry) flag to the contents of registers **RegB** and **RegA** and stores the result in register **RegB**. **SubC** adds the value of the CO condition code (carry) flag and the one's complement of the contents of register **RegA** to the contents of register **RegB** and stores the result in register **RegB**.

Condition Codes:

N— set to "1" if the result is negative; cleared to "0" otherwise

Z— cleared to "0" if the result is not equal to zero; left unchanged otherwise

V— set to "1" if an overflow is generated; cleared to "0" otherwise

C— CO is set to "1" if a carry is generated and cleared to "0" otherwise; C1, C2, and C3 are cleared to "0". For **SubC**, CO = "*not* borrow", and is set if **(RegB)** ≥ [**(RegA)** + CO - 1].

Exceptions: arithmetic overflow

Format:

	15 8	7 4	3 0
AddC	00001110	RegB	RegA
SubC	00011110	RegB	RegA

ADD IMMEDIATE *AddI*

AddI (**RegA**) + **Imm** – 1 → **RegA** if **PV** = "0"

(**RegA**) + [(**PfxR**)<<8 + **Imm** + 1] → **RegA** if **PV** = "1"

Add the effective immediate value to the contents of register **RegA** and store the result in register **RegA**. If the Prefix Valid flag is "0", the effective immediate value is **Imm** + 1. If the Prefix Valid flag is "1", the effective immediate is formed by shifting the contents of the **PfxR** left 8 places and adding **Imm**, and then adding 1 to the result; the Prefix Valid flag is cleared.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— set to "1" if an overflow is generated or if the effective immediate is – 2^{31} and (**RegA**) is negative; cleared to "0" otherwise

C— C0 is set to "1" if a carry is generated or if the effective immediate is 0 and is cleared to "0" otherwise. C1, C2, and C3 are cleared to "0".

Exceptions: arithmetic overflow

Format:

15 12	11 4	3 0
0100	Imm	RegA
	i_7 i_6 i_5 i_4 i_3 i_2 i_1 i_0	

SUBTRACT IMMEDIATE *SubI*

SubI **(RegA)** - [16 - **Imm**] → **RegA**

Subtract the effective immediate value from the contents of register **RegA** and store the result in register **RegA**. The effective immediate is formed by subtracting the immediate field value, **Imm**, from 16, providing an effective immediate range of 1 - 16.

Condition Codes: N— set to "1" if the result is negative; cleared to "0" otherwise

Z— set to "1" if the result equals zero; cleared to "0" otherwise

V— set to "1" if an overflow is generated; cleared to "0" otherwise

C— C0 is set to "1" if a carry is generated or if **(RegA)**, treated as unsigned, is equal to or greater than [16 - **Imm**], and is cleared to "0" otherwise. C1, C2, and C3 are cleared to "0".

Exceptions: arithmetic overflow

Notes: To subtract a constant value greater than 16, **AddI** with a negative prefix value is used.

Format:

15 8	7 4	3 0
00001111	Imm	RegA
	i_3 i_2 i_1 i_0	

ADD/SUBTRACT PARTIAL *AddP/SubP*

AddP **(RegB)**<H/B> + **(RegA)**<H/B> → **RegB**

SubP **(RegB)**<H/B> – **(RegA)**<H/B> → **RegB**

AddP adds the four bytes or the two halfwords, as determined by the H/B mode flag (PsR bit <2>), in register **RegA** to the corresponding bytes or halfwords in register **RegB**, and stores the result in register **RegB**. **SubP** subtracts the four bytes or two halfwords in register **RegA** from the corresponding bytes or halfwords in register **RegB** and stores the result in register **RegB**. Each byte or halfword pair is individually added or subtracted.

Condition Codes: byte mode (H/B = "0")

N— set to the value of bit <31> of the result

Z— set to "1" if the result of any byte addition or subtraction is zero ; cleared to "0" otherwise

V— cleared to "0"

C— Ci, i = 0, 1, 2, 3, is set to "1" if addition or subtraction of byte i generates a carry, and cleared to "0" otherwise.

halfword mode (H/B = "1")

N— set to the value of bit <31> of the result

Z— set to "1" if the result of any halfword addition or subtraction is zero ; cleared to "0" otherwise

V— cleared to "0"

C— C0 is set to "1" if addition or subtraction of halfword 0 generates a carry, and cleared to "0" otherwise. C2 is set to "1" if addition or subtraction of halfword 1 generates a carry, and cleared to "0" otherwise. C1 and C3 are cleared to "0".

Exceptions: none

Format:

	15 8	7 4	3 0
AddP	00001101	RegB	RegA
SubP	00011101	RegB	RegA

COUNT LEADING ZEROES *CLZ*

CLZ (RegA) → RegB

Count the number of leading zeroes (the number of contiguous "0" bits beginning at bit <31> and counting toward bit <0>) in register **RegA** and store this number in register **RegB**.

Condition Codes: N— set to "1" if the value in register **RegA** is negative; cleared to "0" otherwise

Z— set to "1" if the value in register **RegA** is zero; cleared to "0" otherwise

V— cleared to "0"

C— unchanged

Exceptions: none

Format:

15 8	7 4	3 0
00111101	RegB	RegA

DIVIDE *Div*

Div **(RegB)/(RegA) → RegB, RemR**

Divide the contents of register **RegB** (divisor) by the contents of register **RegA** (dividend); store the quotient in register **RegB** and the remainder in the Remainder Register (special register 5). The remainder will have the same sign as the dividend. Both dividend and divisor are treated as 32-bit signed integers.

Condition Codes: unchanged

Exceptions: arithmetic overflow

Notes: An arithmetic overflow exception occurs if the divisor is zero or if the result will not fit in the result register (the latter occurs only if (**RegB**) = -2^{31} and (**RegA**) = –1). If an overflow exception occurs and the Overflow Trap enable flag (PsR bit <1>) is "1", an Overflow Trap is generated and no result is stored; if the Overflow Trap enable flag is "0", no trap is generated and the result stored is unpredictable.

In Antares, **Div** is asynchronous and executes concurrently with subsequently-issued instructions. Overflow, if it occurs, will be detected prior to execution of any subsequent instruction.

Format:

15 8	7 4	3 0
00110011	RegB	RegA

DIVIDE EXTENDED DivE

DivE [(RemR) | | (RegB)]/(RegA) → RegB, RemR

Divide the 64-bit signed integer formed by concatenating the contents of the Remainder Register with the contents of register **RegB** by the 32-bit signed integer in register **RegA**; store the quotient in register **RegB** and the remainder in the Remainder Register (special register 5). The Remainder Register provides dividend bits <63:32> (Remainder Register bit <31> provides the sign), while register **RegB** provides dividend bits <31:0>. The remainder has the same sign as the dividend.

Condition Codes: unchanged

Exceptions: arithmetic overflow

Notes: An arithmetic overflow exception occurs if the divisor is zero or if the result will not fit in the result register. If an overflow exception occurs and the Overflow Trap enable flag (PsR bit <1>) is "1", an Overflow Trap is generated and no result is stored; if the Overflow Trap enable flag is "0", no trap is generated and the result stored is unpredictable.

In Antares, **DivE** is asynchronous and executes concurrently with subsequently-issued instructions. Overflow, if it occurs, will be detected prior to execution of any subsequent instruction.

Format:

15 8	7 4	3 0
0 0 1 1 0 1 1 1	RegB	RegA

DIVIDE UNSIGNED *DivU*

DivU (RegB)/(RegA) → RegB, RemR

Divide the contents of register **RegB** (divisor) by the contents of register **RegA** (dividend); store the quotient in register **RegB** and the remainder in the Remainder Register (special register 5). Both dividend and divisor are treated as 32-bit unsigned integers.

Condition Codes: unchanged

Exceptions: arithmetic overflow

Notes: An arithmetic overflow exception occurs if the divisor is zero. If an overflow exception occurs and the Overflow Trap enable flag (PsR bit <1>) is "1", an Overflow Trap is generated and no result is stored; if the Overflow Trap enable flag is "0", no trap is generated and the result stored is unpredictable.

In Antares, **DivU** is asynchronous and executes concurrently with subsequently-issued instructions. Overflow, if it occurs, will be detected prior to execution of any subsequent instruction.

Format:

15 8	7 4	3 0
00100011	RegB	RegA

DIVIDE UNSIGNED EXTENDED *DivUE*

DivUE [(RemR) | | (RegB)]/(RegA) → RegB, RemR

Divide the 64-bit unsigned integer formed by concatenating the contents of the Remainder Register with the contents of register **RegB** by the 32-bit unsigned integer in register **RegA**; store the quotient in register **RegB** and the remainder in the Remainder Register (special register 5). The Remainder Register provides dividend bits <63:32>, while register **RegB** provides dividend bits <31:0>.

Condition Codes: unchanged

Exceptions: arithmetic overflow

Notes: An arithmetic overflow exception occurs if the divisor is zero or if the result will not fit in the result register. If an overflow exception occurs and the Overflow Trap enable flag (PsR bit <1>) is "1", an Overflow Trap is generated and no result is stored; if the Overflow Trap enable flag is "0", no trap is generated and the result stored is unpredictable.

In Antares, **DivUE** is asynchronous and executes concurrently with subsequently-issued instructions. Overflow, if it occurs, will be detected prior to execution of any subsequent instruction.

Format:

15 8	7 4	3 0
00100111	RegB	RegA

MULTIPLY *Mul*

Mul (RegB)*(RegA) → RegB, ProdR

Multiply the contents of register **RegB** by the contents of register **RegA**; store product bits <63:32> in the Product Register (special register 4) and products bits <31:0> in register **RegB**. Both multiplicand and multiplier are treated as 32-bit signed integers.

Condition Codes: unchanged

Exceptions: none overflow

Notes: In Antares, **Mul** is asynchronous and executes concurrently with subsequently-issued instructions.

Format:

15 8	7 4	3 0
00110001	RegB	RegA

MULTIPLY UNSIGNED *MulU*

MulU (RegB)*(RegA) → RegB, ProdR

Multiply the contents of register **RegB** by the contents of register **RegA**; store product bits <63:32> in the Product Register (special register 4) and products bits <31:0> in register **RegB**. Both multiplicand and multiplier are treated as 32-bit unsigned integers.

Condition Codes: unchanged

Exceptions: none

Notes: In Antares, **MulU** is asynchronous and executes concurrently with subsequently-issued instructions.

Format:

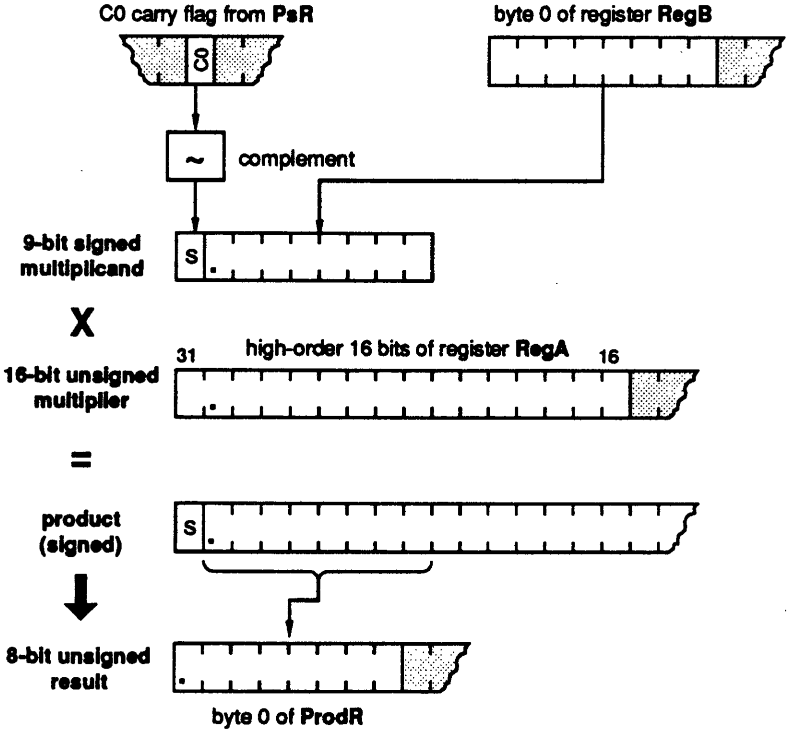
15 8	7 4	3 0
00100001	RegB	RegA

MULTIPLY PARTIAL MulP

MulP **(RegB)**<H/B> * **(RegA)** → **ProdR**

Form a signed multiplicand from each of the four bytes or two halfwords, as determined by the H/B mode flag (PsR bit <2>), in register **RegB**, using the complement of the carry condition code flag for that byte or halfword as its sign. Multiply each multiplicand by the multiplier in bits <31:16> of register **RegA**, and store the results in the corresponding byte or halfword positions of the Product Register. Bits <15:0> or register **RegA** are ignored. Each byte or halfword is independently multiplied.

Each multiplicand is a signed, two's-complement, left-justified, fractional quantity. The multiplier is an unsigned, left-justified, fractional quantity. Each result byte or halfword is an unsigned fractional quantity formed by discarding the sign of the product and storing as the result the most significant byte or halfword of the product. Multiplication of byte 0 via a **MulP** instruction with the H/B flag = "0" is diagrammed below. Multiplication of bytes 1-3, or of halfwords 0 and 1 when H/B = "1", is performed similarly.



Condition Codes: unchanged

Exceptions: none

Notes: In Antares, **MulP** is asynchronous and executes concurrently with subsequently-issued instructions. All four bytes or both halfwords are multiplied simultaneously.

Format:

15 8	7 4	3 0
00110101	RegB	RegA

MULTIPLY PARTIAL UNSIGNED *MulPU*

MulPU (RegB)<H/B> * (RegA) → ProdR

Multiply each of the four bytes or two halfwords, as determined by the H/B mode flag (PsR bit <2>), in register **RegB** by the multiplier in bits <31:16> of register **RegA**, and store the results in the corresponding byte or halfword positions of the Product Register. Bits <15:0> or register **RegA** are ignored. Each byte or halfword is independently multiplied.

Each multiplicand byte or halfword is an unsigned, left-justified, fractional quantity. and the multiplier is an unsigned, left-justified, fractional quantity. Each result byte or halfword also is an unsigned fractional quantity; only the most significant byte or halfword of the product is stored in ProdR

Condition Codes: unchanged

Exceptions: none

Notes: In Antares, **MulPU** is asynchronous and executes concurrently with subsequently-issued instructions. All four bytes or both halfwords are multiplied simultaneously.

Format:

15 8	7 4	3 0
00100101	RegB	RegA

NEGATE *Neg*

Neg **- (RegB) → RegA**

Store the two's complement of the value in register **RegA** in register **RegB**.

Condition Codes:
N— set to "1" if the result is negative; cleared to "0" otherwise
Z— set to "1" if the result equals zero; cleared to "0" otherwise
V— set to "1" if the result or source is – 2^{31}; cleared to "0" otherwise
C— C0 is set to "1" if the result is 0 and is cleared to "0" otherwise. C1, C2, and C3 are cleared to "0".

Exceptions: arithmetic overflow

Format:

15 8	7 4	3 0
00011111	RegB	RegA

8.7 Broadcast and Semaphore Instructions

Scorpius provides three classes of instructions for initiating and coordinating parallel activities: broadcast, inter-PU trap, and semaphore. Broadcast instructions are Start, Resume, Send, Receive, and Wait. The two inter-PU trap instructions, Preempt and Restart, are a broadcast variant and are described in Section 8.9. The semaphore instructions are Lock and Unlock.

Start, Resume, and Send instructions permit a PU to send an instruction address or a data value to other PUs in a single operation. The PUs receiving the address or data value are called the *targets* of the instruction, and are specified by a 4-bit PU Mask field in the instruction. This field has the form $b_3b_2b_1b_0$, where b_i is "0" if PU i is a target of the instruction and "1" otherwise. In these three instructions, the PU Mask bit corresponding to the PU issuing the instruction is ignored (a PU cannot send itself an address or a data value). The Wait instruction also uses the PU Mask field; if the PU Mask bit corresponding to the PU issuing the Wait instruction is "0", the instruction performs a halt operation. Inter-PU trap instructions use the PU Mask field to specify which PUs are to be preempted or restarted.. The result of issuing a broadcast instruction with no target PUs specified (PU Mask = 1111B) is unpredictable.

Broadcast, inter-PU trap, and semaphore instructions are discussed at length in Chapter 5.

RECEIVE *Rcv*

Rcv **RegA**

Wait for another PU to execute a Send instruction with the receiving PU as a target, store the value sent by that PU in register **RegA**, and continue execution. Execution of the Send instruction may have been initiated prior to initiation of the **Rcv** instruction, in which case the value is stored and execution of the **Rcv** instruction completes without waiting.

Condition Codes: unchanged

Exceptions: none

Format:

15 4	3 0
000100000111	RegA

RESUME *Rsm*

Rsm **PUMask**

Cause each target PU of the **Rsm** instruction, if halted, to resume execution at the address in the target's Current PC. Execution of the **Rsm** instruction does not complete until all target PUs have halted (if not initially halted) and had their execution resumed. If the PU issuing the **Rsm** is in user mode, all target PUs must halt in user mode before the **Rsm** instruction can complete. If the issuing PU is in system mode, target PUs may halt in either mode (and will have their execution resumed). A PU is specified as a target PU by setting the corresponding bit in the instruction's **PUMask** field to "0". Clear the PU Available flag in the **PsR** (**PsR** bit <0>).

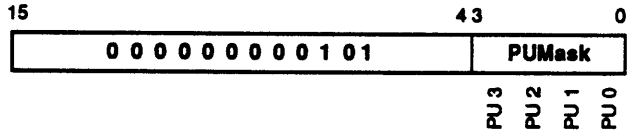
Condition Codes: unchanged

Exceptions: none

Notes: The **PUMask** field bit corresponding to the issuing PU is ignored. Issuing an **Rsm** instruction with no target PUs specified (**PUMask** = "1111B") causes unpredictable results.

In Antares, a synchronization operation is implicit in **Rsm** instruction execution: all target PUs must simultaneously be halted in the appropriate mode before their execution is resumed and **Rsm** execution completes. This synchronization is an attribute of implementation, not architecture, and may not take place in all implementations. If synchronization prior to a **Rsm** is desired, it should be explicitly programmed (via a Wait instruction).

Format:



SEND Send

Send (RegB) → PUMask

Send the contents of register **RegB** to each target PU. Execution of the **Send** instruction does not complete until each target PU has executed a **Rcv** instruction in the same mode as the sender. A PU is specified as a target PU by setting the corresponding bit in the instruction's **PUMask** field to "0".

Condition Codes: unchanged

Exceptions: none

Notes: The **PUMask** field bit corresponding to the issuing PU is ignored. Issuing a **Send** instruction with no target PUs specified (**PUMask** = "1111B") causes unpredictable results.

In Antares, a synchronization operation is implicit in **Send** instruction execution: all target PUs must simultaneously be receiving in the appropriate mode before the data value is broadcast and **Send** execution completes. This synchronization is an attribute of implementation, not architecture, and may not take place in all implementations. If synchronization prior to a **Send** is desired, it should be explicitly programmed (via a Wait instruction).

Format:

15 8	7 4	3 0
00000001	RegB	PUMask
		PU3 PU2 PU1 PU0

START *Strt*

Start @RegB, PUMask

Send the address in register **RegB** to each halted target PU and cause the target PU to begin execution at that address. Execution of the **Strt** instruction does not complete until each target PU has halted in the same mode as the PU issuing the **Strt** instruction, received its new starting address, and begun execution. A PU is specified as a target PU by setting the corresponding bit in the instruction's **PUMask** field to "0". Clear the PU Available flag in the PsR (PsR bit <0>).

Condition Codes: unchanged

Exceptions: none

Notes: The **PUMask** field bit corresponding to the issuing PU is ignored. Issuing a **Strt** instruction with no target PUs specified (**PUMask** = "1111B") causes unpredictable results.

In Antares, a synchronization operation is implicit in **Strt** instruction execution: all target PUs must simultaneously be halted in the appropriate mode before the address is broadcast and **Strt** execution completes. This synchronization is an attribute of implementation, not architecture, and may not take place in all implementations. If synchronization prior to a **Strt** is desired, it should be explicitly programmed (via a Wait instruction).

Format:

15 8	7 4	3 0
00000010	RegB	PUMask
		PU 3 PU 2 PU 1 PU 0

WAIT *Wait*

Wait **PUMask**

Wait serves two functions: it can be used to halt PU execution, or it can be used to synchronize the activities of multiple PUs.

If the PU issuing the **Wait** is itself a target of the **Wait** (i.e., its own bit in the **PUMask** field is "0"), the remaining **PUMask** field bits are ignored, the PU's Halt flag (PsR bit <24>) is set to "1", and the PU halts. Execution on a halted PU can be reinitiated via a **RsM** or **Strt** instruction. A PU is specified as a target PU by setting the corresponding bit in the instruction's **PUMask** field to "0".

If the PU issuing the **Wait** instruction is not itself a target of the **Wait**, completion of the **Wait** instruction is delayed until all target PUs have halted in the same mode as the PU issuing the **Wait.** The Wait instruction then completes and the waiting PU continues execution.

Condition Codes: unchanged

Exceptions: none

Notes: Issuing a **Wait** instruction with no target PUs specified (**PUMask** = "1111B") causes unpredictable results.

Format:

15 … 4	3 … 0
0 0 0 1 0 0 0 0 0 1 0 1	PUMask
	PU 3 PU 2 PU 1 PU 0

LOCK *Lock*

Lock

If the semaphore in the GSR for the current mode is set to "1", it is cleared to "0" and **Lock** instruction execution completes. If the semaphore is "0", **Lock** instruction execution is blocked until the semaphore is set to "1" by another PU; the semaphore then is cleared to "0" and **Lock** instruction completes. GSR bit <31> is the user mode semaphore, and GSR bit <30> is the system mode semaphore. The U/S flag in the PsR determines the current mode.

Condition Codes: unchanged

Exceptions: none

Notes: The order in which PUs attempting to lock a locked semaphore receive service when that semaphore becomes unlocked is described in Section 5.4.

Format:

```
15                              0
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1
```

UNLOCK *Unlk*

Unlk

Set the semaphore in the GSR for the current mode to "1" (regardless of its current value). GSR bit <31> is the user mode semaphore, and GSR bit <30> is the system mode semaphore. The U/S flag in the PsR determines the current mode.

Condition Codes: unchanged

Exceptions: none

Notes: The PU unlocking a semaphore is not necessarily the PU which locked it.

Format:

```
15                              0
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
```

8.8 Cache Control Instructions

Cache control instructions are used to maintain coherence between the instruction and data caches of a single CPU and between caches of different CPUs in a multi-CPU system, to flush instruction and data caches when required (e.g., on an address space switch in Antares), and to improve performance by eliminating unnecessary transfers of lines to and from memory and by prefetching lines in advance of their use.

The Create Data Cache line instruction **CDC** is used to avoid transferring a line from memory into the data cache when the initial contents of that line are no longer needed (as when the line will be completely overwritten). The Flush Data Cache line, Invalidate Data Cache line, Update Data Cache line, and Validate Data Cache line instructions (**FDC**, **IDC**, **UDC**, and **VDC**) provide four different ways of disposing of a data cache line. All four do nothing if the addressed line is not in the cache. **FDC** and **UDC** both write the line to memory if it is modified; **FDC** then marks the cache line invalid, while **UDC** marks the cache line unmodified. **IDC** and **VDC** do not write the line to memory if it is modified; **IDC** marks the cache line invalid; **VDC** marks it unmodified. All these instructions specify their operand — a data cache line — via a byte address in a general register. This address can be prefixed; the prefix provides a signed word displacement from the register address. Bits <31:6> of the operand address identify the memory line; bits <5:0> are ignored.

The Invalidate Instruction Cache line instruction **IIC** marks the specified cache line invalid, while the Invalid Instruction Cache instruction **IICA** invalidates all instruction cache lines. The Read Data Tag by Index instruction **RDTX** returns the tag associated with each data cache line location. The Prefetch Data Cache line instruction **PDC** is used to prefetch a line from memory in advance of its use to reduce or eliminate miss delays.

CREATE DATA CACHE LINE *CDC*

CDC	**@RegA**	if **PV** = "0"
	@RegA + (**PfxR**)<<2	if **PV** = "1"

Examine the data cache set specified by the operand address. If the specified line already is in that set, complete execution. If the line is missing, select an invalid member of that set, if one exists, in which to create the new line. If the set does not contain an invalid line, select the set member marked least recently used, and write the corresponding line to memory if it is modified. Create the new cache line by extracting the tag from the operand address; mark it valid and unmodified. If the Prefix Valid flag is "0", the operand address is the address in register **RegA**. If the Prefix Valid flag is "1", the operand address is formed by shifting the contents of the Prefix Register left two place and adding the result to the address in register **RegA**. (The Prefix Register provides a word displacement from the register **RegA** address.) Clear the Prefix Valid flag

Condition Codes: unchanged

Exceptions: data page fault, data access privilege violation

Notes: In Antares, the newly-created line is marked most-recent-used.

Format:

15 4	3 0
0 0 0 1 0 0 0 0 1 0 0 0	RegA

FLUSH DATA CACHE LINE *FDC*

FDC	**@RegA**	if **PV** = "0"
	@RegA + **(PfxR)**<<2	if **PV** = "1"

If the line specified by the operand address is in the data cache, write it to memory if it is modified and mark the cache line location invalid. If the specified line is not in the data cache, this instruction has no effect. If the Prefix Valid flag is "0", the operand address is the address in register **RegA**. If the Prefix Valid flag is "1", the operand address is formed by shifting the contents of the Prefix Register left two place and adding the result to the address in register **RegA**. Clear the Prefix Valid flag.

Condition Codes: unchanged

Exceptions: data access privilege violation

Format:

15 4	3 0
0 0 0 1 0 0 0 0 1 1 0 1	RegA

INVALIDATE DATA CACHE LINE *IDC*

IDC	**@RegA**	if **PV** = "0"
	@RegA + **(PfxR)**<<2	if **PV** = "1"

If the line specified by the operand address is in the data cache, mark the cache line location invalid *without* writing the line to memory if it is modified. If the specified line is not in the data cache, this instruction has no effect. If the Prefix Valid flag is "0", the operand address is the address in register **RegA**. If the Prefix Valid flag is "1", the operand address is formed by shifting the contents of the Prefix Register left two place and adding the result to the address in register **RegA**. Clear the Prefix Valid flag.

Condition Codes: unchanged

Exceptions: data access privilege violation

Format:

15 4	3 0
0 0 0 1 0 0 0 0 1 1 0 0	RegA

INVALIDATE INSTRUCTION CACHE LINE *IIC*

IIC **@RegA**

If the line specified by the address in register **RegA** is in the instruction cache, mark the cache line location invalid. If the specified line is not in the data cache, this instruction has no effect.

Condition Codes: unchanged

Exceptions: none

Notes: An **IIC** instruction executed in user mode can invalidate a line belonging to a system page; this does not result in correctness or security problems, and does not cause an exception.

Format:

15 4	3 0
0 0 0 1 0 0 0 0 1 0 1 0	RegA

INVALIDATE INSTRUCTION CACHE *IICA*

IICA

Mark all instruction cache lines location invalid.

Condition Codes: unchanged

Exceptions: none

Notes: An **IICA** instruction executed in user mode can invalidate lines belonging to system pages; this does not result in correctness or security problems, and does not cause an exception.

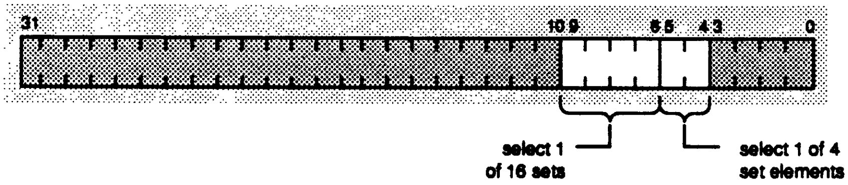
Format:

15 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1

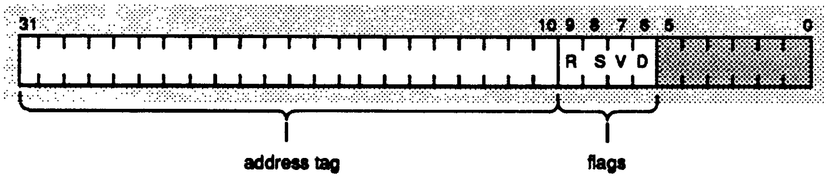
READ DATA TAG BY INDEX RDTX

RDTX @RegA → RegB

RDTX views the data cache as an array of 64 lines indexed 0-63. It specifies a line index as its operand address, and receives as its result the tag of the corresponding cache line location. The line index is contained in bits <9:4> of register **RegA**, as shown in (a) below; bits <9:6> specify one of the sixteen sets, while bits <5:4> select one of the four lines in the selected set. The result is returned in register **RegB**. The result register, shown in (b) below, contains the address tag (bits 31:10> of the virtual address of the line stored in the specified location) in bits <31:10> together with the read-only, system/user, valid, and modified flag bits from the tag in bits <9:6>. (The LRU bits are not returned.)



(a). Operand Address



(b). Result

The flags are interpreted as follows.

R read-only flag (register **RegB** bit <9>). This flag is inherited from the page table entry for the page in which the cache line is located; it is "1" if the page is read-only and "0" otherwise.

S system/user flag (register **RegB** bit <8>). This flag also is inherited from the page table entry for the page in which the cache line is located; it is "1" if the page can be accessed only system mode and "0" if the page can be accessed in both system and user modes.

V valid flag (register **RegB** bit <7>). This flag is "1" if the contents of the cache line location are valid and "0" otherwise.

D dirty flag (register **RegB** bit <6>). This flag is "1" if the line in this location has been modified since being moved in or since being marked unmodified by a UDC or VDC instruction.

The settings of the R, S, and D flags are valid only if V = "1".

Condition Codes: unchanged

Exceptions: operation fault

Notes: **RDTX** is a privileged instruction; attempted execution in user mode causes an operation fault.

This instruction is implementation-dependent; the preceding specification applies only to Antares. Other implementations of the Scorpius architecture may implement **RDTX** differently or not at all; see Section 6.5.

Format:

15 8	7 4	3 0
00010010	RegB	RegA

UPDATE DATA CACHE LINE *UDC*

UDC **@RegA** if **PV** = "0"
@RegA + **(PfxR)**<<2 if **PV** = "1"

If the line specified by the operand address is in the data cache and is modified, write it to memory and mark it unmodified. If the specified line is not in the data cache or is unmodified, this instruction has no effect. If the Prefix Valid flag is "0", the operand address is the address in register **RegA**. If the Prefix Valid flag is "1", the operand address is formed by shifting the contents of the Prefix Register left two place and adding the result to the address in register **RegA**. Clear the Prefix Valid flag.

Condition Codes: unchanged

Exceptions: data access privilege violation

Notes: In Antares, the LRU flags associated with the line are left unchanged.

Format:

15 4	3 0
000100001001	RegA

VALIDATE DATA CACHE LINE — *VDC*

VDC	**@RegA**	if **PV** = "0"
	@RegA + (**PfxR**)<<2	if **PV** = "1"

If the line specified by the operand address is in the data cache, and is modified, mark the cache line location unmodified *without* writing the line to memory. If the specified line is not in the data cache or is unmodified, this instruction has no effect. If the Prefix Valid flag is "0", the operand address is the address in register **RegA**. If the Prefix Valid flag is "1", the operand address is formed by shifting the contents of the Prefix Register left two place and adding the result to the address in register **RegA**. Clear the Prefix Valid flag.

Condition Codes: unchanged

Exceptions: data access privilege violation

Notes: In Antares, the LRU flags associated with the line are left unchanged.

Format:

15 4	3 0
0 0 0 1 0 0 0 0 1 0 0 0	RegA

8.9 Control and Miscellaneous Instructions

This section describes the Clear/Set Mode instructions (**ClrM/SetM**), which set and clear PsR flags, the inter-PU trap instructions (**Prmpt** and **Res**), the Return from Interrupt instruction (**RtI**), and the System Call instruction (**Trap**). The Test Mode (**TstM**) instruction, which tests the state of a PsR flag, is described in Section 8.3. The inter-PU trap instructions are a broadcast instruction variant and are discussed in Section 5.3. Trap and interrupt processing is discussed in Chapter 4; return from interrupt processing is discussed in Section 4.7.

CLEAR/SET MODE *ClrM/SetM*

ClrM No.

SetM No.

Clear PsR bit <**No.**> to "0" (**ClrM**) or set PsR bit <**No.**> to "1" (**SetM**). PsR bits are listed below.

0	PU Available flag
1	Overflow Trap Enable flag
2	Halfword/Byte Mode flag
3	Prefix Valid flag
4-11	reserved
12	PCQ Enable flag
13	User/System Mode flag
14	Taken Branch Trap Enable flag
15	PU Interrupt/Trap Enable flag

Bits <15:8> are privileged and can be tested only in system mode.

Condition Codes: unchanged

Exceptions: operation fault (on attempted access to bits <15:8> while in user mode

Notes: The result of attempting to set or clear a reserved PsR bit is unpredictable. PU state is unpredictable following execution of a **SetM** instruction which sets the PU Available flag, PsR bit <0> (see Section 4.6).

Format:

	15 4	3 0
ClrM	0 0 0 1 0 0 0 0 0 0 1 0	No.
SetM	0 0 0 1 0 0 0 0 0 0 1 1	No.
		b_3 b_2 b_1 b_0

PREEMPT Prmpt

Prmpt PUMask

Cause each target PU to generate a PU Preempt trap. If a target PU is interrupt/trap enabled (PsR bit <15> = "1"), it immediately recognizes the trap and transfers to the appropriate interrupt/trap entry address; otherwise, recognition is deferred until the PU becomes interrupt/trap enabled. Execution of the **Prmpt** instruction does not complete until each PU has recognized the trap. A PU is specified as a target PU by setting the corresponding bit in the instruction's **PUMask** field to "0".

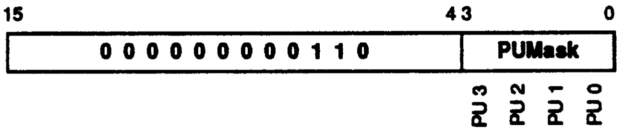
Condition Codes: unchanged

Exceptions: operation fault

Notes: **Prmpt** is a privileged instruction; attempted execution in user mode causes an operation fault. The PUMask field bit corresponding to the issuing PU is ignored. Issuing a **Prmpt** instruction with no target PUs specified (**PUMask** = "1111B") causes unpredictable results.

In Antares, trap generation (and recognition) does not take place until all PUs are *simultaneously* interrupt/trap enabled.

Format:



RESTART *Res*

Res **PUMask**

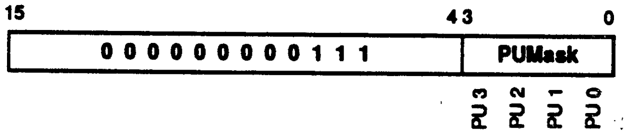
Cause each target PU, regardless of whether or not it is interrupt/trap enabled, to immediately generate and recognize a PU restart trap and transfer control to the corresponding interrupt/trap entry address. A PU is specified as a target PU by setting the corresponding bit in the instruction's **PUMask** field to "0".

Condition Codes: unchanged

Exceptions: operation fault

Notes: **Res** is a privileged instruction; attempted execution in user mode causes an operation fault. The PUMask field bit corresponding to the issuing PU is ignored. Issuing a **Res** instruction with no target PUs specified (**PUMask** = "1111B") causes unpredictable results.

Format:



RETURN FROM INTERRUPT *RtI*

RtI

RtI instructions are used to return from an interrupt or trap, and *must* be executed in pairs. An **RtI** instruction pair restores the Current PC and Next PC from the PCQ and (on the second **RtI**) restores the PsR from the SaveR. If the Halt flag in the restored PsR (bit <24>) is "1", PU execution halts following completion of the second **RtI** instruction; otherwise, execution continues with the instruction whose address is in Current PC.

Condition Codes: unchanged

Exceptions: operation fault

Notes: **RtI** is a privileged instruction; attempted execution in user mode causes an operation fault. The PU must be interrupt/trap disabled (PsR bit <15> = "0") when executing an **RtI** instruction pair, or the results are unpredictable. The result of attempting to execute a single **RtI** instruction is unpredictable.

In Antares, the result of an attempt to execute an **RtI** instruction immediately following a taken branch or a jump, or a **MovTS** access of the PCQ, is undefined.

Format:

```
15                              0
 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0
```

SYSTEM CALL Trap

Trap **No.**

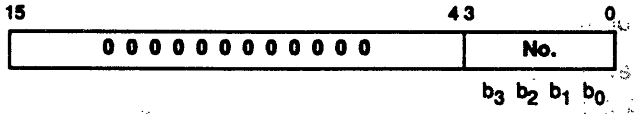
Store the trap number (**No.**) in bits <3:0> of the Trap Register and generate a System Call trap. The values stored in bits <18:4> are undefined. If the PU is interrupt/trap enabled, the trap is recognized and control transferred to the appropriate interrupt/trap entry address; otherwise, a PU Check Trap is generated and recognized.

Condition Codes: unchanged

Exceptions: PU Check

Notes: Trap generation, presentation, and recognition are discussed in Section 4.4. In Antares, instruction bits <15:8> are stored in TrapR bits <11:4> when the System Call trap is recognized.

Format:

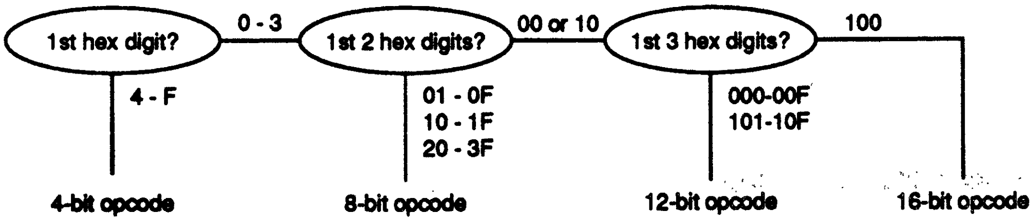


Appendix A. Instruction Formats & Operation Codes

This appendix provides a summary of Scorpius instruction formats and operation codes. All instructions are 16-bits in length; however, to minimize bandwidth, instructions are tightly encoded. Scorpius has a relatively large number of instruction formats, with operation codes varying in length from 3 bits to 16 bits.

Instruction formats are shown in Figures A.1 and A.2. Basic formats (Figure A.1) are instruction formats used by three or more instructions; unique formats (Figure A.2) are instruction formats used by only one or two instructions. Abbreviations are defined in Chapter 8. Note that formats include instructions with immediate and displacement fields of several lengths as well as instructions in which bits of a field are not contiguous. In most, but not all, cases, an immediate value of *I* or a displacement value of *D* is encoded as I - *1* or *D -1*.

Operation codes are listed in Figure A.3, grouped by length as 4-, 8-, 12, and 16-bit codes. While most operation codes are one of these lengths, there is one 3-bit operation code (**Bcc/JmP**) and three 7-bit operation codes (**Msk**, **ShL**, and **ShR**). The operation code length of an arbitrary instruction can be determined as shown below.



In this diagram, the hexadecimal digits of the operation code are examined from high-order to low-order; the "1st hex digit" corresponds to instruction halfword bits <15:12>.

An instruction whose first hexadecimal digit is C or D is a **Bcc** or a **Jmp** instruction, which are distinguished by examining instructions halfword bits <1:0>. Similarly, an instruction whose first two hexadecimal digits are 2A or 2B is a **Msk** instruction, an instruction whose first two hexadecimal digits are 2C or 2D is a **ShR** instruction, and an instruction whose first two hexadecimal digits are 3A or 3B is a **ShL** instruction.

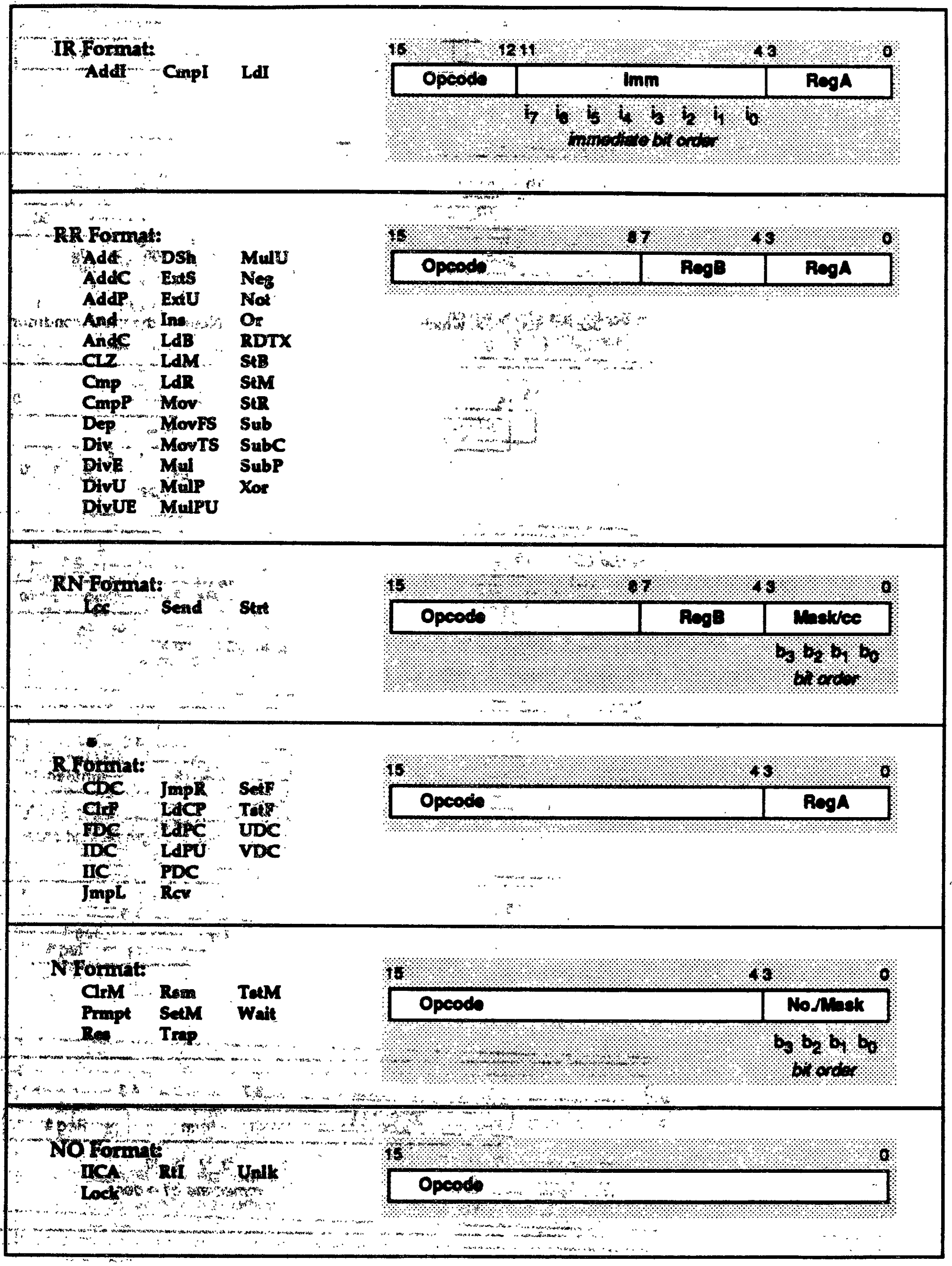
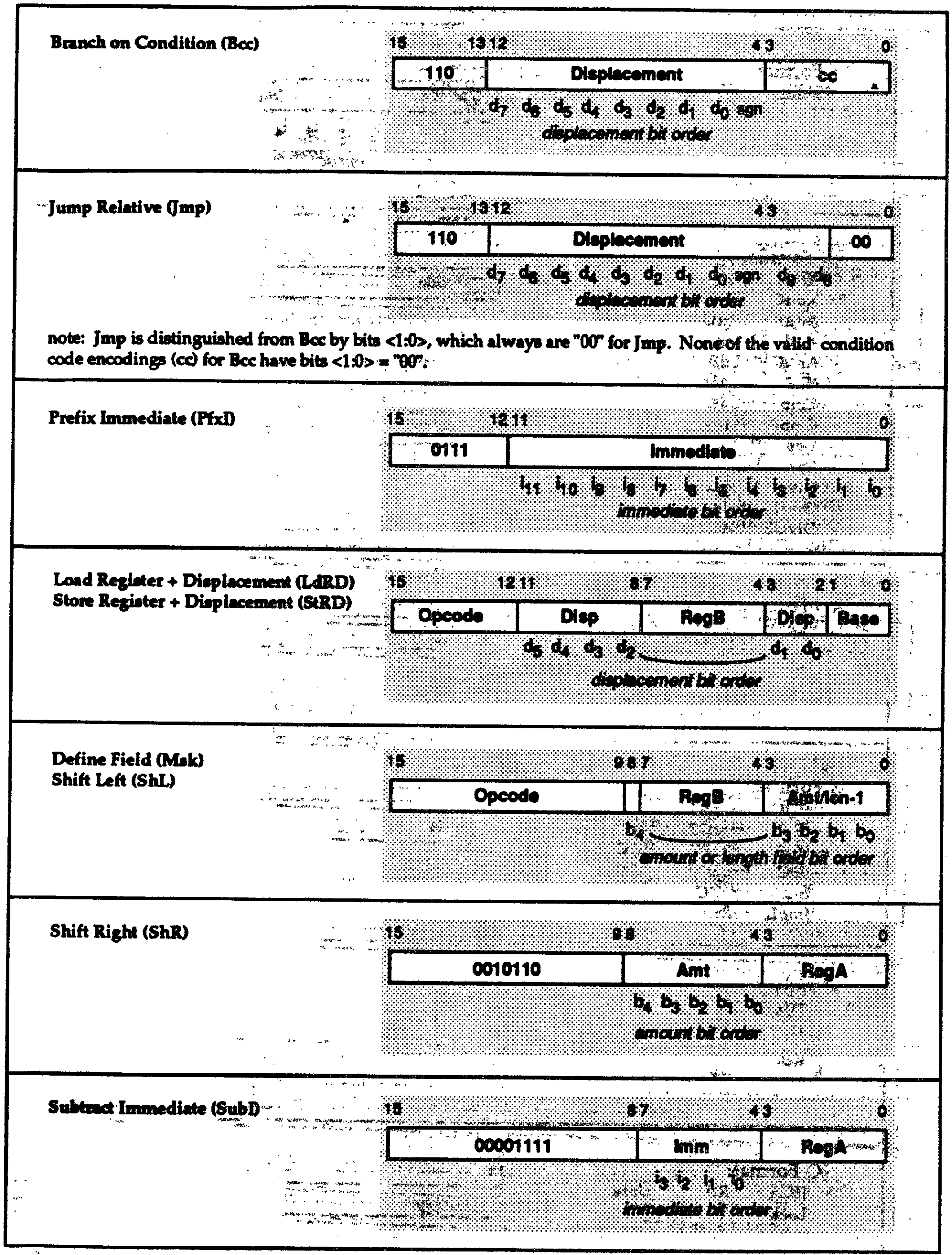


Figure A.1. Basic Formats



note: Jmp is distinguished from Bcc by bits <1:0>, which always are "00" for Jmp. None of the valid condition code encodings (cc) for Bcc have bits <1:0> = "00".

Figure A.2. Unique Formats

Four-Bit Operation Codes

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
0	note 1	4	AdI	8	StRD	C	Bcc/Jmp[2]
1		5	CmpI	9	LdRD	D	
2		6	LdI	A	undef.	E	undef.
3		7	PfxI	B	undef.	F	undef.

Eight-Bit Operation Codes

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
00	note 3	10	note 3	20	undef.	30	undef.
01	Send	11	Lcc	21	MulU	31	Mul
02	Strt	12	RDTX	22	undef.	32	undef.
03	MovTS	13	MovFS	23	DivU	33	Div
04	undef.	14	undef.	24	undef.	34	undef.
05	StR	15	LdR	25	MulFU	35	MulP
06	StB	16	LdB	26	undef.	36	undef.
07	StM	17	LdM	27	DivUE	37	DivE
08	Or	18	Cmp	28	undef.	38	undef.
09	Xor	19	CmpP	29	Dep	39	Ins
0A	And	1A	AndC	2A	Msk[4]	3A	ShL[4]
0B	Mov	1B	Not	2B		3B	
0C	Add	1C	Sub	2C	ShR[4]	3C	undef.
0D	AddP	1D	SubP	2D		3D	CLZ
0E	AddC	1E	SubC	2E	ExtS	3E	undef.
0F	SubI	1F	Neg	2F	ExtU	3F	Dsh

Twelve-Bit Operation Codes

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
000	Trap	008	undef.	100	note 5	108	VDC
001	TstF	009	undef.	101	TstM	109	UDC
002	LdCP	00A	JmpR	102	ClrM	10A	IIC
003	LdPU	00B	JmpL	103	SetM	10B	undef.
004	SetF	00C	undef.	104	ClrF	10C	IDC
005	Rsm	00D	undef.	105	Wait	10D	FDC
006	Prmpt	00E	undef.	106	LdPC	10E	CDC
007	Res	00F	undef.	107	Rcv	10F	PDC

Sixteen-Bit Operation Codes

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
1000	undef.	1004	undef.	1008	undef.	100C	undef.
1001	IICA	1005	undef.	1009	undef.	100D	undef.
1002	Unlk	1006	undef.	100A	undef.	100E	RtI
1003	Lock	1007	undef.	100B	undef.	100F	undef.

Notes:

1. Operation codes whose first (high-order) hexadecimal digit is 0 –3 are 8-, 12-, or 16-bit codes.
2. **Bcc** and **Jmp** can be viewed as a single instruction with a 3-bit operation code; the **Bcc** and **Jmp** instances of this instruction are distinguished by instruction bits <1:0>, which always are "00" for **Jmp** and "01" for **Bcc** .
3. Operation codes whose first two hexadecimal digits are 00 or 10 are 12- or 16-bit codes.
4. **Msk**, **ShR**, and **ShL** have 7-bit operation codes.
5. Operation codes whose first three hexadecimal digits are 100 are 16-bit codes.

Figure A.3. Operation Codes