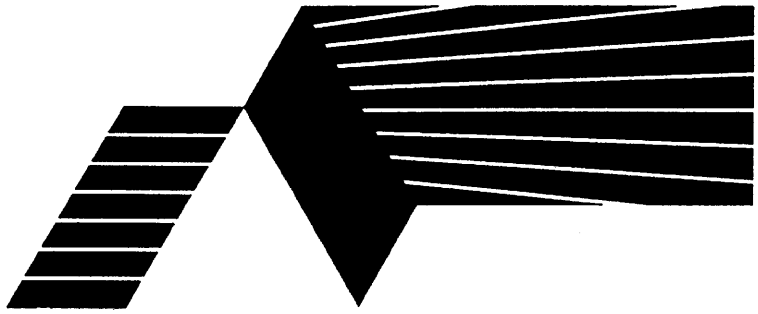
PRISM SYSTEMS



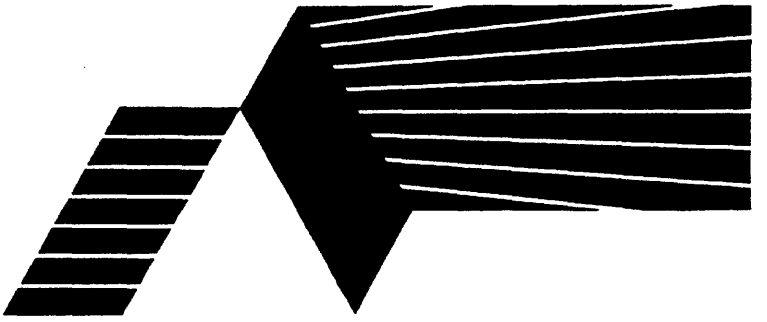
PRISM

System Reference Manual

digital™

Digital Equipment Corporation
Confidential and Proprietary

Restricted Distribution



PRISM

System Reference Manual

Digital Equipment Corporation
Confidential and Proprietary

digital™

This information shall not be disclosed to non-Digital Equipment Corporation personnel or generally distributed within Digital Equipment Corporation. Distribution is restricted to persons authorized and designated by the responsible Engineer or Manager.

This document shall not be left unattended, and, when not in use, shall be stored in a locked storage area.

These restrictions are to be enforced until noted otherwise.

Responsible Engineer/Manager *Date*

Revision No: 3.0 **Document Copy:**

Date: 26 April 1988

Restricted Distribution

CONTENTS

CHAPTER 4 INSTRUCTION DESCRIPTIONS

FIGURES

TABLES

PREFACE

Several competitors and new start-ups are introducing simplified architecture machines and are claiming superior price/performance over VAX. There are currently about a dozen such companies offering machines with RISC architecture (e.g. SUN, MIPS), vector processing (e.g., Convex, Scientific Computer Systems), symmetric multiprocessing (e.g., Encore, Sequent), and fine-grained parallel processing (e.g., Alliant) capabilities.

Most of these competitors are targeting the high end of the VAX market, which is our most profitable product space. Some are targeting the low end of our product family where simplified architectures offer cheaper and faster custom CMOS implementations than VAX.

Several advanced development and research projects within DIGITAL, and projects elsewhere in the computer industry, have produced results substantiating our competitor's claims and questioning the viability of the VAX architecture to sustain DIGITAL through the 1990's.

In response to this challenge, a strategic effort has been initiated within the company to define a new architecture that will complement our current VAX/VMS and VAX/ULTRIX offerings and provide DIGITAL with a competitive architecture through the 1990's and beyond.

The following lists summarize the assumptions, constraints, goals, and non-goals that have been set for the architecture.

Assumptions:

1. Simplified architectures show promise for reducing complexity while improving cost/performance and making higher absolute performance possible when compared with VAX.

2. Vector processing, multiprocessing, and parallel processing are well enough understood to make them a science (rather than a black art), and therefore, are essential to attaining a competitive architecture.

3. Neither DIGITAL nor its customers can afford the resources necessary to support an open architecture philosophy, but rather must be able to leverage software investments across an entire family of compatible products. This implies that any new architecture must be rigid and not allow the instruction set or privileged architecture to be changed from implementation to implementation.

4. The design work that must be performed is similar to the VAX architectural effort. An architectural document, at the same level of detail as produced for VAX, must be produced to guide implementations of the new architecture. It is required that this document receive wide review within the technical community and the company in general. When completed and accepted, the architecture will be placed under ECO control and managed by a central architecture group.

5. The architecture will be compatibly extended over time, and will allow subsets. Each extension will be subsettable and become a permanent part of the architecture which all implementations must adhere to. Features of the architecture that are subsetted in a particular implementation must be emulated transparently in software.

6. VAX compatibility is very important, especially with respect to the way memory is addressed and data is stored. This can be achieved with a combination of software and hardware rather than with just a hardware structure itself.

7. A VMS-like operating system environment can be constructed with a compatible file system, network, and user interface, and a functionally compatible set of system services.

8. ULTRIX will be ported to the new architecture and remain compatible with both the VAX and PDP-11 implementations. An ongoing effort will be made to ensure that all. implementations of ULTRIX remain compatible.

9. Any new architecture must fit into the DIGITAL computing environment and allow connection to local area networks, systems, and clusters.

10. The architecture will be extended in the future to accomodate a larger virtual address space.

Architectural Constraints:

1. The architecture must make it possible to efficiently support VAX data types. This support can be achieved with a combination of software and hardware.

2. The architecture must support VAX-compatible memory addressing.

3. The architecture must provide a VAX-compatible interlock capability so that it is possible to connect VAX processors and I/O peripherals to common memory systems.

4. The architecture must support the execution of identical program images on all implementations.

5. The scalar architecture must provide greater than a factor of two improvement in cost/performance over a VAX implementation using the same technology.

Architectural Goals:

1. To make it possible to build machines that are as good or better than the competition and have higher absolute performance limits than VAX.

2. To define an architecture that is inherently easier to implement than VAX and thus allows shorter development cycles, or alternatively, allows more effort to be expended on performance while holding the development cycle constant.

3. To make it attractive to implement the architecture without microcode.

4. To allow for easy pipelining and parallel instruction execution directly in the architecture, as opposed to esoteric implementation complexity to gain performance.

5. To provide integral vector processing capabilities.

6. To allow for symmetric multiprocessing as well as other forms of parallel processing.

7. To provide an extensible architecture with rules for subsettability.

8. To provide a corporate architecture for the 1990's that is more competitive than VAX and provides more inherent growth capability.

9. To remedy anticipated deficiencies and limitations in the VAX architecture (e.g., number of general registers, page size, physical address space, vector processing etc.).

10. To provide the functional capabilities of the VAX privileged architecture in a more simplified and easier-to-implement form.

11. To make it easy for customers to move applications to the new architecture from VAX.

12. To allow unprivileged VMS and ULTRIX layered products that are written in a higher-level language to be moved to the new architecture via recompilation, without loss of language semantics or file and data type compatibility.

13. To allow for the implementation of a security kernel.

Specific Non-Goals:

1. To include a VAX compatibility mode.
2. To support UNIBUS/QBUS/MASSBUS peripherals.
3. To translate VAX macrocode transparently and efficiently.
4. To address non-architectural issues such as the implementation of fault tolerant systems.
5. To support D_floating, H_floating, or decimal data types directly in hardware.
6. To support efficient handling of unaligned operands.

Revision History:

Revision 3.0, 26 April 1988

1. Minor changes.

Revision 2.0, 24 June 1986

1. No changes

Revision 1.0, 22 December 1985

1. General rewrite and rephrasing of the introduction, assumptions, architectural constraints, and architectural goals.

2. Dropped all references and comparisons with RISC architectures.

3. Added assumption that vector processing, multiprocessing, and parallelism are essential for a competitive new architecture.

4. Added the assumption that the architecture must allow for competitive and cost effective chip implementations.

5. Added a goal to provide integral vector processsing capabilities.

Revision 0.0, July 5, 1985

1. First review distribution.

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The difficulty in building cost-effective, high-performance VAX processors, and the competitive pressure due to recent architectural developments has motivated the design of the PRISM (Parallel Reduced Instruction Set Machine) architecture.

The following sections of this introduction describe:

1. Why building a high-end VAX is difficult.
2. An overview of the PRISM architecture.
3. The PRISM advantages and disadvantages.
4. The constraints and limitations of VAX compatibility on PRISM.
5. Terminology and conventions used in this document.

1.2 DIFFICULTIES IN BUILDING A HIGH PERFORMANCE VAX

VAX is a complex architecture with a large number of intra-instruction and inter-instruction conflicts, which make it difficult to build cost-effective implementations.

Intra-instruction conflicts, in both decode and execution, make pipelining techniques difficult to use. Some examples are:

- The variable instruction lengths and complex operand specifiers require a large amount of instruction decode and conflict-detection logic. VAX instructions can range from 1 byte to over 50 bytes in length, depending on the operand specifiers used.
- The side effects of autoincrement and autodecrement specifiers make pipelining, and the coordinated update of multiple register file copies, difficult.

- Specifying memory operand requests in the same instruction that operates on the data either degrades performance (because the execution unit must wait for the operand) or increases the cost to buffer the instruction and operands in order to pipeline the operation. Fetching a memory operand requires address calculation, address translation, and cache lookup. This will always be slower than reading a general register. VAX has insufficient registers in which to load memory operands prior to operating on the data; 16 are just not enough, especially when four are dedicated to fixed functions.

- The indirect specifiers require two memory references to fetch the operand, making the execution unit wait until the operand arrives. Alternatively, other architectures allow these two references to be separated and scheduled.

- Complex branch instructions, such as Branch on Bit (BBx) and Add Compare and Branch (ACBx), may require several memory references and execution cycles before the branch decision is known. These instructions also have the branch displacement at the end of the instruction requiring several cycles of specifier decode before the branch destination is known.

- Instructions like POPR and RSB have implied operands and implied register modification.

- The bit field instructions require special checks to determine whether the operand is in a register or memory and then additional checks to determine reserved operands.

- Compound instructions, such as CALL and POLY, encounter internal conflicts during execution where the hardware must stall because it has no other work to do. In addition, these instructions must read data operands to determine the semantics of the instruction.

Inter-instruction conflicts make parallel execution and out-of-order completion of VAX instructions very difficult. Some examples are:

- Virtually every instruction alters the condition codes, so the test or compare instruction can never be separated from the conditional branch instruction with intervening instructions. This means that in a pipelined implementation the conditional branch is stalled waiting for the condition codes from the immediately preceding instruction. Branch prediction could be implemented, but this further complicates the design and increases branch latency when the prediction is wrong.

- The register interlock and bypass logic is complicated by implied register operands, quadword and octaword register writes starting at an arbitrary register, and byte and word write merges into the general registers.

Most of the general functionality in the VAX architecture is infrequently used. Studies of operand specifier usage have shown that register, short literal, register deferred, and displacement mode operand specifiers constitute 85% to 95% of all operand specifiers used. The bit field instructions can take arbitrary specifiers for

the size and position operands, but in one study over 90% of the size and position specifiers were short literals. \

1.3 PRISM ARCHITECTURE OVERVIEW

The design of the PRISM architecture was guided by:

- The cost/performance and higher absolute performance advantages of simplified instruction set architectures.
- Advances in compiler technology. In particular, the ability to compile procedures inline, better register allocation algorithms, and instruction scheduling.
- A processor organization model that allows parallel instruction execution and out-of-order instruction completion.
- The ability to implement both chip-level and high-end machines.
- The declining cost of memory.

PRISM has some of the characteristics of the so-called RISC architectures but a better comparison would be the CRAY machines. Below is a brief overview of the PRISM instruction set characteristics followed by a description of how a pipelined processor might be implemented.

1.3.1 Instruction Set Characteristics

- All instructions are 32 bits long and have a regular format.
- There are 64 scalar registers (R0 through R63), each 32 bits wide. R0 reads as zero and writes to R0 are ignored. R1 is the current stack pointer and is referred to as SP.
- There are 16 vector registers (V0 through V15), each containing 64 elements, 64 bits wide. There is a 7-bit Vector Length register (VL), a 7-bit Vector Count register (VC), and a 64-bit Vector Mask register (VM).
- All scalar data manipulation is between scalar registers, with up to two register source operands (one may be an 8-bit literal) and one register destination operand.
- All vector data manipulation instructions get their source operands from one or two vector registers and write their results to a destination vector register.
- All memory reference instructions are of the load/store type that move data between scalar or vector registers and memory.
- There are no branch condition codes. Branch instructions test a scalar register value which may be the result of a

previous compare.

- Integer and logical instructions operate on longwords.
- Floating-point instructions operate on G_floating and F_floating operands.

1.3.2 Pipelined Processor Model

The processor model that guided the architecture definition consists of multiple pipelined function units, each of which executes a class of instructions. For example, one function unit for the load/store instructions, one for shifts, one for floating add/subtract, one for integer and floating multiply, and one for integer and floating divide. The multiply and divide units may or may not be pipelined.

The following outline shows one way to organize a pipelined design of the PRISM architecture. It should be emphasized that this is only one model; other implementation models are also possible.

1. Instruction fetch - The instruction to execute is fetched from the instruction cache.

2. Instruction decode and issue - The instruction is broken down into its constituent parts and data-independent control and address signals are generated. Before an instruction can begin execution ("issue"), several constraints must be satisfied:

   - All source and destination registers for the instruction must be free, i.e., there must be no outstanding writes to a needed register from prior instructions.
   - The register write path must be available at the future cycle in which this instruction will store its result. Only one result can be stored into the registers per cycle. All instructions, with the exception of loads, have a fixed, data-independent execution time. Loads are predicted on the basis of cache hits.
   - The function unit to be used by the instruction during execution must be free. All units, with the exception of divide, are pipelined and can accept a new scalar instruction each machine cycle. The divide unit is iterative and will accept a new instruction when the previous divide instruction completes. A vector instruction reserves the function unit for the duration of the vector operation.

     When a memory load/store instruction experiences a cache miss, at some point the load/store unit busy flag will cause subsequent load/store instructions to hold-issue until the miss completes.

     When an instruction does issue, the destination register and write path cycle for the result are reserved.

3. Operand setup - All instruction-independent register addresses are generated, operands are read and latched, and data-dependent control signals are generated.

4. Instruction execution - The instruction operands and control signals are passed to a function unit for execution.

5. Result store - The result from the function unit is stored in the register files or the cache as necessary.

Although this list is sequential, the five activities can be pipelined. For instance, making control signals data-independent and instruction formats regular means that more instruction decode and operand access can be done in parallel, with less logic and greatly simplified control.

Once an instruction is issued, it may take multiple cycles before the result of the calculation is available. Meanwhile, in the next cycle the next instruction can be decoded and, if all its issue conditions are satisfied, it can be issued. Therefore, instructions are decoded and issued **in I-Stream order** but because of the varying execution times of different operations the results can be stored into the registers **out of I-Stream order**. This complicates exception handling and hardware retry of failing instructions; however, these are rare events and the substantial performance gain and hardware savings from out-of-order completion of compiler-scheduled code favors this trade-off.

The regular nature of the instruction set and implementation result in a simple set of rules that compilers can use to schedule instructions and thereby increase performance through parallel instruction execution.

1.4 ADVANTAGES AND DISADVANTAGES OF PRISM

The characteristics of the PRISM architecture will allow developers to build processors with substantially more performance than a VAX for the same hardware cost in the same technology. The reasons for this are:

1. Fixed-length, quickly decoded instructions.

2. 64 scalar registers to reduce memory references and provide more temporary registers for compiler instruction scheduling and procedure use.

3. Parallel instruction execution and out-of-order instruction completion.

4. No branch condition codes.

5. No complex compound instructions with internal data dependencies, e.g., CALL/RET, CASE, ACBx, INSV/EXTV, Decimal. Inline code for complex functions will be better than VAX microcode because:

   - A compiler can pick the best code based on the knowledge it has and can eliminate special checks, e.g., string

overlap, procedure entry mask, sign of ACBx loop increment, whether a bit field is in a register or memory.

- VAX microcode must maintain additional state so that in the event of an exception or interrupt it can either backup the instruction or save enough state to continue using First Part Done.

- VAX microcode must make many reserved operand checks that add overhead, e.g., size and position operands in bit field instructions with different checks depending on whether the bit field is in registers or memory.

6. No microcode is required for instruction decode or execution.

7. A small instruction set emphasizing high frequency operations. Far less logic is spent on functionality that does not contribute to performance.

8. A larger branch displacement (22 bits versus 8 bits on VAX) eliminates double branches for conditional branches.

9. A larger page size (8 Kbytes) improves Translation Buffer (TB) effectiveness and allows the cache and TB lookup to occur in parallel.

The disadvantages of the PRISM architecture are:

1. PRISM programs may require 2 to 3 times the code size (in bytes) over VAX with a corresponding increase in instruction stream bandwidth. However, this trade-off is preferred because instruction cache miss rates are low and it is easier to build more instruction stream bandwidth than massive parallel instruction stream decode.

2. The 8-Kbyte page size will result in more memory fragmentation. Declining memory costs will help offset this.

3. Unaligned references will be slower because they may be implemented by macrocode.

4. Context switch time will increase because of the additional scalar registers that must be saved and restored.

1.5 VAX COMPATIBILITY

The PRISM architecture was constrained in a number of ways to support our existing VAX customer base. The goal is to make it both possible and easier for a VAX customer to integrate PRISM with VAX and to move an application to PRISM rather than to a competitor's machine. This goal impacts both the architecture and the system software.

1. The architecture uses VAX data types and allows byte addressing of memory.

2. The PRISM language compilers will retain their VAX-specific language semantics, e.g., data types and parameter passing, thus allowing customers to recompile most VAX programs without alteration.

1.5.1 Compatibility Limitations

There are some compatibility limitations between PRISM and the VAX architecture that may require changes to some high-level language programs in order to run them on PRISM:

1. Floating-point arithmetic - There are no PRISM instructions to compute D_floating and H_floating results. These operations can be performed by software emulation.

   PRISM has neither VAX POLY nor EMOD instructions. These instructions keep extra guard bits.

2. Memory protection granularity - PRISM has a page size larger than VAX. Therefore, VAX programs which rely on 512-byte protection granularity will not work.

3. Exceptions - Instructions may have been executed after an instruction that signals an arithmetic exception. Exception handlers that assume no further instructions have been executed will not work without changes to make the exception precise.

4. Dynamic instruction creation - Programs which dynamically construct and execute VAX instruction sequences and/or calculate addresses or offsets based on the sizes of VAX instructions will not work.

5. Instruction atomicity - Programs that rely on the atomicity of VAX instructions may not work, e.g., a multi-threaded application (such as an AST routine) in which shared memory data is guaranteed to be in a consistent state only between VAX instructions with no other means of synchronization being used. Any uninterruptable VAX instruction which makes more than one memory reference, e.g., INCL mem or ADDL3 mem1,mem2,mem3, could be used in this way. On PRISM the operation would require multiple instructions and, depending on where a thread was interrupted, stale data could be used.

6. Data structures - Code that depends upon VAX architected data structures such as the VAX PSL or call frames will not work.

7. PRISM supports a multiprocessing model that is different from VAX. The ordering of writes to memory is not specified by the PRISM architecture except at interlock boundaries. This means that shared data must be accessed only after acquiring a semaphore variable with an interlocked operation.

8. The granularity of sharing in a multiprocessor system is a longword on PRISM and byte on VAX.

1.5.2 Why No VAX Compatibility Mode Is Provided

VAX compatibility mode is not provided in the PRISM architecture (in the same way that PDP-11 compatibility mode is provided on VAX) for the following reasons:

1. The complexity of the VAX architecture would make it very expensive and difficult to provide a VAX compatibility mode with reasonable performance. VAX requires complex instruction decode logic, special data path support, e.g., condition codes, different memory management, and a microcode control store. This would defeat the purpose of a simplified architecture.

2. The majority of applications are written in high-level languages and can be recompiled. If programs are not recompiled the performance gain from the additional PRISM scalar registers, vector registers and instruction scheduling is lost.

3. The desirable software goal is to network PRISM and VAX processors so customer applications on VAX systems can share data with applications on PRISM. Customers will already own VAX systems on which to run those applications that they don't wish to port to PRISM.

1.6 TERMINOLOGY AND CONVENTIONS

1.6.1 Numbering

All numbers are decimal unless otherwise indicated. Where there is ambiguity, numbers other than decimal are indicated with the name of the base following the number in parentheses, e.g., FF (hex).

1.6.2 UNPREDICTABLE And UNDEFINED

RESULTS specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.

OPERATIONS specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing, to stopping system operation. UNDEFINED operations must not cause the processor to hang, i.e., reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions.

Note the distinction between result and operation. Non-privileged software cannot invoke UNDEFINED operations.

1.6.3 Ranges And Extents

Ranges are specified by a pair of numbers separated by a ".." and are inclusive, e.g., a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets separated by a colon and are inclusive; e.g., bits <7:3> specify an extent of bits including bits 7, 6, 5, 4, and 3.

1.6.4 Must Be Zero (MBZ)

Fields specified as Must Be Zero (MBZ) must never be filled by software with a non-zero value. If the processor encounters a non-zero value in a field specified as MBZ, an Illegal Operand exception occurs. See Chapter 6, Exceptions and Interrupts, Section 6.4.4.

1.6.5 Read As Zero (RAZ)

Fields specified as Read As Zero (RAZ) return a zero when read.

1.6.6 Should Be Zero (SBZ)

Fields specified as Should Be Zero (SBZ) should be filled by software with a zero value. These fields may be used at some future time. Non-zero values in SBZ fields produce UNPREDICTABLE results.

1.6.7 Ignore (IGN)

Fields specified as Ignore (IGN) are ignored when written.

1.6.8 Figure Drawing Conventions

Figures which depict registers or memory follow the convention that increasing addresses run right to left and top to bottom.

NOTE

\A note on the manual format: At certain points in the manual, comments on why certain decisions were made, unresolved issues, etc., are between a pair of backslashes. These comments provide additional clarification and will be removed from externally distributed editions.\

Revision History:

Revision 3.0, 26 April 1988

1. Minor updates to reflect software strategy for PRISM.

Revision 2.0, 24 June 1986

1. Typographical corrections and clarifications.
2. Vector Length register changed from 6 to 7 bits.

Revision 1.0, 22 December 1985

1. Change register width from 64 bits to 32 bits.
2. Remove PC from scalar registers.
3. Specify R0 reads zero, writes are ignored.
4. Specify SP mapped to register R1.
5. Add vector registers.

Revision 0.0, 5 July 1985

1. First review distribution.

CHAPTER 2

BASIC ARCHITECTURE

2.1 ADDRESSING

The basic addressable unit in PRISM is the 8-bit byte. Virtual addresses are 32 bits long; hence, the virtual address space is 2**32 (approximately 4.3 billion) bytes. Virtual addresses as seen by the program are translated into physical memory addresses by the memory management mechanism described in Chapter 5, Memory Management.

2.2 DATA TYPES

2.2.1 Byte

A byte is eight contiguous bits starting on an addressable byte boundary. The bits are numbered from right to left, 0 through 7:

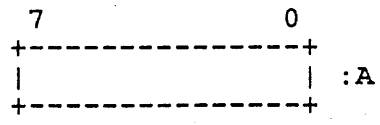


Figure 2-1: Byte Format

A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in PRISM by zero extended load and store instructions.

2.2.2 Word

A word is two contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 15:

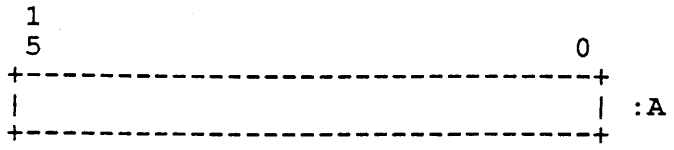


Figure 2-2: Word Format

A word is specified by its address A, the address of the byte containing bit 0. A word is a 16-bit value. The word is only supported in PRISM by zero extended load and store instructions.

NOTE

PRISM implementations are likely to impose a significant performance penalty on access to word operands that are not naturally aligned. (A naturally aligned word has zero as the low-order bit of its address.)

NOTE

\On many of the VAX implementations unaligned operands incurred approximately a 2x performance penalty, i.e., two memory references instead of one. It is expected that most PRISM implementations will implement unaligned accesses via software exceptions with the operating system providing emulation of the load or store of the unaligned data. The performance penalty may be expected to be up to 100x, depending on the particular implementation.\

2.2.3 Longword

A longword is four contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left 0 through 31:

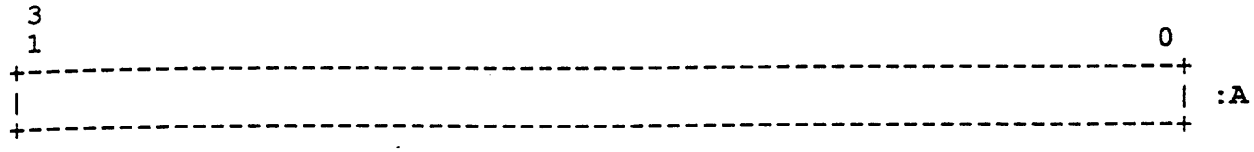


Figure 2-3: Longword Format

A longword is specified by its address A, the address of the byte containing bit 0. A longword is 32-bit value. When interpreted arithmetically, a longword is a two's complement integer with bits of increasing significance going 0 through 30. Bit 31 is the sign bit. The value of the integer is in the range -2,147,483,648..2,147,483,647. For performing addition, subtraction, multiplication, and comparison, PRISM instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits of increasing significance going 0 through 31. The value of the unsigned integer is in the range 0..4,294,967,295.

NOTE

> PRISM implementations are likely to impose a significant performance penalty when accessing longword operands that are not naturally aligned. (A naturally aligned longword has zero as the low-order two bits of its address.)

2.2.4 Quadword

A quadword is eight contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left 0 through 63:

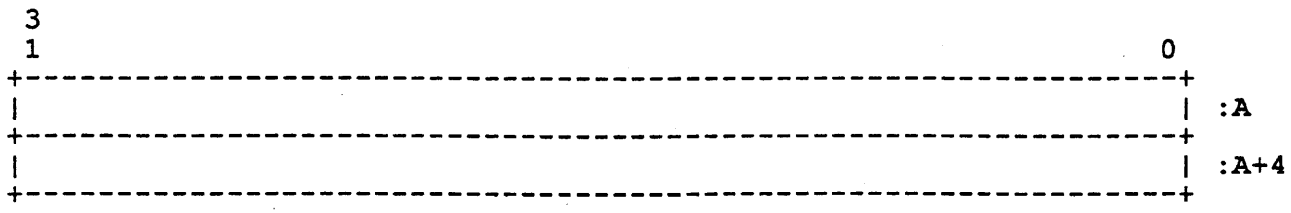


Figure 2-4: Quadword Format

A quadword is specified by its address A, the address of the byte containing bit 0. A quadword is a 64-bit value. The quadword is only supported in PRISM by load and store instructions.

NOTE

PRISM implementations are likely to impose a significant performance penalty when accessing quadword operands that are not naturally aligned. (A naturally aligned quadword has zero as the low-order three bits of its address.)

2.2.5 F_floating

An F_floating datum is four contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left 0 through 31.

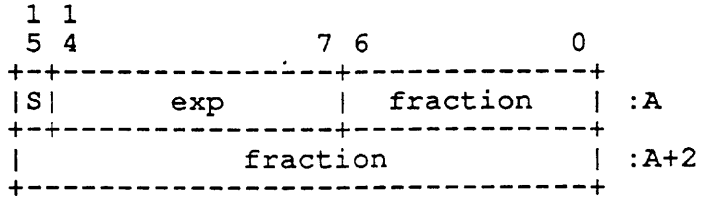


Figure 2-5: F_floating Format

An F_floating datum is specified by its address A, the address of the byte containing bit 0. The form of an F_floating datum is sign magnitude with bit 15 the sign bit, bits <14:7> an excess 128 binary exponent, and bits <6:0> and <31:16> a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the F_floating datum has a value of 0. If the result of a floating point instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..255 indicate true binary exponents of -127..127. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating-point instructions processing a reserved operand take an Arithmetic exception (see Chapter 6, Exceptions and Interrupts, Section 6.4.1). The value of an F_floating datum is in the approximate range 0.29*10**-38..1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, i.e., typically 7 decimal digits.

NOTE

PRISM implementations are likely to impose a significant performance penalty when accessing F_floating operands that are not naturally aligned. (A naturally aligned F_floating datum has zero as the low-order two bits of its address).

2.2.6 G_floating

A G_floating datum is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left 0 through 63:

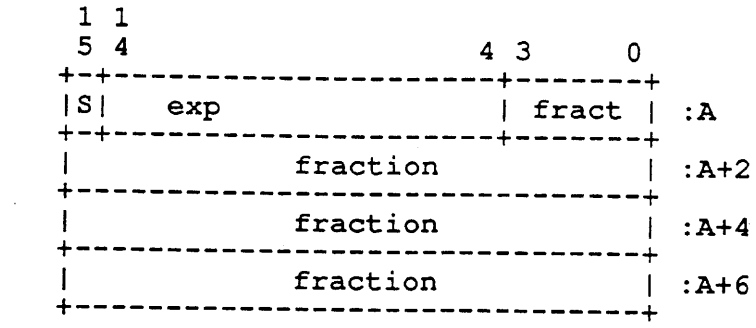


Figure 2-6: G_floating Format

A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits <14:4> an excess 1024 binary exponent, and bits <3:0> and <63:16> a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0. If the result of a floating point instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..2047 indicate true binary exponents of -1023..1023. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating-point instructions processing a reserved operand take an Arithmetic exception (see Chapter 6, Exceptions and Interrupts, Section 6.4.1). The value of a G_floating datum is in the approximate range 0.56*10**-308..0.9*10**308. The precision of a G_floating datum is approximately one part in 2**52, i.e., typically 15 decimal digits.

NOTE

PRISM implementations are likely to impose a significant performance penalty when accessing G_floating operands that are not naturally aligned. (A naturally aligned G_floating datum has zero as the low-order three bits of its address.)

2.2.7 Data Types With No Hardware Support

The following VAX data types are not directly supported in PRISM hardware, (see the **VAX Architecture Standard** for detailed information on these data types).

- Octaword
- D_floating
- H_floating
- Variable Length Bit Field
- Character String
- Trailing Numeric String
- Leading Separate Numeric String
- Packed Decimal String
- Queues

Revision History:

Revision 3.0, 26 April 1988

1. Dirty zero cannot be produced as a floating point result.

Revision 2.0, 24 June 1986

1. Minor edits.

Revision 1.0, December 22, 1985

1. Removed signed and unsigned descriptions for Byte, Word, and Quadword.
2. Changed formatting as per Rev 1.0 format.

Revision 0.0, July 5, 1985

1. First Review Distribution

CHAPTER 3

INSTRUCTION FORMATS

3.1 PRISM REGISTERS

3.1.1 Scalar Registers

There are 64 scalar registers (R0 through R63), each 32 bits wide. R1 is the stack pointer (SP).

When R0 is specified as a register source operand, a zero valued operand is supplied. When R0 is specified as a register destination, the result of the operation is discarded. If an exception is detected during the execution of an instruction that specifies R0 as the destination, it is UNPREDICTABLE whether or not the exception is actually signaled.

Some instructions read and write quadword register operands. Quadword register operands must be specified in even-odd register pairs. Bits <31:0> of the quadword are in the even register and bits <63:32> are in the odd register. If bit <0> of an instruction register field specifying a quadword operand is not 0, the result of the operation, including exception signaling, is UNPREDICTABLE.

When R0 is specified as a quadword source operand, bits <31:0> are zero and bits <63:32> are UNPREDICTABLE. When R0 is specified as a quadword destination, bits <31:0> are ignored (IGN) and bits <63:32> (the contents of R1) are UNPREDICTABLE.

3.1.2 Vector Registers

There are 16 vector registers, V0 through V15. Each vector register contains 64 elements numbered 0 through 63 and each element is 64 bits wide. A vector instruction that operates on longword or F_floating data reads bits <31:0> of each source element and writes a result in bits <31:0> of each destination element. Depending on the specific instruction, bits <63:32> of each destination element may receive bits <63:32> of one of the source operands, or may be UNPREDICTABLE.

If the same vector register is used as both a source and a destination in a Vector Gather (VGATH) instruction, the result of the operation is UNPREDICTABLE.

The 7-bit Vector Length register (VL) controls how many vector elements are processed. VL is loaded before executing a vector instruction. The value in VL may range from 0 to 64. A value greater

than 64 produces UNPREDICTABLE results. A value of zero means that no elements are processed. Once loaded, VL specifies the number of elements processed in all subsequent vector instructions until VL is loaded with a new value. Elements beyond VL in the destination vector register are not modified.

The Vector Mask register (VM) has 64 bits, each corresponding to an element in a vector register. Bit 0 corresponds to vector element 0.

The 7-bit Vector Count register (VC) receives the length of the offset vector generated by the IOTA instruction.

3.1.3 Program Counter

The Program Counter (PC) is a special register that addresses the instruction stream. As each instruction is decoded the PC is advanced to the beginning of the next sequential instruction. This is referred to as the "updated PC." Any instruction that uses the value of the PC will use the updated PC. The PC includes only bits <31:2> with bits <1:0> treated as RAZ/IGN. This quantity is a longword aligned byte address. The PC is not mapped to a scalar register, rather it is an implied operand on conditional branch and subroutine jump instructions.

3.1.4 Cycle Count Register

The Cycle Count register is a 64-bit register that counts processor cycles. Its resolution is within 128 cycles. It is saved and restored in the Hardware Process Control Block (see Chapter 7) by the Swap Process Context Instruction. It can be read with a Read Cycle Count Register instruction (see page 4-83.). It can be written in kernel mode by the Write Cycle Count Register instruction (see page 4-102.) When the 64-bit count overflows, the counter wraps around to zero.

3.2 NOTATION

The notation used to describe the operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax.

3.2.1 Scalar Operand Values

The notations Rav and Rbv are used to denote the values of the two scalar source operands, Ra and Rb.

Rav refers to the value of the Ra operand. This could be the contents of scalar register Ra or a zero extended 8-bit literal in the case of an Operate format instruction. If the instruction calls for a quadword operand then the contents of the even-odd register pair designated by Ra is used or again, a zero extended 8-bit literal may be specified.

Rbv refers to the value of the Rb operand. This is the contents of scalar register Rb. If the instruction calls for a quadword operand then the contents of the even-odd register pair designated by Rb is used.

Other Expression Operands:

IPR_x	Contents of Internal Processor Register x
PC	Updated PC value
PS	Processor Status
QRn	Quadword contents of even-odd scalar register n
Rn	Contents of scalar register n
Vn	Vector register n
X[m]	Element m of array X

3.2.2 Operators

The following operators are used:

!	Comment delimiter
+	Addition
-	Subtraction
*	Signed multiplication
*U	Unsigned multiplication
/	Division
<-	Replacement
||	Bit concatenation
{}	Indicates explicit operator precedence
(x)	Contents of memory location whose address is x
x<m:n>	Contents of bit field of x defined by bits n thru m
ACCESS(x,y)	Accessibility of the location whose address is x using the access mode y.
AND	Logical product
BIT_ROTATE(x,y)	Left circular shift of the first operand by the second operand

LEFT_SHIFT(x,y)	Logical left shift of first operand by the second operand
NOT	Logical (one's) complement
OR	Logical sum
RELATIONSHIP	
LT	Less than signed
LTU	Less than unsigned
LE	Less or equal signed
LEU	Less or equal unsigned
EQ	Equal signed and unsigned
NE	Not equal signed and unsigned
GE	Greater or equal signed
GEU	Greater or equal unsigned
GT	Greater signed
GTU	Greater unsigned
REM(x,y)	Remainder of x and y, such that x REM y has the same sign as the dividend x
ARITH_SHIFT(x,y)	Arithmetic shift right of first operand by the second operand
RIGHT_SHIFT(x,y)	Logical right shift of first operand by the second operand
SEXT(x)	X is sign extended to the required size
TEST(x)	Contents of register x tested for branch condition true
XOR	Logical difference
ZEXT(x)	X is zero extended to the required size

The following conventions are used:

1. Only operands appearing on the left-hand side of a replacement operator are modified.

2. No operator precedence is assumed other than that replacement (<-) has the lowest precedence. Explicit precedence is indicated by the use of "{}."

3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example, "+" applied to G_floating operands specifies a G_floating add while "+"

applied to longword operands indicates an integer add. Similarly, "LT" is a G_floating comparison when applied to G_floating operands and an integer comparison when applied to longword operands.

3.3 INSTRUCTION FORMATS

There are four basic PRISM instruction formats. They are:

1. Memory
2. Branch
3. Operate
4. Epicode

All instruction formats are 32 bits long with a 6-bit major opcode field in bits <31:26> of the instruction.

3.3.1 Memory Instruction Format

The Memory format is used to transfer data between scalar registers and memory, load an effective address, and for subroutine jumps. It has the following format:

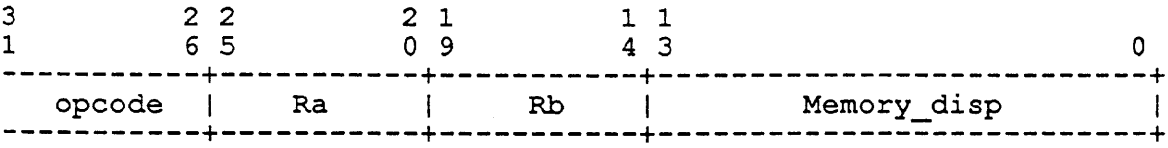


Figure 3-1: Memory Instruction Format

There is a 6-bit opcode field, two 6-bit register address fields, Ra and Rb, and a 14-bit signed displacement field.

The displacement field is a signed byte offset and is added to the contents of register Rb to form a virtual address. Overflow is ignored in this calculation.

The virtual address is used as a memory load/store address or a result value depending on the specific instruction. The virtual address (va) is computed as follows:

```
va <- Rbv + SEXT(Memory_disp)
```

3.3.2 Branch Instruction Format

The Branch format is used for the conditional branch instructions and

for the PC relative subroutine jumps. It has the following format:

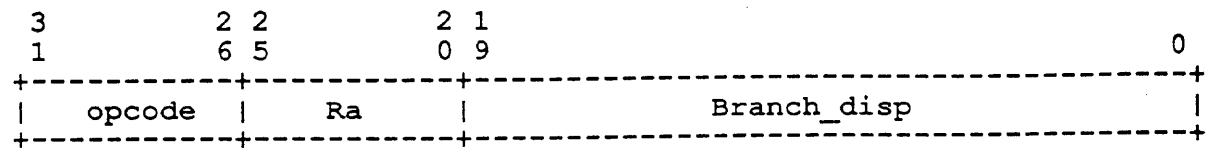


Figure 3-2: Branch Instruction Format

There is a 6-bit opcode field, one 6-bit register address field (Ra), and a 20-bit signed displacement field.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign extended to 32 bits and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as follows:

va <- PC + {4*SEXT(Branch_disp)}

3.3.3 Operate Instruction Format

The Operate format is used for instructions that perform scalar register-to-register operations and vector instructions. The Operate format allows the specification of one destination operand and two source operands. One of the source operands can be a literal constant. The Operate format is shown below for the two cases when bit <8> of the instruction, the Literal field (L), is 0 and 1.

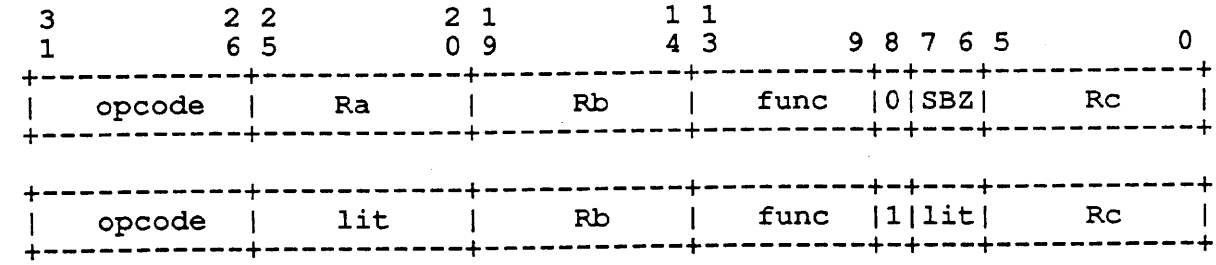


Figure 3-3: Operate Instruction Format

There is a 6-bit opcode field and a 5-bit function field (func). Unused function encodings produce UNPREDICTABLE but not UNDEFINED results; i.e., they are not security holes.

There are three operand fields, Ra, Rb, and Rc. Each operand field specifies either a scalar or vector operand as defined by the instruction. If a vector operand field contains a vector register number greater than 15, the result of the vector operation is UNPREDICTABLE. Note that vector register V0 can contain data, unlike scalar register R0.

The Ra field specifies a source operand. Scalar operands can specify

a literal or a scalar register using the literal control bit (L) in the instruction. Vector operands can specify a vector register only. The result of the vector operation is UNPREDICTABLE if a literal is specified for a vector operand.

If L is 0, the Ra field specifies a source register operand. Bits <7:6> of the instruction Should Be Zero.

If L is 1, an 8-bit zero extended literal constant is formed by combining the Ra field with bits <7:6> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero extended to 32 bits (64 bits for quadword operands). Symbolically, the scalar Rav operand is formed as follows,

```
IF L EQ 1 THEN
    Rav <- ZEXT(inst<25:20> || inst<7:6>)     !longword
    QRav <- ZEXT(inst<25:20> || inst<7:6>)    !quadword
ELSE
    BEGIN
    Rav <- Ra                                 !longword
    QRav <- QRa                               !quadword
    END
```

The Rb field specifies a source operand. Symbolically, the scalar Rbv operand is formed as follows,

```
Rbv <- Rb                   !longword
QRbv <- QRb                 !quadword
```

The Rc field specifies a destination operand.

Convert instructions use a subset of the Operate format and perform register-to-register conversion operations. The Ra operand specifies the source; the Rb field Should Be Zero.

3.3.3.1 Masked Vector Arithmetic Operate Instruction Format

The Masked Vector Arithmetic Operate format is used for instructions that perform register-to-register vector operations under the control of the Vector Mask register. This variant of the Operate format allows the specification of one destination operand and two source operands.

The Ra field specifies a source scalar or vector operand. The Rb field contains the Vb sub-field that specifies a source vector register and the Mask Selector (S) bit. The Rc field contains the Vc sub-field that specifies the destination vector register and the Masked Mode enable (M) bit.

The Masked Vector Arithmetic Operate format is shown below for the two cases when bit <8> of the instruction, the Literal field (L), is 0 and 1.

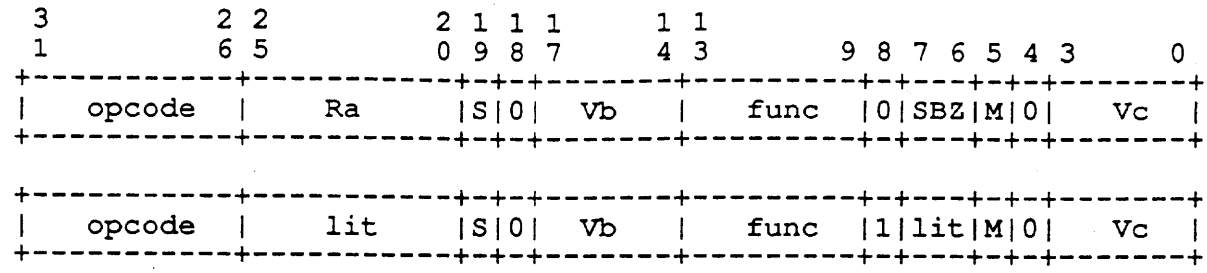


Figure 3-4: Masked Vector Arithmetic Operate Instruction Format

The M bit controls whether the vector mask register is used to select which vector elements are operated on. When M is one (zero) masked mode operation is enabled (disabled). When vector masked mode is enabled, the vector operation is performed on element i if the corresponding bit in the Vector Mask register (VM<i>) matches the value of the Mask Selector (S) bit. If VM<i> does not match S, then the element i vector result is not written and no exception on that result is signaled. When masked mode is disabled, all elements are operated upon. The result of the vector operation is UNPREDICTABLE if M is zero and S is one.

NOTE

The IOTA and VMERGE instructions also use the Masked Vector Arithmetic Operate format. However, only the S bit is used; the M bit Should Be Zero. Refer to Page 4-42 and 4-80 for a description of how the S bit is used. A /0 opcode qualifier specifies that the S bit is 0. A /1 opcode qualifier specifies that the S bit is 1. Opcode qualifiers are described in Section 4.1.4.

3.3.3.2 Masked Vector Memory Operate Instruction Format

The Masked Vector Memory Operate format is used for instructions that perform vector load or store operations under the control of the Vector Mask register. This variant of the Operate format allows the specification of one destination operand and two source operands.

The Ra field specifies a source scalar or vector operand. The Rb field specifies a source scalar or vector register. The Rc field contains the Vc sub-field that specifies the destination vector register and the Masked Mode enable (M) bit. Bit <12> of the instruction in the function field contains the Mask Selector (S) bit. Note that the location of the S bit for this format is different from that for the Masked Vector Arithmetic Operate format.

The Masked Vector Memory Operate format is shown below for the two cases when bit <8> of the instruction, the Literal field (L), is 0 and 1

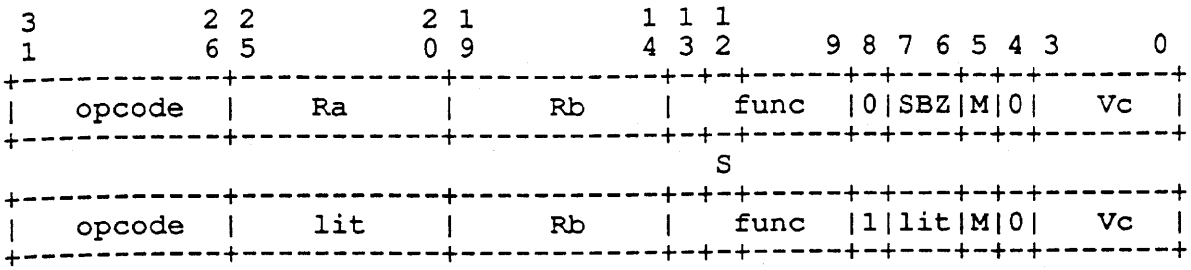


Figure 3-5: Masked Vector Memory Operate Instruction Format

The M bit controls whether the vector mask register is used to select which vector elements are operated on. When M is one (zero) masked mode operation is enabled (disabled). When vector masked mode is enabled, the vector operation is performed on element i if the corresponding bit in the Vector Mask register (VM<i>) matches the value of the Mask Selector (S) bit. If VM<i> does not match S, then the element i vector result is not written and no exception on that result is signaled. When masked mode is disabled, VL elements are operated upon. The result of the vector operation is UNPREDICTABLE if M is zero and S is one.

3.3.4 Epicode Instruction Format

The Extended Processor Instruction (Epicode) format is used to specify extended processor functions. It has the following format:

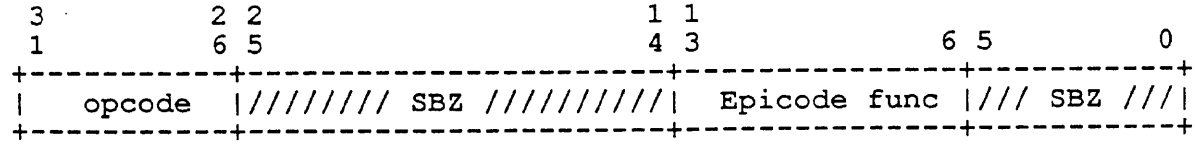


Figure 3-6: Epicode Instruction Format

The 8-bit Epicode function field specifies the operation.

The source and destination operands for Epicode instructions are supplied in fixed scalar registers that are specified in the individual instruction descriptions.

An opcode of zero and an Epicode function of zero specify the HALT instruction. An opcode of zero and Epicode function values between 64 and 127 specify implementation specific epicode instructions.

\The Epicode function field can be used to form a hardware dispatch address. The processor transfers control to a function specific Epicode routine. Many of the complex instructions that implement the privileged architecture, e.g., MxPR, REI, etc., are implemented as Epicode routines. In addition, memory management (TB fill) and hardware exception handling (Translation Not Valid fault, arithmetic trap) may be performed in Epicode. However, Epicode functions may be implemented in hardware.

Epicode instructions must drain the pipeline so that user exceptions resulting from prior instructions will not be reported after entering the Epicode routine. The signaling of user exceptions has priority over the execution of the Epicode instruction. See Chapter 10, Extended Processor Instruction Code, for more details.\

Revision History:

Revision 3.0, 26 April 1988

1. Document implementation specific epicode instructions.
2. Delete coprocessor support.
3. Vector instructions with longword results (except logicals) leave bits <63:32> of destination UNPREDICTABLE.
4. Add Masked Operate formats.

Revision 2.0, 24 June 1986

1. Minor clarifications.
2. Vector length register changed from 6 bits to 7 bits.
3. For vector instructions that produce 32-bit results, bits <63:32> of the destination vector register elements receive different values.

Revision 1.0, 22 December 1985

1. Change register width from 64 bits to 32 bits.
2. Remove PC from scalar registers.
3. Specify R0 reads zero, writes are ignored.
4. Specify SP mapped to register R1.
5. Defined quadwords in even-odd register pairs.
6. Renamed Move format to Memory format.
7. Changed Operate format to write Rc and use Ra field for literal.
8. Eliminated Operate format address calculation.
9. Eliminated JSR and Convert format descriptions.
10. Added vector registers, VM, VL, VC.
11. Added Coprocessor instruction format.

Revision 0.0, 5 July 1985

1. First review distribution.

CHAPTER 4

INSTRUCTION DESCRIPTIONS

4.1 INSTRUCTION SET OVERVIEW AND NOTATION

This chapter describes the instructions implemented by the PRISM architecture. The instruction set is divided into the following sections:

1. Memory Load and Store
2. Integer arithmetic
3. Logical and Shift
4. Floating-point arithmetic
5. Control
6. Miscellaneous
7. Privileged

Within each major section, closely related instructions are combined into groups and described together. The instruction group description is composed of the following:

- The group name.
- The format of each instruction in the group. This gives the name, access type, and data type of each instruction operand.
- The operation of the instruction.
- Exceptions specific to the instruction.
- The mnemonic and name of each instruction in the group.
- A description of the instruction operation.
- Programming examples and optional notes on the instruction.

4.1.1 Subsetting Rules

An instruction that is omitted in a subset implementation of the PRISM architecture is not performed in either hardware or Epicode. System software may provide emulation routines for subsetted instructions.

The following four groups of instructions may be omitted as a group in a subset implementation. Note, if one instruction in a group is provided then all other instructions in that group must be provided:

1. Integer multiplication.
2. F_ and G_floating.
3. Vectors.
4. Integer division.

The first three groups are hierarchical. If F_ and G_floating are supported, then integer multiplication must also be supported. If vectors are supported, then F_ and G_floating must be supported along with integer multiplication. The individual instruction descriptions indicate whether an instruction can be subsetted.

4.1.2 Vector Instructions

The PRISM architecture provides vector instructions for most arithmetic and data movement operations. There are 16 vector registers, each 64 elements long. All vector instructions use the Operate instruction format. Most vector instructions get their source operands from one or two vector registers and write their results to another vector register. There are also vector load and store instructions to move data between memory and the vector registers.

Generally, two variations of each vector instruction are provided. One variant operates on data from two vector registers and writes the result into a destination vector register. The other variant operates on data from a scalar register and a vector register, writing the result into a destination vector register.

The instruction descriptions distinguish the two variations by specifying in the first instruction operand position a vector operand (Va) or a scalar operand (Ra or a literal). This corresponds to the register field "Ra" in the Operate format instruction. The actual opcode assignment for each variation is different.

Vector instructions are only executed when Vector Enable (VEN) is set in the Processor Status (PS). If PS<VEN> is clear, a Vector Enable exception is generated when a vector instruction is executed. See Chapter 6, Exceptions and Interrupts, Sections 6.2 and 6.4.4.3.

4.1.3 Instruction Operand Notation

The notation used to describe instruction operands follows from the operand specifier notation used in the **VAX Architecture Standard**. Instruction operands are described as follows:

<name>.<access type><data type>

where:

1. Name specifies the instruction field (Ra, Rb, Rc, or disp) and register type of the operand (scalar or vector). It can be one of the following:

   - o disp - The displacement field of the instruction.
   - o Ra - A scalar register operand in the Ra field of the instruction.
   - o #a - A scalar literal operand in the Ra field of the instruction.
   - o Rb - A scalar register operand in the Rb field of the instruction.
   - o Rc - A scalar register operand in the Rc field of the instruction.
   - o Va - A vector register operand in the Ra field of the instruction.
   - o Vb - A vector register operand in the Rb field of the instruction.
   - o Vc - A vector register operand in the Rc field of the instruction.

2. Access type is a letter denoting the operand access type:

   - o a - The operand is used in an address calculation to form an effective address. The data type code that follows indicates the units of addressability (or scale factor) applied to this operand when the instruction is decoded, e.g., ".al" means scale by 4 (longwords) to get byte units (used in branch displacements), ".ab" means the operand is already in byte units (used in load/store instructions).
   - o i - The operand is an 8-bit immediate literal in the instruction.
   - o r - The operand is read only.
   - o w - The operand is write only.

3. Data type is a letter denoting the data type of the operand:

   - o b - Byte
   - o f - F_floating
   - o g - G_floating

- o l - Longword
- o q - Quadword
- o w - Word
- o x - The data type is specified by the instruction

Quadword and G_floating data that are in scalar registers must be in even-odd register pairs. The even register number should be specified in the instruction register fields.

4.1.4 Opcode Qualifiers

Some operate format scalar and vector instructions have several variants. For example, Add F_floating (ADDF) is supported with and without floating underflow enabled, and with either chopped or VAX rounding. The different variants of such instructions are denoted by opcode qualifiers, which consist of a slash (/) followed by a string of selected qualifiers. Each qualifier is denoted by a single character as shown below:

- o C - Chopped Rounding
- o U - Floating Underflow Enable
- o V - Integer Overflow Enable
- o {0,1} - Enable Masked Operation and operate on elements for which vector mask register bit <i> matches this qualifier. This qualifier is used for instructions that use the Masked Vector Arithmetic Operate and Masked Vector Memory Operate formats. When this qualifier is used, the Masked Mode (M) bit is set (except for VMERGE and IOTA) and the Mask Selector (S) bit takes on the value of the qualifier. See Sections 3.3.3.1 and 3.3.3.2. The VMERGE and IOTA use this qualifier to specify the Mask Selector bit, but do not use Masked Mode (see instruction descriptions on Page 4-42 and 4-80).
- o W - Write Intent. This qualifier is used by the VGATH and VLD instructions. It indicates that the data being read might be written in the near future. This feature can be used to optimize write-back cache performance in a multiprocessor system. Memory management does not check for write accessibility.

The default values are VAX Rounding, Floating Underflow Disabled, Integer Overflow Disabled, and Masked Operation Disabled.

4.2 MEMORY LOAD/STORE INSTRUCTIONS

The instructions in this section move data between the scalar registers and memory, move data between the vector registers and memory, and perform interlocked operations on shared memory data.

They use the Memory and Epicode instruction formats. The instructions are summarized below:

Mnemonic	Operation
CMPSWLI	Compare and Swap Longword, Interlocked
CMPSWQI	Compare and Swap Quadword, Interlocked
LDA	Load Address
LDB	Load Zero Extended Byte
LDW	Load Zero Extended Word
LDL	Load Longword
LDQ	Load Quadword
RMALI	Read, Mask, Add Longword, Interlocked
RMAQI	Read, Mask, Add Quadword, Interlocked
STB	Store Byte
STW	Store Word
STL	Store Longword
STQ	Store Quadword
VGATHL	Vector Gather Longword
VGATHQ	Vector Gather Quadword
VLDL	Vector Load Longword
VLDQ	Vector Load Quadword
VSCATL	Vector Scatter Longword
VSCATQ	Vector Scatter Quadword
VSTL	Vector Store Longword
VSTQ	Vector Store Quadword

If Vector Load, Store, Scatter, or Gather are used to access I/O space, the results are UNPREDICTABLE.

An implementation may allow scalar and vector memory references to proceed concurrently on the same processor. Software is responsible for determining when read/write memory data conflicts between scalar and vector references might produce incorrect results. If such conflicts exist, software must insert DRAINM instructions (see Page 4-77) to ensure correct operation. Likewise, software is also responsible for determining when read/write memory data conflicts between multiple vector references might produce incorrect results. If such conflicts exist, software must insert DRAINV instructions (see Chapter 9) to ensure correct operation.

Compare and Swap Longword, Interlocked

Format: Epicode format

```
CMPSWLI
```

Operation:

```
! R4 contains address of comparand 1
! R5 contains comparand 2
! R6 contains swap value
! R4<0> receives the swap status

addr <- R4
IF addr<1:0> NE 0 THEN
    {Illegal Operand exception}

{check for ACV, FOR, FOW, TNV and take Memory Management exceptior

tmp <- (addr){interlocked}      !acquire hardware interlock.

IF tmp EQ R5 THEN
    BEGIN
    (addr){interlocked} <- R6   !release hardware interlock
    R4 <- R4 OR 1               !set successful compare status
    END
ELSE
    BEGIN
    (addr){interlocked} <- tmp  !release hardware interlock
    R5 <- tmp
    END
END
```

Exceptions:

```
Access Violation
Fault On Read
Fault On Write
Illegal Operand
Translation Not Valid
```

Opcodes:

```
CMPSWLI Compare and Swap Longword, Interlocked
```

Description:

The longword aligned memory operand, whose virtual address is in R4, is fetched and compared to R5. If the two operands are equal, R6 is written to the memory location and a swap status flag (R4<0>) is set. If the two operands are not equal, the original memory operand is written into R5 and the memory location is left unchanged.

This instruction performs an interlocked memory access in that no other processor or I/O device can perform an interlocked operation on the same operand until the current interlocked operation has completed.

If the operand address in R4 is not longword aligned, an Illegal Operand exception is signaled. The operation is UNPREDICTABLE if

CMPSWLI accesses I/O space. If both Fault On Read and Fault On Write conditions exist, it is UNPREDICTABLE which is taken.

To make an insertion in an interlocked LIFO queue, the following instructions would be executed:

```
        lda     head,r4                 ; get address of queue header
        ldl     (r4),r5                 ; get address of first entry
        lda     entry,r6                ; get address of queue entry
10$:    stl     r5,(r6)                 ; link old first to new first
        cmpswli                         ; compare and swap longword
        blbc    r4,10$                  ; if lbc, repeat operation
```

To remove the entire list of queue elements, the following instructions would be used:

```
        lda     head,r4                 ; get address of queue header
        ldl     (r4),r5                 ; get address of first entry
        or      r0,r0,r6                ; set address of new queue entry
10$:    cmpswli                         ; compare and swap longword
        blbc    r4,10$                  ; if lbc, repeat operation
```

If it is desirable to remove a single entry from the front of a queue, then CMPSWQI must be used and a sequence number must be included to avoid the problem of reading the address of the first entry in the queue, reading its link to the next entry, and then getting interrupted in such a way as to allow another reader of the queue to remove the first two entries in the queue and then insert an entry in the queue which has the same address as the previous first entry in the queue.

Compare and Swap Quadword, Interlocked

Format: Epicode format

```
CMPSWQI
```

Operation:

```
! R4 contains address of comparand 1
! QR6 contains comparand 2
! QR8 contains swap value
! R4<0> receives the swap status

addr <- R4
IF addr<2:0> NE 0 THEN
    {Illegal Operand exception}

{check for ACV, FOR, FOW, TNV and take Memory Management exception}

tmp <- (addr){interlocked}        !acquire hardware interlock.

IF tmp EQ QR6 THEN
    BEGIN
    (addr){interlocked} <- QR8    !release hardware interlock
    R4 <- R4 OR 1                 !set successful compare status
    END
ELSE
    BEGIN
    (addr){interlocked} <- tmp    !release hardware interlock
    QR6 <- tmp
    END
END
```

Exceptions:

```
Access Violation
Fault On Read
Fault On Write
Illegal Operand
Translation Not Valid
```

Opcodes:

```
CMPSWQI Compare and Swap Quadword, Interlocked
```

Description:

The quadword aligned memory operand, whose virtual address is in R4, is fetched and compared to QR6. If the two operands are equal, QR8 is written to the memory location and a swap status flag (R4<0>) is set. If the two operands are not equal, the original memory operand is written into QR6 and the memory location is left unchanged.

This instruction performs an interlocked memory access in that no other processor or I/O device can perform an interlocked operation on the same operand until the current interlocked operation has completed.

If the operand address in R4 is not quadword aligned an Illegal Operand exception is signaled. The operation is UNPREDICTABLE if CMPSWQI accesses I/O space. If both Fault On Read and Fault On Write conditions exist, it is UNPREDICTABLE which is taken.

Load Address

Format: Memory format

```
LDA     disp.ab(Rb.ab),Ra.wl
```

Operation:

```
Ra <- Rbv + SEXT(disp)
```

Exceptions:

None

Opcodes:

```
LDA     Load Address
```

Description:

The virtual address is computed by adding register Rb to the sign extended 14-bit displacement. The 32-bit result is written to register Ra.

When Rb is R0 the signed 14-bit displacement is written to register Ra.

Load **Memory Data into Scalar Register**

Format: **Memory format**

```
LD      disp.ab(Rb.ab),Ra.wx
```

Operation:

```
va <- Rbv + SEXT(disp)

Ra  <- ZEXT((va)<7:0>)          !LDB
Ra  <- ZEXT((va)<15:0>)         !LDW
Ra  <- (va)<31:0>               !LDL
QRa <- (va)<63:0>               !LDQ
```

Exceptions:

Access Violation
Fault On Read
Scalar Alignment
Translation Not Valid

Opcodes :

LDB	Load Zero Extended Byte from Memory to Register
LDW	Load Zero Extended Word from Memory to Register
LDL	Load Longword from Memory to Register
LDQ	Load Quadword from Memory to Register Pair

Description :

The **virtual** address is computed by adding register Rb to the sign extended 14-bit displacement. The source operand is fetched from memory zero extended to a longword for LDB and LDW, and written to register Ra.

LDQ **fetches** a quadword from memory and writes it to the even-odd pair specified by Ra.

Software Note:

In Some **implementations** these instructions may be emulated if the memory operand is not naturally aligned. This could be on the order of 100 **times** slower. Consequently, when compilers can detect this (e.g. a field in a packed record), they should use an inline multi- **instruction** sequence to fetch the operand in pieces rather than incur **the** emulation overhead.

Read, Mask, Add Longword Interlocked

Format: Epicode format

```
RMALI
```

Operation:

```
! R4 contains the longword aligned virtual address
! R5 contains the longword mask data
! R6 contains the longword addend data
! R4 receives the longword read data

addr <- R4
IF addr<1:0> NE 0 THEN
    {Illegal Operand exception}

{check for ACV, FOR, FOW, TNV and take Memory Management exception}

R4 <- (addr){interlocked}        !acquire hardware interlock.

(addr){interlocked} <- {R4 AND R5} + R6
                                 !release hardware interlock
```

Exceptions:

```
Access Violation
Fault On Read
Fault On Write
Illegal Operand
Translation Not Valid
```

Opcodes:

```
RMALI   Read, Mask, Add Longword, Interlocked
```

Description:

The longword aligned memory operand, whose virtual address is in R4, is fetched and written to R4. The memory operand is ANDed with the mask in R5 and then added to the addend data in R6. The result is then written to the original memory location.

This instruction performs an interlocked memory access in that no other processor or I/O device can perform an interlocked operation on the same operand until the current interlocked operation has completed.

If the operand address in R4 is not longword aligned an Illegal Operand exception is signaled. The operation is UNPREDICTABLE if RMALI accesses I/O space. If both Fault On Read and Fault On Write conditions exist, it is UNPREDICTABLE which is taken.

Read, Mask, Add Quadword Interlocked

Format: Epicode format

```
RMAQI
```

Operation:

```
! R4 contains the quadword aligned virtual address
! QR6 contains the quadword mask data
! QR8 contains the quadword addend data
! QR4 receives the quadword read data

addr <- R4
IF addr<2:0> NE 0 THEN
    {Illegal Operand exception}

{check for ACV, FOR, FOW, TNV and take Memory Management exception}

QR4 <- (addr){interlocked}          !acquire hardware interlock.

(addr){interlocked} <- {QR4 AND QR6} + QR8
                                    !release hardware interlock
```

Exceptions:

```
Access Violation
Fault On Read
Fault On Write
Illegal Operand
Translation Not Valid
```

Opcodes:

```
RMAQI   Read, Mask, Add Quadword, Interlocked
```

Description:

The quadword aligned memory operand, whose virtual address is in R4, is fetched and written to QR4. The memory operand is ANDed with the mask in QR6 and then added to the addend data in QR8. The result is then written to the original memory location.

This instruction performs an interlocked memory access in that no other processor or I/O device can perform an interlocked operation on the same operand until the current interlocked operation has completed.

If the operand address in R4 is not quadword aligned an Illegal Operand exception is signaled. The operation is UNPREDICTABLE if RMAQI accesses I/O space. If both Fault On Read and Fault On Write conditions exist, it is UNPREDICTABLE which is taken.

Store Scalar Register Data into Memory

Format: Memory format

```
ST      Ra.rx,disp.ab(Rb.ab)
```

Operation:

```
va <- Rbv + SEXT(disp)

(va) <- Rav<7:0>                !STB
(va) <- Rav<15:0>               !STW
(va) <- Rav                     !STL
(va) <- QRav                    !STQ
```

Exceptions:

Access Violation
Fault On Write
Scalar Alignment
Translation Not Valid

Opcodes:

STB	Store Byte from Register to Memory
STW	Store Word from Register to Memory
STL	Store Longword from Register to Memory
STQ	Store Quadword from Register Pair to Memory

Description:

The virtual address is computed by adding register Rb to the sign extended 14-bit displacement. The Ra operand is written to memory at this address.

STQ stores to memory the contents of the even-odd register pair specified by Ra.

Software Note:

In some implementations these instructions may be emulated if the memory operand is not naturally aligned. This could be on the order of 100 times slower. Consequently, when compilers can detect this (e.g. a field in a packed record), they should use an inline multi-instruction sequence to store the operand in pieces rather than incur the emulation overhead.

Gather Memory Data into Vector Register

Format: Masked Vector Memory Operate Format

```
VGATH   Ra.rl,Vb.rl,Vc.wx
VGATH   #a.ib,Vb.rl,Vc.wx
```

Operation:

```
FOR i <- 0 TO VL-1
    BEGIN
    va <- Rav + Vb[i]<31:0>
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        IF {va unaligned} THEN
            {Vector Alignment exception}
        Vc[i] <- (va)<31:0>             !VGATHL
        Vc[i] <- (va)<63:0>             !VGATHQ
        END
    END
```

Exceptions:

Access Violation
Fault On Read
Translation Not Valid
Vector Alignment
Vector Enable

Opcodes:

VGATHL Gather Longword Vector from Memory to Vector Register
VGATHQ Gather Quadword Vector from Memory to Vector Register

Qualifiers:

Masked Operations, Write Intent

Description:

The source operand vector is fetched from memory and written to vector register Vc. The length of the vector is specified by the VL register. The virtual address of the vector is computed using the base address in Ra and the element offsets in vector register Vb. The address of element i (0 LE i LE VL-1) is computed as {Rav + Vb[i]}. The element offset is a signed longword.

The Write Intent qualifier indicates that the data being read might be written in the near future. This feature can be used to optimize write-back cache performance in a multiprocessor system. Memory management does not check for write accessibility.

In VGATHL, bits <31:0> of each destination vector element receive the memory data and bits <63:32> are UNPREDICTABLE. If any vector element is not naturally aligned in memory, a Vector Alignment exception occurs.

These instructions may be omitted in a subset implementation.

Notes:

1. If the same vector register is used as both a source (Vb) and a destination (Vc), the result of the operation is UNPREDICTABLE.

2. When a memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

Load Memory Data into Vector Register

Format: Masked Vector Memory Operate Format

```
VLD     Ra.rl,Rb.rl,Vc.wx
VLD     #a.ib,Rb.rl,Vc.wx
```

Operation:

```
va <- Rbv
FOR i <- 0 TO VL-1
    BEGIN
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        IF {va unaligned} THEN
            {Vector Alignment exception}
        Vc[i] <- (va)<31:0>              !VLDL
        Vc[i] <- (va)<63:0>              !VLDQ
        END
    va <- va + Rav                !Increment by stride
    END
```

Exceptions:

```
Access Violation
Fault On Read
Translation Not Valid
Vector Alignment
Vector Enable
```

Opcodes:

```
VLDL    Load Longword Vector from Memory to Vector Register
VLDQ    Load Quadword Vector from Memory to Vector Register
```

Qualifiers:

Masked Operations, Write Intent

Description:

The source operand vector is fetched from memory and written to vector register Vc. The length of the vector is specified by the VL register. The virtual address of the vector is computed using the base address in Rb and the stride in Ra. The address of element i (0 LE i LE VL-1) is computed as {Rbv + {i*Rav}}. The stride is a signed longword.

The Write Intent qualifier indicates that the data being read might be written in the near future. This feature can be used to optimize write-back cache performance in a multiprocessor system. Memory management does not check for write accessibility.

In VLDL, bits <31:0> of each destination vector element receive the memory data and bits <63:32> are UNPREDICTABLE.

If the vector operand is not naturally aligned in memory a Vector Alignment exception occurs.

These instructions may be omitted in a subset implementation.

Note: When a memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

Scatter Vector Register Data into Memory

Format: Masked Vector Memory Operate Format

```
VSCAT   Ra.rl,Vb.rl,Vc.rx
VSCAT   #a.ib,Vb.rl,Vc.rx
```

Operation:

```
FOR i <- 0 TO VL-1
    BEGIN
    va <- Rav + Vb[i]<31:0>
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        IF {va unaligned} THEN
            {Vector Alignment exception}
        (va) <- Vc[i]<31:0>                 !VSCATL
        (va) <- Vc[i]                       !VSCATQ
        END
    END
```

Exceptions:

Access Violation
Fault On Write
Translation Not Valid
Vector Alignment
Vector Enable

Opcodes:

VSCATL Scatter Longword Vector from Vector Register to Memory
VSCATQ Scatter Quadword Vector from Vector Register to Memory

Qualifiers:

Masked Operations

Description:

The source operand vector is read from vector register Vc and written to memory. The length of the vector is specified by the VL register. The virtual address of the vector is computed using the base address in Ra and the element offsets in vector register Vb. The address of element i (0 LE i LE VL-1) is computed as {Rav + Vb[i]}. The element offset is a signed longword.

If any vector element is not naturally aligned in memory, a Vector Alignment exception occurs.

An implementation may store the vector elements in parallel; therefore, the order in which the elements are stored is UNPREDICTABLE.

These instructions may be omitted in a subset implementation.

Note: When a memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

Store Vector Register Data into Memory

Format: Masked Vector Memory Operate Format

```
VST     Ra.rl,Rb.rl,Vc.rx
VST     #a.ib,Rb.rl,Vc.rx
```

Operation:

```
va <- Rbv

FOR i <- 0 TO VL-1
    BEGIN
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        IF {va unaligned} THEN
            {Vector Alignment exception}
        (va) <- Vc[i]<31:0>                 !VSTL
        (va) <- Vc[i]                       !VSTQ
        END
    va <- va + Rav                          !Increment by stride
    END
```

Exceptions:

Access Violation
Fault On Write
Translation Not Valid
Vector Alignment
Vector Enable

Opcodes:

VSTL	Store Longword Vector from Vector Register to Memory
VSTQ	Store Quadword Vector from Vector Register to Memory

Qualifiers:

Masked Operations

Description:

The source operand vector is read from vector register Vc and written to memory. The length of the vector is specified by the VL register. The virtual address of the vector is computed using the base address in Rb and the stride in Ra. The address of element i (0 LE i LE VL-1) is computed as {Rbv + {i*Rav}}. The stride is a signed longword. If the vector operand is not naturally aligned in memory, a Vector Alignment exception occurs.

An implementation may store the vector elements in parallel; therefore, the order in which the elements are stored is UNPREDICTABLE.

These instructions may be omitted in a subset implementation.

Note: When a memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

4.3 INTEGER ARITHMETIC INSTRUCTIONS

The integer arithmetic instructions perform add, subtract, multiply, divide, and signed and unsigned compare operations.

The integer instructions are summarized below:

Mnemonic	Operation
ADD	Add Longword
CMPEQ	Compare Signed Longword Equal
CMPNE	Compare Signed Longword Not Equal
CMPLT	Compare Signed Longword Less Than
CMPLE	Compare Signed Longword Less Than or Equal
CMPGT	Compare Signed Longword Greater Than
CMPGE	Compare Signed Longword Greater Than or Equal
CMPULT	Compare Unsigned Longword Less Than
CMPULE	Compare Unsigned Longword Less Than or Equal
CMPUGT	Compare Unsigned Longword Greater Than
CMPUGE	Compare Unsigned Longword Greater Than or Equal
DIV	Divide Longword
MULL	Multiply Longword and Return Low 32 Product Bits
UMULH	Unsigned Multiply Longword and Return High 32 Product Bits
SUB	Subtract Longword

Mnemonic	Operation
VADD	Vector Add Longword
VCMPEQ	Vector Compare Signed Longword Equal
VCMPNE	Vector Compare Signed Longword Not Equal
VCMPLT	Vector Compare Signed Longword Less Than
VCMPLE	Vector Compare Signed Longword Less Than or Equal
VCMPGT	Vector Compare Signed Longword Greater Than
VCMPGE	Vector Compare Signed Longword Greater Than or Equal
VCMPULT	Vector Compare Unsigned Longword Less Than
VCMPULE	Vector Compare Unsigned Longword Less Than or Equal
VCMPUGT	Vector Compare Unsigned Longword Greater Than
VCMPUGE	Vector Compare Unsigned Longword Greater Than or Equal
VMULL	Vector Multiply Longword and Return Low 32 Product Bits
VUMULH	Vector Unsigned Multiply Longword and Return High 32 Product Bits
VSUB	Vector Subtract Longword

Integer Add

Format: Operate format

```
ADD     Ra.rl,Rb.rl,Rc.wl
ADD     #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- Rav + Rbv
```

Exceptions:

Integer Overflow

Opcodes:

```
ADD     Add Longword
```

Qualifiers:

Integer Overflow Enable

Description:

Register Ra or a literal is added to register Rb and the 32-bit sum is written to register Rc. On overflow, the least significant 32 bits of the true result are written to the destination register.

The unsigned compare instructions can be used to generate a carry. After adding two values, if the sum is less than (unsigned) either one of the inputs, there was a carry out of the most significant bit.

Integer Signed Compare

Format: Operate format

```
CMP      Ra.rl,Rb.rl,Rc.wl
CMP      #a.ib,Rb.rl,Rc.wl
```

Operation:

```
IF  Rav SIGNED_RELATION Rbv  THEN
    Rc <- 1
ELSE
    Rc <- 0
```

Exceptions:

None

Opcodes:

CMPEQ	Compare Signed Longword Equal
CMPNE	Compare Signed Longword Not Equal
CMPLT	Compare Signed Longword Less Than
CMPLE	Compare Signed Longword Less Than or Equal
CMPGT	Compare Signed Longword Greater Than
CMPGE	Compare Signed Longword Greater Than or Equal

Description:

Register Ra or a literal is compared to Register Rb. If the specified relationship is true, one is written to register Rc; otherwise, zero is written to Rc.

Integer Unsigned Compare

Format: Operate format

```
CMP      Ra.rl,Rb.rl,Rc.wl
CMP      #a.ib,Rb.rl,Rc.wl
```

Operation:

```
IF  Rav UNSIGNED_RELATION Rbv  THEN
    Rc <- 1
ELSE
    Rc <- 0
```

Exceptions:

None

Opcodes:

```
CMPULT  Compare Unsigned Longword Less Than
CMPULE  Compare Unsigned Longword Less Than or Equal
CMPUGT  Compare Unsigned Longword Greater Than
CMPUGE  Compare Unsigned Longword Greater Than or Equal
```

Description:

Register Ra or a literal is compared to Register Rb. If the specified relationship is true, one is written to register Rc; otherwise, zero is written to Rc.

Integer Divide

Format: Operate format

```
DIV  Ra.rl,Rb.rl,Rc.wl
DIV  #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- Rav / Rbv
```

Exceptions:

```
Integer Divide by Zero
Integer Overflow
```

Opcodes:

```
DIV     Divide Longword
```

Qualifiers:

```
Integer Overflow Enable
```

Description:

Register Ra or a literal is divided by register Rb and the quotient is written to register Rc.

On overflow, the least significant 32 bits of the true result are written to the destination register. The quotient result with a zero divisor is UNPREDICTABLE.

These instructions may be omitted in a subset implementation.

NOTE

The first PRISM chips will not implement DIV. DIV will be trapped as a reserved opcode and emulated in software. DIV may be implemented in hardware in a future implementation. Compilers should avoid using DIV. Instead they should generate in-line code to perform the divide operation. See Appendix B for a discussion of how division might be performed. \

Integer Multiply

Format: Operate format

```
MUL     Ra.rl,Rb.rl,Rc.wl
MUL     #a.ib,Rb.rl,Rc.wl
```

Operation:

```
tmp  <- Rav * Rbv          !Signed multiply for MULL
tmp  <- Rav *U Rbv         !Unsigned multiply for UMULH
Rc   <- tmp<31:0>          !MULL
Rc   <- tmp<63:32>         !UMULH
```

Exceptions:

Integer Overflow

Opcodes:

MULL	Multiply Longword and Return Low 32 Product Bits
UMULH	Unsigned Multiply Longword and Return High 32 Product Bits

Qualifiers:

Integer Overflow Enable

Description:

Register Ra or a literal is multiplied by register Rb and either the least or most significant 32 bits of the 64-bit product are written to the destination register. The multiplication is signed for MULL, and unsigned for UMULH.

MULL writes the least significant 32 product bits.

On overflow, the least significant 32 bits of the true result are written to the destination register.

UMULH writes the most significant 32 bits of the unsigned product.

These instructions may be omitted in a subset implementation.

Integer Subtract

Format: Operate format

```
SUB     Ra.rl,Rb.rl,Rc.wl
SUB     #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- Rbv - Rav
```

Exceptions:

Integer Overflow

Opcodes:

SUB Subtract Longword

Qualifiers:

Integer Overflow Enable

Description:

Register Ra or a literal is subtracted from register Rb and the 32-bit difference is written to register Rc. On overflow, the least significant 32 bits of the true result are written to the destination register.

The unsigned compare instructions can be used to generate a borrow. If the minuend (Rbv) is less than (unsigned) the subtrahend (Rav), there will be a borrow.

Vector Integer Add

Format: Masked Vector Arithmetic Operate Format

```
VADD    Va.rl,Vb.rl,Vc.wl
VADD    Ra.rl,Vb.rl,Vc.wl
VADD    #a.ib,Vb.rl,Vc.wl
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        Vc[i]<31:0> <- Va[i]<31:0> + Vb[i]<31:0>  !Vector + Vector
        Vc[i]<31:0> <- Rav + Vb[i]<31:0>          !Scalar + Vector
        END
```

Exceptions:

Integer Overflow
Vector Enable

Opcodes:

VADD Vector Add Longword

Qualifiers:

Integer Overflow Enable, Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) is added, element-wise, to vector register Vb and the 32-bit sum is written to vector register Vc. Only bits <31:0> of each vector element participate in the add operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The length of the vector is specified by the VL register.

If integer overflow is detected, an Integer Overflow exception occurs when the vector operation completes. On overflow, the least significant 32 bits of the true result are written to bits <31:0> of the destination element.

These instructions may be omitted in a subset implementation.

Vector Integer Signed Compare

Format: Masked Vector Arithmetic Operate Format

```
VCMP    Va.rl,Vb.rl
VCMP    Ra.rl,Vb.rl
VCMP    #a.ib,Vb.rl
```

Operation:

```
FOR i <- 0 TO VL-1
    BEGIN
                                                    !Vector cmp Vector
    IF {NOT(masked_mode) OR
            {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0>  THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
                                                    !Scalar cmp Vector
    IF {NOT(masked_mode) OR
            {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Rav SIGNED_RELATION Vb[i]<31:0>  THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
    END
```

Exceptions:

Vector Enable

Opcodes:

```
VCMPEQ  Vector Compare Signed Longword Equal
VCMPNE  Vector Compare Signed Longword Not Equal
VCMPLT  Vector Compare Signed Longword Less Than
VCMPLE  Vector Compare Signed Longword Less Than or Equal
VCMPGT  Vector Compare Signed Longword Greater Than
VCMPGE  Vector Compare Signed Longword Greater Than or Equal
```

Qualifiers:

Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) is compared, element-wise, with vector register Vb. The length of the vector is specified by the VL register.

In masked mode, the compare operation is performed only on elements for which VM<i> matches the mask selector bit in the instruction. If the mask selector bit does not match VM<i>, no comparison is performed and VM<i> is left unchanged.

If masked mode is not used or if VM<i> matches the mask selector bit in masked mode, the specified comparison is performed. If the specified relationship is true, the Vector Mask bit (VM<i>) corresponding to the vector element is set to 1. If the specified

relationship is not true, the Vector Mask bit (VM<i>) corresponding to the vector element is cleared. VM bits beyond the vector length are always left unchanged. Only bits <31:0> of each vector element participate in the operation.

These instructions may be omitted in a subset implementation.

Vector Integer Unsigned Compare

Format: Masked Vector Arithmetic Operate Format

```
VCMP    Va.rl,Vb.rl
VCMP    Ra.rl,Vb.rl
VCMP    #a.ib,Vb.rl
```

Operation:

```
FOR i <- 0 TO VL-1
    BEGIN
                                                    !Vector cmp Vector
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Va[i]<31:0> UNSIGNED_RELATION Vb[i]<31:0>  THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
                                                    !Scalar cmp Vector
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Rav UNSIGNED_RELATION Vb[i]<31:0>  THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
    END
```

Exceptions:

Vector Enable

Opcodes:

VCMPULT Vector Compare Unsigned Longword Less Than
VCMPULE Vector Compare Unsigned Longword Less Than or Equal
VCMPUGT Vector Compare Unsigned Longword Greater Than
VCMPUGE Vector Compare Unsigned Longword Greater Than or Equal

Qualifiers:

Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) is compared, element-wise, with vector register Vb. The length of the vector is specified by the VL register.

In masked mode, the compare operation is performed only on elements for which VM<i> matches the mask selector bit in the instruction. If the mask selector bit does not match VM<i>, no comparison is performed and VM<i> is left unchanged.

If masked mode is not used or if VM<i> matches the mask selector bit in masked mode, the specified comparison is performed. If the specified relationship is true, the Vector Mask bit (VM<i>) corresponding to the vector element is set to 1. If the specified relationship is not true, the Vector Mask bit (VM<i>) corresponding to the vector element is cleared. VM bits beyond the vector length are

always left unchanged. Only bits <31:0> of each vector element participate in the operation.

These instructions may be omitted in a subset implementation.

Vector Integer Multiply

Format: Masked Vector Arithmetic Operate Format

```
VMUL    Va.rl,Vb.rl,Vc.wl
VMUL    Ra.rl,Vb.rl,Vc.wl
VMUL    #a.ib,Vb.rl,Vc.wl
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN                                       !Vector * Vector
        Vc[i] <- {Va[i]<31:0> * Vb[i]<31:0>}<31:0>    !VMULL
        Vc[i] <- {Va[i]<31:0> *U Vb[i]<31:0>}<63:32>  !VUMULH

                                                    !Scalar * Vector
        Vc[i] <- {Rav * Vb[i]<31:0>}<31:0>            !VMULL
        Vc[i] <- {Rav *U Vb[i]<31:0>}<63:32>          !VUMULH
        END
```

Exceptions:

Integer Overflow
Vector Enable

Opcodes:

VMULL	Vector Multiply Longword
VUMULH	Vector Unsigned Multiply Longword and Return High 32 Product Bits

Qualifiers:

Integer Overflow Enable, Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) is multiplied, element-wise, by vector register Vb and the least or most significant 32 bits of the 64-bit product are written to vector register Vc. Only bits <31:0> of each vector element participate in the multiply operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The length of the vector is specified by the VL register.

If overflow is enabled and integer overflow is detected, an Integer Overflow exception occurs when the vector operation completes.

VMULL writes the least significant 32 product product bits. On overflow, the least significant 32 bits of the true result are written to bits <31:0> of the destination element.

VUMULH writes the most significant 32 bits of the unsigned product.

These instructions may be omitted in a subset implementation.

Vector Integer Subtract

Format: Masked Vector Arithmetic Operate Format

```
VSUB    Va.rl,Vb.rl,Vc.wl
VSUB    Ra.rl,Vb.rl,Vc.wl
VSUB    #a.ib,Vb.rl,Vc.wl
```

Operation:

```
FOR i <- 0 TO VL-1
    IF (NOT(masked_mode) OR
                {masked_mode AND (VM<i> EQ selector)}) THEN
        BEGIN
        Vc[i]<31:0> <- Va[i]<31:0> - Vb[i]<31:0> !Vector - Vector
        Vc[i]<31:0> <- Rav - Vb[i]<31:0>         !Scalar - Vector
        END
```

Exceptions:

Integer Overflow
Vector Enable

Opcodes:

VSUB Vector Subtract Longword

Qualifiers:

Integer Overflow Enable, Masked Operations

Description:

A vector operand in register Vb is subtracted, element-wise, from a vector operand (in register Va) or a scalar operand (in register Ra or a literal). The 32-bit difference is written to vector register Vc. Only bits <31:0> of each vector element participate in the subtract operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The length of the vector is specified by the VL register.

If overflow is enabled and integer overflow is detected, an Integer Overflow exception occurs when the vector operation completes. On overflow, the least significant 32 bits of the true result are written to bits <31:0> of the destination element.

These instructions may be omitted in a subset implementation.

4.4 LOGICAL AND SHIFT INSTRUCTIONS

The logical instructions perform longword Boolean operations. The shift instructions perform left and right logical shift, right arithmetic shift, and rotate operations. These are summarized below:

Mnemonic	Operation
AND	Logical Product
BIC	Logical Product with Complement
OR	Logical Sum
ORNOT	Logical Sum with Complement
XOR	Logical Difference
EQV	Logical Equivalence
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
ROT	Rotate
VAND	Vector Logical Product
VBIC	Vector Logical Product with Complement
VOR	Vector Logical Sum
VORNOT	Vector Logical Sum with Complement
VMERGE	Vector Merge
VXOR	Vector Logical Difference
VEQV	Vector Logical Equivalence
VSLL	Vector Shift Left Logical
VSRL	Vector Shift Right Logical

\There is no arithmetic left-shift instruction because, typically, where an arithmetic left shift would be used, a logical shift will do. For multiplying by a small power of two in address computations, logical left shift is acceptable. Arithmetic left shift is more complicated because it requires overflow detection. Integer multiply should be used to perform an arithmetic left shift with overflow checking.

Bit field extracts can be done with two logical shifts. Sign extension can be done with left logical shift and a right arithmetic shift.

There are no quadword to longword shifts because this can be done with a three instruction sequence (SLL, SRL, OR).\

Logical Functions

Format: Operate format

```
opcode   Ra.rl,Rb.rl,Rc.wl
opcode   #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- Rav AND Rbv              !AND
Rc <- Rav OR  Rbv              !OR
Rc <- Rav XOR Rbv              !XOR
Rc <- {NOT Rav} AND Rbv        !BIC
Rc <- {NOT Rav} OR  Rbv        !ORNOT
Rc <- {NOT Rav} XOR Rbv        !EQV
```

Exceptions:

None

Opcodes:

AND	Logical Product
OR	Logical Sum
XOR	Logical Difference
BIC	Bit Clear
ORNOT	Logical Sum with Complement
EQV	Logical Equivalence

Description:

These instructions perform the designated Boolean function between register Ra or a literal and register Rb. The result is written to register Rc.

The "NOT" function can be performed by doing an ORNOT with zero (Rb = R0).

Shift Logical

Format: Operate format

```
opcode   Ra.rl,Rb.rl,Rc.wl
opcode   #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- LEFT_SHIFT(Rbv,  Rav<4:0>)        !SLL
Rc <- RIGHT_SHIFT(Rbv, Rav<4:0>)        !SRL
```

Exceptions:

None

Opcodes:

SLL	Shift Left Logical
SRL	Shift Right Logical

Description:

Register Rb is shifted logically left or right 0 to 31 bits by the count in register Ra or a literal. The result is written to register Rc. Zero bits are propagated into the vacated bit positions.

Bits <31:5> of the count operand are ignored.

Shift Arithmetic

Format: Operate format

```
SRA     Ra.rl,Rb.rl,Rc.wl
SRA     #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- ARITH_SHIFT(Rbv, Rav<4:0>)
```

Exceptions:

None

Opcodes:

SRA Shift Right Arithmetic

Description:

Register Rb is right shifted arithmetically 0 to 31 bits by the count in register Ra or a literal. The result is written to register Rc. The sign bit (Rbv<31>) is propagated into the vacated bit positions.

Bits <31:5> of the count operand are ignored.

Rotate

Format: Operate format

```
ROT     Ra.rl,Rb.rl,Rc.wl
ROT     #a.ib,Rb.rl,Rc.wl
```

Operation:

```
Rc <- BIT_ROTATE(Rbv, Rav<4:0>)
```

Exceptions:

None

Opcodes:

ROT	Rotate Bits

Description:

Register Rb is rotated left 0 to 31 bits by the count in register Ra or literal. The result is written to register Rc.

Bits <31:5> of the count operand are ignored.

Vector Logical Functions

Format: Masked Vector Arithmetic Operate Format

```
opcode   Va.rl,Vb.rl,Vc.wl
opcode   Ra.rl,Vb.rl,Vc.wl
opcode   #a.ib,Vb.rl,Vc.wl
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        ! Vector op Vector
        Vc[i]<31:0> <- Va[i]<31:0> AND Vb[i]<31:0>          !VAND
        Vc[i]<31:0> <- Va[i]<31:0> OR  Vb[i]<31:0>          !VOR
        Vc[i]<31:0> <- Va[i]<31:0> XOR Vb[i]<31:0>          !VXOR
        Vc[i]<31:0> <- {NOT Va[i]<31:0>} AND Vb[i]<31:0>    !VBIC
        Vc[i]<31:0> <- {NOT Va[i]<31:0>} OR  Vb[i]<31:0>    !VORNOT
        Vc[i]<31:0> <- {NOT Va[i]<31:0>} XOR Vb[i]<31:0>    !VEQV

        ! Scalar op Vector
        Vc[i]<31:0> <- Rav AND Vb[i]<31:0>                  !VAND
        Vc[i]<31:0> <- Rav OR  Vb[i]<31:0>                  !VOR
        Vc[i]<31:0> <- Rav XOR Vb[i]<31:0>                  !VXOR
        Vc[i]<31:0> <- {NOT Rav} AND Vb[i]<31:0>            !VBIC
        Vc[i]<31:0> <- {NOT Rav} OR  Vb[i]<31:0>            !VORNOT
        Vc[i]<31:0> <- {NOT Rav} XOR Vb[i]<31:0>            !VEQV

        Vc[i]<63:32> <- Vb[i]<63:32>
        END
```

Exceptions:

Vector Enable

Opcodes:

VAND	Vector Logical Product
VOR	Vector Logical Sum
VXOR	Vector Logical Difference
VBIC	Vector Logical Product with Complement
VORNOT	Vector Logical Sum with Complement
VEQV	Vector Logical Equivalence

Qualifiers:

Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) are combined, element-wise, using the specified Boolean function, with vector register Vb and the 32-bit result is written to vector register Vc. Only bits <31:0> of each vector element participate in the Boolean operation. Bits <63:32> of the destination vector elements receive bits <63:32> of the Vb elements. The length of the vector is specified by the VL register.

These instructions may be omitted in a subset implementation.

Vector Merge

Format: Masked Vector Arithmetic Operate Format

```
VMERGE    Va.rq,Vb.rq,Vc.wq
VMERGE    Ra.rq,Vb.rq,Vc.wq
VMERGE    #a.ib,Vb.rq,Vc.wq
```

Operation:

```
FOR i <- 0 TO VL-1
    BEGIN
    IF  VM<i> EQ S THEN          !Vector op Vector
        Vc[i] <- Va[i]
    ELSE
        Vc[i] <- Vb[i]

    IF  VM<i> EQ S THEN          !Scalar op Vector
        Vc[i] <- Rav
    ELSE
        Vc[i] <- Vb[i]
    END
```

Exceptions:

Vector Enable

Opcodes:

VMERGE Vector Merge

Qualifiers:

Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) are merged, element-wise, with vector register Vb and the resulting vector is written to vector register Vc. The length of the vector operation is specified by the VL register.

For each vector element, i, if the corresponding Vector Mask bit (VM<i>) matches the Mask Selector (S) bit, Va[i] or Rav is written to the destination vector element Vc[i]. If VM<i> does not match S, Vb[i] is written to the destination vector element.

Refer to Section 3.3.3.1 for a description of the Mask Selector bit.

This instruction may be omitted in a subset implementation.

Vector Shift Logical

Format: Masked Vector Arithmetic Operate Format

```
opcode  Va.rl,Vb.rl,Vc.wl
opcode  Ra.rl,Vb.rl,Vc.wl
opcode  #a.ib,Vb.rl,Vc.wl
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        ! shift vector by vector
        Vc[i]<31:0> <- LEFT_SHIFT(Vb[i]<31:0>,  Va[i]<4:0>) !VS
        Vc[i]<31:0> <- RIGHT_SHIFT(Vb[i]<31:0>, Va[i]<4:0>) !VS

        ! shift vector by scalar
        Vc[i]<31:0> <- LEFT_SHIFT(Vb[i]<31:0>,  Rav<4:0>)   !VS
        Vc[i]<31:0> <- RIGHT_SHIFT(Vb[i]<31:0>, Rav<4:0>)   !VS
        END
```

Exceptions:

Vector Enable

Opcodes:

VSLL	Vector Shift Left Logical
VSRL	Vector Shift Right Logical

Qualifiers:

Masked Operations

Description:

Each element in vector register Vb is shifted logically left or right 0 to 31 bits by the count specified by a vector operand (in register Va) or a scalar operand (in register Ra or a literal). The shifted results are written to vector register Vc. Zero bits are propagated into the vacated bit positions. Only bits <4:0> of the count operand and bits <31:0> of each Vb element participate in the shift operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The length of the vector is specified by the VL register.

These instructions may be omitted in a subset implementation.

4.5 FLOATING-POINT INSTRUCTIONS

PRISM provides instructions for operating on VAX G_floating and F_floating-point operand formats. The floating-point arithmetic instructions are add, subtract, compare, multiply, and divide. Two rounding modes are provided: VAX rounding and round toward zero (chopped).

All G_floating operands must be in even-odd register pairs or the result of the operation is UNPREDICTABLE.

Data conversion instructions are provided to convert operands between G_floating and F_floating and longword integer.

The instructions provided are summarized below:

Mnemonic	Operation
ADDF	Add F_floating
CMPFEQ	Compare F_floating Equal
CMPFNE	Compare F_floating Not Equal
CMPFLT	Compare F_floating Less Than
CMPFLE	Compare F_floating Less Than or Equal
CMPFGT	Compare F_floating Greater Than
CMPFGE	Compare F_floating Greater Than or Equal
CVTLF	Convert Longword to F_floating
CVTFL	Convert F_floating to Longword
CVTFG	Convert F_floating to G_floating
DIVF	Divide F_floating
MULF	Multiply F_floating
SUBF	Subtract F_floating
ADDG	Add G_floating
CMPGEQ	Compare G_floating Equal
CMPGNE	Compare G_floating Not Equal
CMPGLT	Compare G_floating Less Than
CMPGLE	Compare G_floating Less Than or Equal
CMPGGT	Compare G_floating Greater Than
CMPGGE	Compare G_floating Greater Than or Equal
CVTGF	Convert G_ to F_floating
CVTLG	Convert Longword to G_floating
CVTGL	Convert G_floating to Longword
DIVG	Divide G_floating
MULG	Multiply G_floating
SUBG	Subtract G_floating

Mnemonic	Operation
VADDF	Vector Add F_floating
VCMPFEQ	Vector Compare F_floating Equal
VCMPFNE	Vector Compare F_floating Not Equal
VCMPFLT	Vector Compare F_floating Less Than
VCMPFLE	Vector Compare F_floating Less Than or Equal
VCMPFGT	Vector Compare F_floating Greater Than
VCMPFGE	Vector Compare F_floating Greater Than or Equal
VCVTLF	Vector Convert Longword to F_floating
VCVTFL	Vector Convert F_floating to Longword
VCVTFG	Vector Convert F_floating to G_floating
VDIVF	Vector Divide F_floating
VMULF	Vector Multiply F_floating
VSUBF	Vector Subtract F_floating
VADDG	Vector Add G_floating
VCMPGEQ	Vector Compare G_floating Equal
VCMPGNE	Vector Compare G_floating Not Equal
VCMPGLT	Vector Compare G_floating Less Than
VCMPGLE	Vector Compare G_floating Less Than or Equal
VCMPGGT	Vector Compare G_floating Greater Than
VCMPGGE	Vector Compare G_floating Greater Than or Equal
VCVTGF	Vector Convert G_ to F_floating
VCVTLG	Vector Convert Longword to G_floating
VCVTGL	Vector Convert G_floating to Longword
VDIVG	Vector Divide G_floating
VMULG	Vector Multiply G_floating
VSUBG	Vector Subtract G_floating

4.5.1 Literals

Literals used as floating-point operands produce UNPREDICTABLE results. Literals are allowed in longword convert instructions for integer source operands.

4.5.2 Accuracy

Using VAX rounding, PRISM generates floating-point results with an error bound of 1/2 Least Significant Bit (LSB) for all floating-point instructions.

General comments on the accuracy of the PRISM floating-point instruction set are presented here.

An instruction is defined to be exact if its result, extended on the right by an infinite sequence of zeros, is identical to that of an infinite-precision calculation involving the same operands. The a priori accuracy of the operands is thus ignored. For all arithmetic operations, except DIV, a zero operand implies that the instruction is exact. The same statement holds for DIV if the zero operand is the dividend. But if it is the divisor, division is undefined, the result is UNPREDICTABLE, and the operation causes an Arithmetic exception.

For non-zero floating-point operands, the fractional factor is binary normalized with 24 or 53 bits for single (F_floating) or double precision (G_Floating), respectively.

\For ADD, SUB, MUL, and DIV, an overflow bit, on the left, and two guard bits, on the right, are necessary and sufficient to guarantee return of a rounded result identical to the corresponding infinite-precision operation rounded to the specified word length. Thus with two guard bits, a rounded result has an error bound of 1/2 LSB.\

Note that an arithmetic result is exact if no non-zero bits are lost in chopping the infinite-precision result to the data length to be stored. Chopping is defined to mean that the 24 (F_floating) or 53 (G_floating) high-order bits of the normalized result fraction are stored; the rest of the bits are discarded. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows:

1. If the rounding bit is 1, the rounded result is the chopped result incremented by an LSB.

2. If the rounding bit is 0, the rounded and chopped results are identical.

All PRISM processors implement rounding so as to produce results identical to the results produced by the following algorithm. After normalization, add a 1 to the rounding bit, and propagate the carry, if it occurs. Note that re-normalization may be required after rounding takes place. The following statements summarize the relations among chopped, rounded, and true (infinite-precision) results:

- If a stored result is exact:

  rounded value = chopped value = true value.

- If a stored result is not exact, its magnitude is always:

  1. Less than that of the true result for chopping.

  2. Less than that of the true result for rounding if the rounding bit is 0.

  3. Greater than that of the true result for rounding if the rounding bit is 1.

One overflow bit and two guard bits are adequate to guarantee accuracy of rounded ADD, SUB, MUL, or DIV, provided that the algorithms are properly chosen:

- ADD or SUB: Note, first, that ADD or SUB may result in propagation of a carry, and hence the overflow bit is necessary. Second, if in ADD or SUB there is a one-bit loss of significance with an alignment shift of two or more bits, the first guard bit is needed for the LSB of the normalized result, and the second is then the rounding bit. Therefore, the three bits are necessary. A number of constraints must be observed in selection of the algorithms for the basic operations, in order for these three bits to be sufficient to guarantee an error bound of 1/2 LSB for unbiased rounding:

  1. If the alignment shift does not exceed two, there are no constraints, because no bits can be lost.

  2. If the alignment shift exceeds two (or however many guard bits are used, say g GE 2), no negations may be made after the alignment shift takes place.

  3. If the above constraint is observed, the error bound for a rounded result is 1/2 LSB. If, however, a negation follows the alignment shift, the error bound will be:

     (1/2)*(1 + 2**(-g + 2))LSB

     This is because a "borrow" will be lost on an implicit subtraction, if non-zero bits were lost in the alignment shift. Note: The error bound is 1 LSB if the constraint is ignored and there are only two guard bits (g = 2).

  4. The constraint on no negations after the alignment shift may be replaced by keeping track of non-zero bits lost during the alignment shift, and then negating by one's complement if any "ones" were lost, and by two's complement if none were lost. If this is done, the error bound will be 1/2 LSB.

- MUL:

  1. The product of two normalized binary fractions can be as small as 1/4, and must be less than one. The overflow bit is not needed for MUL, but the first guard bit will be necessary for normalization if the product is less than 1/2, and, in this case, the second guard bit is the rounding bit.

  2. The first constraint on MUL is that the product be generated from the least to the most significant bit. Low order bits, in positions to the right of the second guard bit, may be discarded, but ONLY AFTER they have made their contribution to carries which could propagate into the guard bits or beyond.

  3. For the same reasons as for ADD or SUB, if low order bits of the product have been discarded, no negations can be made after generating the product.

- DIV:

  1. For standard algorithms it is necessary that the remainder be generated exactly at each step; the overflow and two guard bits are adequate for this purpose. The register receiving the quotient must have a guard bit for the rounding bit, and the quotient must be developed to include the rounding bit.

  2. The Newton-Raphson quadratic convergence algorithms, which might make good use of high-speed multiplication logic, require a number of guard bits equal to twice the number of bits desired in the result if the correctness of the rounding bit is to be guaranteed.

4.5.3 Floating-Point Exceptions

All floating-point exceptions are traps; see Chapter 6, Exceptions and Interrupts, Section 6.4.1. The floating-point operation completes by writing a reserved operand (see Chapter 2, Sections 2.2.5 and 2.2.6) with the exception type encoded in it. The figure below illustrates this:

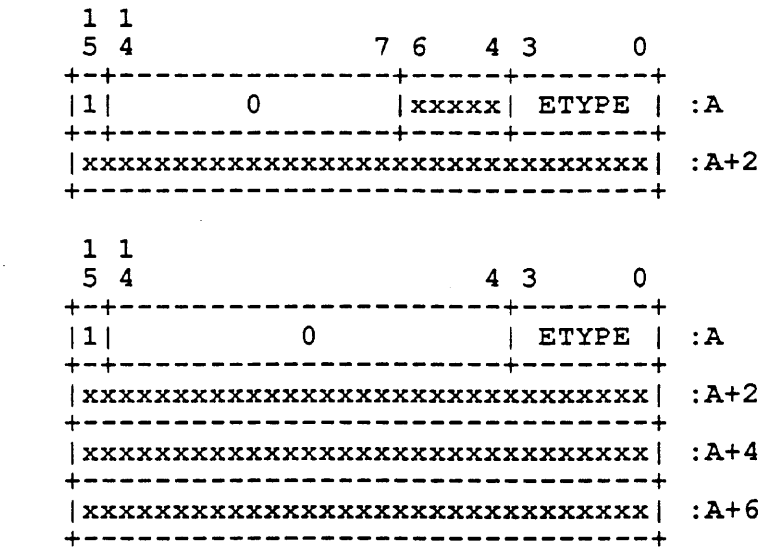


Figure 4-1: F_ and G_floating Exception Code Format

The sign, bit <15>, is 1 and the exponent (bits <14:7> for F_floating and bits <14:4> for G_floating) is zero. The exception type (ETYPE) is encoded in bits <3:0>, so as to correspond to bits <3:0> in the exception summary (see Chapter 6, Exceptions and Interrupts, Figure 6-4, Page 6-14). If multiple exceptions occur (i.e. reserved operand divided by zero), multiple bits may be set in the ETYPE field.

The state of all other bits in the result (denoted by an "x") are UNPREDICTABLE.

If the Floating Underflow exception is suppressed by the instruction, a zero result is written to the destination register and no Underflow exception is signaled. Floating Overflow, Floating Reserved Operand, and Floating Divide by Zero are always enabled.

Floating Add

Format: Operate format

```
        ADD     Ra.rx,Rb.rx,Rc.wx
```

Operation:

```
        Rc <- Rav + Rbv                 !F_floating
        QRc <- QRav + QRbv              !G_floating
```

Exceptions:

```
        Floating Overflow
        Floating Reserved Operand
        Floating Underflow
```

Opcodes:

```
        ADDF    Add F_Floating
        ADDG    Add G_Floating
```

Qualifiers:

```
        Floating Underflow Enable, Chopped
```

Description:

Register Ra (QRa) is added to register Rb (QRb) and the sum is written to register Rc (QRc). If Floating Underflow is disabled, zero is written to the destination register Rc (QRc) when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Floating Compare

Format: Operate format

```
CMP     Ra.rx,Rb.rx,Rc.wl
```

Operation:

```
IF  Rav SIGNED_RELATION Rbv  THEN       !F_floating
    Rc <- 1
ELSE
    Rc <- 0

IF  QRav SIGNED_RELATION QRbv  THEN     !G_floating
    Rc <- 1
ELSE
    Rc <- 0
```

Exceptions:

Floating Reserved Operand

Opcodes:

```
CMPFEQ  Compare F_floating Equal
CMPFNE  Compare F_floating Not Equal
CMPFLT  Compare F_floating Less Than
CMPFLE  Compare F_floating Less Than or Equal
CMPFGT  Compare F_floating Greater Than
CMPFGE  Compare F_floating Greater Than or Equal

CMPGEQ  Compare G_floating Equal
CMPGNE  Compare G_floating Not Equal
CMPGLT  Compare G_floating Less Than
CMPGLE  Compare G_floating Less Than or Equal
CMPGGT  Compare G_floating Greater Than
CMPGGE  Compare G_floating Greater Than or Equal
```

Description:

The two F_ or G_floating operands in Ra (QRa) and Rb (QRb) are compared. If the specified relationship is true, one is written to register Rc; otherwise, zero is written to Rc. Rc is UNPREDICTABLE if the source operand in Ra (QRa) or Rb (QRb) is a floating reserved operand.

These instructions may be omitted in a subset implementation.

Convert F_Floating to G_Floating

Format: Operate format

```
CVT     Ra.rf,Rc.wg
```

Operation:

```
QRc <- {conversion of Rav}
```

Exceptions:

Floating Reserved Operand

Opcodes:

```
CVTFG   Convert F_floating to G_floating
```

Description:

The F_floating source operand in register Ra is converted to a G_floating result and written to register QRc. No rounding is required because there are more fraction bits in a G_floating operand than in an F_floating operand.

This instruction may be omitted in a subset implementation.

Convert G_Floating to F_Floating

Format: Operate format

```
CVT     Ra.rg,Rc.wf
```

Operation:

```
Rc <- {conversion of QRav}
```

Exceptions:

```
Floating Overflow
Floating Reserved Operand
Floating Underflow
```

Opcodes:

```
CVTGF   Convert G_floating to F_floating
```

Qualifiers:

```
Floating Underflow Enable, Chopped
```

Description:

The G_floating source operand in register QRa is rounded to an F_floating result and written to register Rc. If Floating Underflow is disabled, zero is written to the destination register Rc when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Convert Floating to Longword

Format: Operate format

```
CVT     Ra.rx,Rc.wl
```

Operation:

```
Rc <- {conversion of Rav}                !F_floating
Rc <- {conversion of QRav}               !G_floating
```

Exceptions:

Integer Overflow
Floating Reserved Operand

Opcodes:

```
CVTFL   Convert F_floating to Longword
CVTGL   Convert G_floating to Longword
```

Qualifiers:

Chopped

Description:

The F_ or G_floating operand in register Ra (QRa) is converted to a longword and written to register Rc. Rc is UNPREDICTABLE if the source operand in Ra (QRa) is a floating reserved operand.

On overflow, the least significant 32 bits of the true result are written to the destination register.

These instructions may be omitted in a subset implementation.

Convert Longword to Floating

Format: Operate format

```
CVT     Ra.rl,Rc.wx
CVT     #a.ib,Rc.wx
```

Operation:

```
Rc <- {conversion of Rav}                !F_floating
QRc <- {conversion of Rav}               !G_floating
```

Exceptions:

None

Opcodes:

```
CVTLF   Convert Longword to F_floating
CVTLG   Convert Longword to G_floating
```

Qualifiers:

```
Chopped                  !CVTLF only
```

Description:

The longword operand in register Ra or a literal is converted to an F_ or G_floating result and written to register Rc (QRc). No rounding is required on CVTLG because the result is exact.

These instructions may be omitted in a subset implementation.

Floating Divide

Format: Operate format

```
DIV     Ra.rx,Rb.rx,Rc.wx
```

Operation:

```
Rc <- Rav / Rbv                 !F_floating
QRc <- QRav / QRbv              !G_floating
```

Exceptions:

Floating Divide by Zero
Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

DIVF	Divide F_floating
DIVG	Divide G_floating

Qualifiers:

Floating Underflow Enable, Chopped

Description:

The dividend in register Ra (QRa) is divided by the divisor in register Rb (QRb), and the quotient is written to register Rc (QRc). If Floating Underflow is disabled, zero is written to the destination register Rc (QRc) when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Floating Multiply

Format: Operate format

```
MUL     Ra.rx,Rb.rx,Rc.wx
```

Operation:

```
Rc <- Rbv * Rav                 !F_floating
QRc <- QRbv * QRav              !G_floating
```

Exceptions:

```
Floating Overflow
Floating Reserved Operand
Floating Underflow
```

Opcodes:

```
MULF    Multiply F_floating
MULG    Multiply G_floating
```

Qualifiers:

Floating Underflow Enable, Chopped

Description:

The multiplicand in register Rb (QRb) is multiplied by the multiplier in register Ra (QRa), and the product is written to register Rc (QRc). If Floating Underflow is disabled, zero is written to the destination register Rc (QRc) when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Floating Subtract

Format: Operate format

```
SUB     Ra.rx,Rb.rx,Rc.wx
```

Operation:

```
Rc <- Rav - Rbv                 !F_floating
QRc <- QRav - QRbv              !G_floating
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

```
SUBF    Subtract F_floating
SUBG    Subtract G_floating
```

Qualifiers:

Floating Underflow Enable, Chopped

Description:

The subtrahend operand in register Rb (QRb) is subtracted from the minuend operand in register Ra (QRa), and the difference is written to register Rc (QRc). If Floating Underflow is disabled, zero is written to the destination register Rc (QRc) when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Vector Floating Add

Format: Masked Vector Arithmetic Operate Format

```
VADD    Va.rx,Vb.rx,Vc.wx
VADD    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
                                                    !VADDF
        Vc[i] <- Va[i]<31:0> + Vb[i]<31:0>          !Vector + Vector
        Vc[i] <- Rav + Vb[i]<31:0>                  !Scalar + Vector
                                                    !VADDG
        Vc[i] <- Va[i] + Vb[i]                      !Vector + Vector
        Vc[i] <- QRav  + Vb[i]                      !Scalar + Vector
        END
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow
Vector Enable

Opcodes:

VADDF Vector Add F_Floating
VADDG Vector Add G_Floating

Qualifiers:

Masked Operations, Floating Underflow Enable, Chopped

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or QRa) is added, element-wise, to vector register Vb and the sum is written to vector register Vc. The length of the vector is specified by the VL register.

In VADDF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. If an exception is detected, it occurs when the vector operation completes. If Floating Underflow is disabled, zero is written to the destination element when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Vector Floating Compare

Format: Masked Vector Arithmetic Operate Format

```
VCMP    Va.rx,Vb.rx
VCMP    Ra.rx,Vb.rx
```

Operation:

```
FOR i <- 0 TO VL-1
    BEGIN
                                !VCMPF Vector cmp Vector
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0>  THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
                                !VCMPF Scalar cmp Vector
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Rav SIGNED_RELATION Vb[i]<31:0>  THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
                                !VCMPG Vector cmp Vector
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  Va[i] SIGNED_RELATION Vb[i] THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
                                !VCMPG Scalar cmp Vector
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        IF  QRav SIGNED_RELATION Vb[i] THEN
            VM<i> <- 1
        ELSE
            VM<i> <- 0
    END
```

Exceptions:

Floating Reserved Operand
Vector Enable

Opcodes:

VCMPFEQ Vector Compare F_floating Equal
VCMPFNE Vector Compare F_floating Not Equal
VCMPFLT Vector Compare F_floating Less Than
VCMPFLE Vector Compare F_floating Less Than or Equal
VCMPFGT Vector Compare F_floating Greater Than
VCMPFGE Vector Compare F_floating Greater Than or Equal

VCMPGEQ Vector Compare G_floating Equal
VCMPGNE Vector Compare G_floating Not Equal
VCMPGLT Vector Compare G_floating Less Than
VCMPGLE Vector Compare G_floating Less Than or Equal
VCMPGGT Vector Compare G_floating Greater Than
VCMPGGE Vector Compare G_floating Greater Than or Equal

Qualifiers:

Masked Operations

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or QRa) is compared, element-wise, with vector register Vb. The length of the vector is specified by the VL register.

In masked mode, the compare operation is performed only on elements for which VM<i> matches the mask selector bit in the instruction. If the mask selector bit does not match VM<i>, no comparison is performed and VM<i> is left unchanged.

If masked mode is not used or if VM<i> matches the mask selector bit in masked mode, the specified comparison is performed. If the specified relationship is true, the Vector Mask bit (VM<i>) corresponding to the vector element is set to 1. If the specified relationship is not true, the Vector Mask bit (VM<i>) corresponding to the vector element is cleared. VM bits beyond the vector length are always left unchanged.

In VCMPFx, only bits <31:0> of each vector element participate in the operation. VM<i> is UNPREDICTABLE if a reserved operand is detected in Va[i], Vb[i], or Rav (Qrav). If an exception is detected, it occurs when the vector operation completes.

These instructions may be omitted in a subset implementation.

Vector Convert F_Floating to G_Floating

Format: Masked Vector Arithmetic Operate Format

```
VCVT     Vb.rf,Vc.wg
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        Vc[i] <- {conversion of Vb[i]<31:0>}
        END
```

Exceptions:

Floating Reserved Operand
Vector Enable

Opcodes:

VCVTFG Vector Convert F_floating to G_floating

Qualifiers:

Masked Operations

Description:

The F_floating vector elements in vector register Vb are converted to G_floating results and written to vector register Vc. No rounding is required because all F_floating fraction bits fit within a G_floating fraction. The length of the vector is specified by the VL register.

If an exception is detected, it occurs when the vector operation completes.

This instruction may be omitted in a subset implementation.

Vector Convert G_Floating to F_Floating

Format: Masked Vector Arithmetic Operate Format

```
VCVT     Vb.rg,Vc.wf
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        Vc[i] <- {conversion of Vb[i]}
        END
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow
Vector Enable

Opcodes:

VCVTGF Vector Convert G_floating to F_floating

Qualifiers:

Masked Operations, Floating Underflow Enable, Chopped

Description:

The G_floating vector elements in vector register Vb are converted to F_floating results and written to bits <31:0> of vector register Vc. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The length of the vector is specified by the VL register. If Floating Underflow is disabled, zero is written to the destination vector element when an exponent underflow occurs.

If an exception is detected, it occurs when the vector operation completes.

These instructions may be omitted in a subset implementation.

Vector Convert Floating to Longword

Format: Masked Vector Arithmetic Operate Format

```
VCVT     Vb.rx,Vc.wl
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        Vc[i] <- {conversion of Vb[i]}          !VCVTGL
        Vc[i] <- {conversion of Vb[i]<31:0>}    !VCVTFL
    END
```

Exceptions:

Floating Reserved Operand
Integer Overflow
Vector Enable

Opcodes:

VCVTFL Vector Convert F_floating to Longword
VCVTGL Vector Convert G_floating to Longword

Qualifiers:

Masked Operations, Chopped

Description:

The F_ or G_floating vector elements in vector register Vb are converted to longwords and written to bits <31:0> of the vector register Vc. Bits <63:32> of the destination vector elements are UNPREDICTABLE. Vc[i] is UNPREDICTABLE if the source operand in Vb[i] is a floating reserved operand. The length of the vector is specified by the VL register.

If an exception is detected, it occurs when the vector operation completes. On overflow, the least significant 32 bits of the true result are written to the destination vector element.

These instructions may be omitted in a subset implementation.

Vector Convert Longword to Floating

Format: Masked Vector Arithmetic Operate Format

```
VCVT     Vb.rl,Vc.wx
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
        Vc[i] <- {conversion of Vb[i]<31:0>}
        END
```

Exceptions:

Vector Enable

Opcodes:

VCVTLF	Vector Convert Longword to F_floating	
VCVTLG	Vector Convert Longword to G_floating	

Qualifiers:

Masked Operations, Chopped	!VCVTLF
Masked Operations	!VCVTLG

Description:

The longword integer vector elements in register Vb are converted to F_ or G_floating results and written to vector register Vc. In VCVTLF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. No rounding is required on VCVTLG because the result is exact. The length of the vector is specified by the VL register.

These instructions may be omitted in a subset implementation.

Vector Floating Divide

Format: Masked Vector Arithmetic Operate Format

```
VDIV    Va.rx,Vb.rx,Vc.wx
VDIV    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
FOR i <- 0 TO VL-1
    IF (NOT(masked_mode) OR
                {masked_mode AND (VM<i> EQ selector)}) THEN
        BEGIN
                                                !VDIVF
        Vc[i] <- Va[i]<31:0> / Vb[i]<31:0>   !Vector / Vector
        Vc[i] <- Rav / Vb[i]<31:0>           !Scalar / Vector
                                                !VDIVG
        Vc[i] <- Va[i] / Vb[i]               !Vector / Vector
        Vc[i] <- QRav / Vb[i]                !Scalar / Vector
        END
```

Exceptions:

```
Floating Divide by Zero
Floating Overflow
Floating Reserved Operand
Floating Underflow
Vector Enable
```

Opcodes:

```
VDIVF   Vector Divide F_floating
VDIVG   Vector Divide G_floating
```

Qualifiers:

Masked Operations, Floating Underflow Enable, Chopped

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or QRa) is divided, element-wise, by a vector operand in register Vb and the quotient is written to vector register Vc. The length of the vector is specified by the VL register.

In VDIVF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

If an exception is detected, it occurs when the vector operation completes. If Floating Underflow is disabled, zero is written to the destination vector element when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Vector Floating Multiply

Format: Masked Vector Arithmetic Operate Format

```
VMUL    Va.rx,Vb.rx,Vc.wx
VMUL    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
FOR i <- 0 TO VL-1
    IF (NOT(masked_mode) OR
                {masked_mode AND (VM<i> EQ selector)}) THEN
        BEGIN
                                                !VMULF
        Vc[i] <- Va[i]<31:0> * Vb[i]<31:0>   !Vector * Vector
        Vc[i] <- Rav * Vb[i]<31:0>               !Scalar * Vector
                                                !VMULG
        Vc[i] <- Va[i] * Vb[i]                   !Vector * Vector
        Vc[i] <- QRav  * Vb[i]                   !Scalar * Vector
        END
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow
Vector Enable

Opcodes:

VMULF	Vector Multiply F_floating
VMULG	Vector Multiply G_floating

Qualifiers:

Masked Operations, Floating Underflow Enable, Chopped

Description:

The multiplicand in vector register Vb is multiplied, element-wise, by the multiplier vector operand (in register Va) or a scalar operand (in register Ra or QRa), and the product is written to vector register Vc. The length of the vector is specified by the VL register.

In VMULF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. If an exception is detected, it occurs when the vector operation completes. If Floating Underflow is disabled, zero is written to the destination vector element when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

Vector Floating Subtract

Format: Masked Vector Arithmetic Operate Format

```
VSUB    Va.rx,Vb.rx,Vc.wx
VSUB    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
FOR i <- 0 TO VL-1
    IF {NOT(masked_mode) OR
                {masked_mode AND {VM<i> EQ selector}}} THEN
        BEGIN
                                                    !VSUBF
        Vc[i] <- Va[i]<31:0> - Vb[i]<31:0>          !Vector - Vector
        Vc[i] <- Rav - Vb[i]<31:0>                  !Scalar - Vector
                                                    !VSUBG
        Vc[i] <- Va[i] - Vb[i]                      !Vector - Vector
        Vc[i] <- QRav - Vb[i]                       !Scalar - Vector
        END
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow
Vector Enable

Opcodes:

VSUBF	Vector Subtract F_floating
VSUBG	Vector Subtract G_floating

Qualifiers:

Masked Operations, Floating Underflow Enable, Chopped

Description:

A vector operand in register Vb is subtracted, element-wise, from a vector operand (in register Va) or a scalar operand (in register Ra or QRa). The difference is written to vector register Vc. The length of the vector is specified by the VL register.

In VSUBF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. If an exception is detected, it occurs when the vector operation completes. If Floating Underflow is disabled, zero is written to the destination element when an exponent underflow occurs.

These instructions may be omitted in a subset implementation.

4.6 CONTROL INSTRUCTIONS

PRISM provides eight conditional branch instructions, a Fault On Bit instruction, and a Jump To Subroutine instruction.

Mnemonic	Operation
BEQ	Branch if Register Equal to Zero
BNE	Branch if Register Not Equal to Zero
BLT	Branch if Register Less Than Zero
BLE	Branch if Register Less Than or Equal to Zero
BGT	Branch if Register Greater Than Zero
BGE	Branch if Register Greater Than or Equal to Zero
BLBS	Branch if Register Low Bit is Set
BLBC	Branch if Register Low Bit is Clear
FLBC	Fault On Low Bit Clear
JSR	Jump to Subroutine

Conditional Branch

Format: Branch format

```
Bxx   Ra.rl,disp.al
```

Operation:

```
va <- PC + {4*SEXT(disp)}
IF  TEST(Rav)  THEN
    PC <- va
```

Exceptions:

None

Opcodes:

BEQ	Branch if Register Equal to Zero
BNE	Branch if Register Not Equal to Zero
BLT	Branch if Register Less Than Zero
BLE	Branch if Register Less Than or equal to Zero
BGT	Branch if Register Greater Than Zero
BGE	Branch if Register Greater Than or Equal to Zero
BLBS	Branch if Register Low Bit is Set
BLBC	Branch if Register Low Bit is Clear

Description:

Register Ra is tested. If the specified relationship is true, the PC is loaded with the target virtual address; otherwise, execution continues with the next sequential instruction.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign extended to 32 bits, and added to the updated PC to form the target virtual address.

The conditional branch instructions are PC-relative only. The 20-bit signed displacement gives a forward/backward branch distance of +/- 512K instructions.

The test is on the longword integer interpretation of the register contents. To test floating data, first compare the data with zero using CMPF or CMPG, and then branch on the result of the compare.

PC-relative unconditional branches can be performed by "BEQ R0,target".

Fault On Low Bit Clear

Format: Branch format

```
FLBC  Ra.rl,disp.al
```

Operation:

```
IF  Rav<0> EQ 0  THEN
    {FLBC exception}
```

Exceptions:

Fault On Low Bit Clear

Opcodes:

FLBC Fault On Low Bit Clear

Description:

Bit <0> of Register Ra is tested. If it is zero, a Fault On Low Bit Clear exception is generated (see Chapter 6, Exceptions and Interrupts, Section 6.4.3.2; otherwise, execution continues with the next sequential instruction.

The displacement field of this instruction may be used by software to code exception type information.

Jump to Subroutine

Format: Branch or Memory Format

```
JSR  Ra.wl,disp.al                    !Branch format
JSR  Ra.wl,(Rb.ab)                    !Memory format
```

Operation:

```
va <- PC + {4*SEXT(disp)}             !Branch format
va <- Rbv AND {NOT 3}                 !Memory format

Ra <- PC
PC <- va
```

Exceptions:

None

Opcodes:

JSR Jump to Subroutine

Description:

The PC of the instruction following the JSR instruction (the updated PC) is written to register Ra, followed by loading the PC with the target virtual address.

The JSR instruction has two formats: Branch and Memory.

In the Branch format, the displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign extended to 32 bits, and added to the updated PC to form the target virtual address.

In the Memory format, the new PC is supplied from register Rb and the displacement field Should Be Zero. The low two bits of the target address are ignored.

An unconditional jump can be performed by "JSR R0,target".

Co-routine linkage can be performed by specifying the same register in both the Ra and Rb operands.

4.7 MISCELLANEOUS INSTRUCTIONS

PRISM provides the following miscellaneous instructions:

Mnemonic	Operation
BPT	Breakpoint
DRAIN	Drain Instruction Pipeline
DRAINM	Drain Memory activity
DRAINV	Drain Vector Memory activity
IFLUSH	Flush I-Stream Cache
IOTA	Generate Compressed Iota Vector
MOVPS	Move Processor Status
PROBER	Probe Read Access
PROBEW	Probe Write Access
RDCC	Read Cycle Count Register
RDVC	Read Vector Count Register
RDVL	Read Vector Length Register
RDVMH	Read Vector Mask Register, High Part
RDVML	Read Vector Mask Register, Low Part
REI	Return from Exception or Interrupt
SWASTEN	Swap AST Enable
WRVC	Write Vector Count Register
WRVL	Write Vector Length Register
WRVMH	Write Vector Mask Register, High Part
WRVML	Write Vector Mask Register, Low Part

Breakpoint

Format: Epicode format

BPT

Operation:

{push current PC and PS on Kernel stack}

{push R5 and R4 on Kernel stack}

R4 <- Breakpoint SCB vector

{Change Mode to Kernel}

{dispatch through Breakpoint SCB vector}

Exceptions:

Kernel Stack Not Valid

Opcodes:

BPT Breakpoint

Description:

This instruction is provided for program debugging. It switches to Kernel mode and pushes the current PC, PS, R5, and R4 on the Kernel stack. It then copies the address in the Breakpoint SCB vector into R4 and dispatches to that address. See Chapter 6, Exceptions and Interrupts, Section 6.4.3.1.

Drain Instruction Pipeline

Format: Operate Format

```
DRAIN
DRAINM
DRAINV
```

Operation:

```
{Stall  instruction issuing until all              !DRAIN
prior instructions  are guaranteed to
complete without incurring exceptions.}

{Stall instruction issuing until all prior         !DRAINM
vector and scalar load/stores  have  passed
memory management checks and all pending cache
invalidates/updates have been processed by the
scalar and vector caches.
Stall all future vector and scalar load/store
instruction issuing  until  all  memory
references have been completed.}

If {multiple vector load/store paths to memory} THEN
    {Stall instruction issuing until all prior        !DRAINV
     vector load/stores  have  passed
     memory management checks.
     Stall future vector load/store instruction
     issuing until all vector memory references
     have been completed.}
```

Exceptions:

None

Opcodes:

```
DRAIN   Drain Instruction Pipeline
DRAINM  Drain Memory Pipeline
DRAINV  Drain Vector Memory Pipeline
```

Description:

The DRAIN instruction allows software to guarantee that, in a pipelined implementation, all previous instructions will complete without incurring exceptions before any more instructions are issued. For example, it should be used before changing an exception handler to ensure that all exceptions on previous instructions are processed in the current exception-handling environment.

The DRAIN instruction is not issued until all previous instructions are guaranteed to complete without exceptions. If an exception occurs, the continuation PC in the exception stack frame points to the DRAIN instruction.

The DRAINM instruction allows software to guarantee that, in an implementation allowing concurrent scalar/vector or vector/vector memory references, all previous memory references will not incur memory management exceptions after the DRAINM issues and also that all prior memory reads and writes have completed before any more

load/store instructions are issued. DRAINM also ensures that all pending cache invalidates/updates have been processed by the scalar and vector caches.

DRAINM should be used in the following cases when there is a possibility that the scalar and vector instructions could reference the same memory location:

- A scalar store followed by a vector load or store.
- A scalar load followed by a vector store.
- A vector store followed by a scalar load or store.
- A vector load followed by a scalar store.

DRAINM can also be used between vector loads and vector stores that access the same memory location. However, DRAINV provides a more efficient means for synchronizing between vector load/stores.

DRAINV should be used between vector loads and vector stores that access overlapping memory locations without any intervening operations that would prevent them from being executed concurrently in an implementation that has multiple vector load/store paths to memory. DRAINV can be implemented as a no-op in an implementation with a single vector load/store path to memory.

DRAINV also ensures that all pending cache invalidates/updates have been processed by the vector cache.

Chapter 9 specifies the rules for the synchronization between vector and scalar memory accesses.

Flush Instruction Cache

Format: Epicode format

IFLUSH

Operation:

{Invalidate instruction prefetch and instruction cache}

Exceptions:

None

Opcodes:

IFLUSH Flush Instruction Cache

Description:

An IFLUSH instruction must be executed when software or I/O devices write into the instruction stream. An implementation may contain an instruction cache that does not track either processor or I/O writes into the instruction stream. The instruction cache and any prefetched instructions are invalidated by an IFLUSH instruction.

The cache coherency and sharing rules are described in Chapter 9, System Architecture and Programming Implications.

Generate Compressed Iota Vector

Format: Masked Vector Arithmetic Operate Format

```
IOTA    Ra.rl,Vc.wl
IOTA    #a.ib,Vc.wl
```

Operation:

```
j   <- 0
tmp <- 0
FOR i <- 0 TO VL-1
    BEGIN
    IF  VM<i> EQ S  THEN
        BEGIN
        Vc[j]<31:0> <- tmp
        j <- j + 1
        END
    tmp <- tmp + Rav
    END
VC <- j                                  !return vector count
```

Exceptions:

Vector Enable

Opcodes:

IOTA Generate Compressed Iota Vector

Description:

IOTA constructs a vector of offsets for use by the vector gather/scatter instructions VGATH and VSCAT.

IOTA first generates an iota vector of length VL using the stride operand in register Ra (or a literal). An iota vector is a vector whose first element is zero and whose subsequent elements are spaced by the stride increment. That is:

0*Rav, 1*Rav, 2*Rav, 3*Rav, ..., {VL-1}*Rav

The stride is a signed longword. Overflow is ignored in this calculation. The iota vector is then compressed using the contents of the Vector Mask register (VM). Elements of the iota vector for which the corresponding Vector Mask Register bit matches the Mask Selector (S) bit are written to contiguous elements of the destination vector register, Vc. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The number of elements written to Vc is returned in the Vector Count register (VC) for use as a vector length in subsequent operations. The values of the elements in the destination vector register between the new value of VC and VL are UNPREDICTABLE.

Refer to Section 3.3.3.1 for a description of the Mask Selector bit.

This instruction may be omitted in a subset implementation.

Move Processor Status

Format: Epicode format

MOVPS

Operation:

R4 <- PS

Exceptions:

None

Opcodes:

MOVPS Move Processor Status

Description:

MOVPS writes the Processor Status (PS) to register R4. The Processor Status is described in Chapter 6, Exceptions and Interrupts, Section 6.2.

Probe Memory Access

Format: Epicode format

```
PROBE
```

Operation:

```
! R4 contains the base address
! R5 contains the signed offset
! R6 contains the processor mode
! R7 receives the completion status
!     Bit <0>  <- 1 if success, 0 if failure
!     Bit <31:1> <- 0

first <- R4
last  <- R4+R5
probe_mode <- MAXU(R6<0>, PS<CM>)
IF  ACCESS(first, probe_mode) AND ACCESS(last, probe_mode)  THEN
    R7 <- 1
ELSE
    R7 <- 0
```

Exceptions:

Translation Not Valid

Opcodes:

```
PROBER  Probe for Read Access
PROBEW  Probe for Write Access
```

Description:

PROBE checks the read or write accessibility of the first and last byte specified by the base address and the signed offset; the bytes in between are not checked. System software must check all pages between the two bytes if they are to be accessed. If both bytes are accessible, PROBE returns one in R7; otherwise, PROBE returns zero. The Fault On Read and Fault On Write PTE bits are not checked. A Translation Not Valid exception is signaled only if the first level PTE is invalid.

The protection is checked against the less privileged of the modes specified by R6<0> and the Current Mode (PS<CM>). See Chapter 6, Exceptions and Interrupts, Section 6.2 for processor mode encodings.

PROBE is only intended to check a single datum for accessibility. It does not check all intervening pages because this could result in excessive interrupt latency.

Read Cycle Count Register

Format: Epicode Format

RDCC

Operation:

QR4 <- CCR

Exceptions:

None

Opcodes:

RDCC Read Cycle Count Register

Description:

RDCC reads the 64-bit Cycle Count register and writes it to registers R4 and R5.

Read/Write Vector Count Register

Format: Operate Format

```
RDVC    Rc.wl
WRVC    Ra.rl
WRVC    #a.ib
```

Operation:

```
Rc <- ZEXT(VC)                          !RDVC

VC <- Rav<6:0>                          !WRVC
```

Exceptions:

Vector Enable

Opcodes:

```
RDVC    Read Vector Count Register
WRVC    Write Vector Count Register
```

Description:

RDVC reads the 7-bit Vector Count register and writes it zero extended to register Rc.

WRVC writes Rav<6:0> to the Vector Count register.

The Vector Count register is also written as a result of executing the IOTA instruction.

These instructions may be omitted in a subset implementation.

Read/Write Vector Length Register

Format: Operate Format

```
RDVL    Rc.wl
WRVL    Ra.rl
WRVL    #a.ib
```

Operation:

```
Rc <- ZEXT(VL)                          !RDVL

VL <- Rav<6:0>                          !WRVL
```

Exceptions:

Vector Enable

Opcodes:

```
RDVL    Read Vector Length Register
WRVL    Write Vector Length Register
```

Description:

RDVL reads the 7-bit Vector Length register and writes it zero extended to register Rc.

WRVL writes Rav<6:0> to the Vector Length register. Writing a value greater than 64 produces UNPREDICTABLE results.

These instructions may be omitted in a subset implementation.

Read/Write Vector Mask Register

Format: Operate Format

```
RDVM    Rc.wl
WRVM    Ra.rl
WRVM    #a.ib
```

Operation:

```
Rc <- VM<63:32>                          !RDVMH
Rc <- VM<31:0>                           !RDVML

VM<63:32> <- Rav                         !WRVMH
VM<31:0> <- Rav                          !WRVML
```

Exceptions:

Vector Enable

Opcodes:

RDVMH	Read Vector Mask Register, High Part
RDVML	Read Vector Mask Register, Low Part
WRVMH	Write Vector Mask Register, High Part
WRVML	Write Vector Mask Register, Low Part

Description:

RDVM reads the high or low 32 bits of the 64-bit Vector Mask register and writes them to register Rc.

WRVM writes the high or low 32 bits of the 64-bit Vector Mask register from register Ra or a literal.

These instructions may be omitted in a subset implementation.

Return from Exception or Interrupt

Format: Epicode format

```
REI
```

Operation:

```
IF  SP<2:0> NE 0  THEN
    {Illegal Operand exception}

tmp1 <- (SP)              !pick up saved PS
tmp2 <- (SP+4)            !pick up saved PC

IF  PS<CM> NE 0  THEN
    BEGIN
    IF  {tmp1<CM> EQ 0} OR
        {tmp1<VMM> NE 0} OR
        {tmp1<MBZ> NE 0} OR
        {tmp1<IPL> NE 0}  THEN
        {Illegal Operand exception}

    tmp1<VEN> <- tmp1<VEN> AND PS<VEN>
    END

IF  tmp1<VRF> EQ 1  THEN
    BEGIN
    tmp1<VRF> <- 0
    tmp2 <- (SCBB + Vector Restart Fault offset)
    END

SP <- SP + 8
IPR_SP[PS<CM>] <- SP
SP <- IPR_SP[tmp1<CM>]          !switch stack

PC <- tmp2 AND {NOT 3}
PS <- tmp1

{check for pending ASTs or interrupts}
```

Exceptions:

```
Access Violation
Fault on Read
Illegal Operand
Kernel Stack Not Valid
Translation Not Valid
```

Opcodes:

```
REI     Return from Exception or Interrupt
```

Description:

The PS and PC are popped from the current stack and held in temporary PS and PC registers. The new PS is checked for validity and consistency. If <VRF> is set in the new PS then REI will initiate a vector restart fault by dispatching through the vector restart fault vector in the SCB. See Chapter 6, Exceptions and Interrupts, Section 6.4.7.1 for details. The current stack pointer is saved and a new stack pointer is selected according to the new PS<CM> field. A check is made to determine if an AST or interrupt is pending (see Chapter 6, Exceptions and Interrupts, Section 6.7.5).

If the enabling conditions are present for an interrupt at the completion of this instruction, the interrupt occurs before the next instruction.

Notes:

1. \This instruction differs from the VAX REI instruction in that instruction lookahead in the processor is NOT re-initialized. Also, there is no interrupt stack and in Kernel mode the checks are eliminated.\

2. The low two bits of the new PC are ignored.

Swap AST Enable

Format: Epicode format

```
SWASTEN
```

Operation:

```
tmp <- R4<0>
R4 <- ZEXT(ASTEN<PS<CM>>)
ASTEN<PS<CM>> <- tmp

{check for pending ASTs}
```

Exceptions:

None

Opcodes:

SWASTEN Swap AST Enable for Current Mode

Description:

SWASTEN swaps the AST enable bit for the current mode. The new state for the enable bit is supplied in register R4<0> and previous state of the enable bit is returned, zero extended, in R4.

A check is made to determine if an AST is pending (see Chapter 6, Exceptions and Interrupts, Section 6.7.5.4).

If the enabling conditions are present for an interrupt at the completion of this instruction, the interrupt occurs before the next instruction.

4.8 PRIVILEGED INSTRUCTIONS

Privileged instructions are allowed in Kernel mode only; otherwise, a Privileged Instruction exception occurs. The following privileged instructions are provided:

Mnemonic	Operation
BOOT	Boot Processor
CMPSWQIP	Compare and Swap Quadword, Interlocked, Physical
HALT	Halt Processor
LDQP	Load Quadword Physical
MFPR	Move From Processor Register
MTPR	Move To Processor Register
STQP	Store Quadword Physical
SWPCTX	Swap Privileged Context
SWIPL	Swap IPL
TBFLUSH	Flush Translation Buffer
WRCC	Write Cycle Count Register

Boot

Format: Epicode format

BOOT

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}
ELSE
    {boot}
```

Exceptions:

Privileged Instruction

Opcodes:

BOOT Boot Processor

Description:

The BOOT instruction initiates a bootstrap sequence. See Chapter 11, System Bootstrapping and Console, Section 11.2.2.

Compare and Swap Quadword, Interlocked, Physical

Format: Epicode format

```
CMPSWQIP
```

Operation:

```
! QR4 contains address of comparand 1
! QR6 contains comparand 2
! QR8 contains swap value
! R4<0> receives the swap status

IF PS<CM> NE 0 THEN
    {privileged instruction exception}
addr <- QR4 AND {NOT 7}

tmp <- (addr){interlocked}      !acquire hardware interlock.

IF tmp EQ QR6 THEN
    BEGIN
    (addr){interlocked} <- QR8  !release hardware interlock
    R4 <- addr<31:0> OR 1       !set successful compare status
    END
ELSE
    BEGIN
    (addr){interlocked} <- tmp  !release hardware interlock
    QR6 <- tmp
    END
END
```

Exceptions:

```
Machine Check
Privileged Instruction
```

Opcodes:

```
CMPSWQIP  Compare and Swap Quadword, Interlocked, Physical
```

Description:

The quadword aligned memory operand, whose physical address is in QR4, is fetched and compared to QR6. The low 3 bits of the operand address in QR4 are ignored. If the two operands are equal, QR8 is written to the memory location and a swap status flag (R4<0>) is set. If the two operands are not equal, the original memory operand is written into QR6 and the memory location is left unchanged.

This instruction performs an interlocked memory access in that no other processor or I/O device can perform an interlocked operation on the same operand until the current interlocked operation has completed.

The operation is UNDEFINED if CMPSWQIP accesses I/O space. A reference to non-existent memory causes a Machine Check exception. Unimplemented physical address bits are SBZ; the operation is UNDEFINED if any of these bits are set.

Halt

Format: Epicode format

```
HALT
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}

CASE {halt_action} OF
        halt:                   {halt}
        restart/halt:           {restart/halt}
        restart/boot/halt:      {restart/boot/halt}
        boot/halt:              {boot/halt}
END
```

Exceptions:

Privileged Instruction

Opcodes:

HALT Halt Processor

Description:

The **HALT** instruction stops normal instruction processing, and depending on the **HALT** action setting, the processor may either enter console mode or the restart sequence. See Chapter 11, System Bootstrapping and Console, Section 11.2.2.

Load Quadword Physical

Format: Epicode format

```
LDQP
```

Operation:

```
! QR4 contains the quadword aligned physical address
! QR6 receives the data from memory

IF PS<CM> NE 0 then
    {Privileged Instruction exception}

addr <- QR4 AND {NOT 7}
QR6 <- (addr)<63:0>
```

Exceptions:

Privileged Instruction

Opcodes:

LDQP Load Quadword Physical

Description:

The quadword aligned memory operand, whose physical address is in QR4, is fetched and written to QR6.

Move From Processor Register

Format: Epicode format

```
MFPR   IPR_Name
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}

{result <- IPR specific function}

! IPR specific results are returned in R4, R5, and R6.
```

Exceptions:

Privileged Instruction

Opcodes:

MFPR Move From Processor Register

Description:

The internal processor register specified by the Epicode function field is written to the IPR-specific scalar register(s). Processor registers are implemented such that any side effects that may happen as a result of reading the register (e.g. an interrupt request is cleared) are guaranteed to occur exactly once.

See Chapter 8, Internal Processor Registers, for a description of each IPR.

Move To Processor Register

Format: Epicode format

```
MTPR    IPR_Name
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}

! R4 and R5 contain IPR specific source operands

{IPR <- result of IPR specific function}
```

Exceptions:

Privileged Instruction

Opcodes:

MTPR Move To Processor Register

Description:

The IPR-specific source operands in scalar registers R4 and R5 are written to the internal processor register specified by the Epicode function field. The effect of loading a processor register is guaranteed to be active on the next instruction.

See Chapter 8, Internal Processor Registers, for a description of each IPR.

Store Quadword Physical

Format: Epicode format

```
STQP
```

Operation:

```
! QR4 contains the quadword aligned physical address
! QR6 contains the data to be written

IF PS<CM> NE 0 then
    {Privileged Instruction exception}

addr <- QR4 AND {NOT 7}
(addr) <- QR6
```

Exceptions:

Privileged Instruction

Opcodes:

STQP Store Quadword Physical

Description:

The quadword contents of QR6 are written to the memory location, whose physical address is in QR4.

Swap Privileged Context

Format: Epicode format

```
SWPCTX
```

Operation:

```
! QR4 contains the physical address of the new HWPCB.

IF  PS<CM> NE 0  THEN
    {privileged instruction exception}

! Store old HWPCB contents

(IPR_PCBB + HWPCB_KSP) <- SP
IF  {internal registers for stack pointers}  THEN
    (IPR_PCBB + HWPCB_USP) <- IPR_USP

(IPR_PCBB + HWPCB_ASTSR) <- IPR_ASTSR
(IPR_PCBB + HWPCB_ASTEN) <- IPR_ASTEN
(IPR_PCBB + HWPCB_CCR) <- IPR_CCR

! Load new HWPCB contents

IPR_PCBB <- QR4 AND {NOT 7}

IF  {if ASN's not implemented in virtual instruction cache}
THEN
    {flush instruction cache}

IF  {ASNs not implemented in TB} THEN
    IF  {IPR_PTBR NE (IPR_PCBB + HWPCB_PTBR)}  THEN
        {invalidate trans. buffer entries with PTE<ASM> EQ 0}
ELSE
    IPR_ASN <- (IPR_PCBB + HWPCB_ASN)

SP <- (IPR_PCBB + HWPCB_KSP)

IF {internal registers for stack pointers}  THEN
    IPR_USP <- (IPR_PCBB + HWPCB_USP)

IPR_PTBR  <- (IPR_PCBB + HWPCB_PTBR)
IPR_ASTSR <- (IPR_PCBB + HWPCB_ASTSR)
IPR_ASTEN <- (IPR_PCBB + HWPCB_ASTEN)
IPR_CCR <- (IPR_PCBB + HWPCB_CCR)
```

Exceptions:

```
Machine Check
Privileged Instruction
```

Opcodes:

```
SWPCTX  Swap Privileged Context
```

Description:

The SWPCTX instruction returns ownership of the current Hardware Privileged Context Block (HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.

SWPCTX saves the privileged context from the internal processor registers into the HWPCB specified by the physical address in the PCBB internal processor register. It then loads the privileged context from the new HWPCB specified by the physical address in QR4. Note that the actual sequence of the save and restore operation is not specified so any overlap of the current and new HWPCB storage areas produces UNDEFINED results.

The privileged context includes the two stack pointers, the Page Table Base Register (PTBR), the Address Space Number (ASN), the AST enable and summary registers, and the Cycle Count Register. However, PTBR is never saved in the HWPCB and it is UNPREDICTABLE whether or not ASN is saved. These values cannot be changed for a running process. The process scalar and vector registers are saved and restored by the operating system. See Chapter 7, Process Structure, Figure 7-1, for the HWPCB format.

Any change to the current HWPCB while the processor has ownership results in UNDEFINED operation. All the values in the current HWPCB can be read through IPRs.

If the enabling conditions are present for an interrupt at the completion of this instruction, the interrupt occurs before the next instruction.

Epicode sets up the PCBB at boot time to point to the HWPCB storage area in the Restart Parameter Block (RPB). See Chapter 11, System Bootstrapping and Console.

The operation is UNDEFINED if SWPCTX accesses I/O space.

A reference to non-existent memory causes a Machine Check exception. Unimplemented physical address bits are SBZ. The operation is UNDEFINED if any of these bits are set.

Notes:

Processors may keep a copy of each of the per-process stack pointers in internal registers. In those processors, SWPCTX stores the internal registers into the HWPCB. Processors that do not keep a copy of the stack pointers in internal registers, keep only the stack pointer for the current processor mode in SP and switch this with the HWPCB contents whenever the current processor mode changes.

\For performance reasons, it is strongly recommended that the TB and Virtual Instruction Cache not be flushed if the old and new PTBR are the same and ASNs are not implemented.\

Swap IPL

Format: Epicode format

```
SWIPL
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}
tmp <- R4<2:0>
R4  <- ZEXT(PS<IPL>)
PS<IPL> <- tmp

{check for pending ASTs or interrupts}
```

Exceptions:

Privileged Instruction

Opcodes:

SWIPL Swap Processor IPL level

Description:

SWIPL swaps the processor IPL level. The new IPL level is supplied in register R4<2:0> and the previous IPL level is returned in R4.

A check is made to determine if an AST or interrupt is pending (see Chapter 6, Exceptions and Interrupts, Section 6.7.5).

If the enabling conditions are present for an interrupt at the completion of this instruction, the interrupt occurs before the next instruction.

Flush Translation Buffer

Format: Epicode format

```
TBFLUSH
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}

{Invalidate all translation buffer entries}
```

Exceptions:

Privileged Instruction

Opcodes:

TBFLUSH Flush Translation Buffer

Description:

The TBFLUSH instruction is used to invalidate all TB entries and flush virtual data caches. To invalidate a single TB entry, use the MTPR TBIS instruction.

Write Cycle Count Register

Format: Epicode format

```
WRCC
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}
ELSE
    CCR <- QR4
```

Exceptions:

Privileged Instruction

Opcodes:

```
WRCC    Write Cycle Count Register
```

Description:

WRCC writes the quadword in registers R4 and R5 into the 64-bit Cycle Count register.

Revision History:

Revision 3.0, 26 April 1988

1. Change source operand of vector convert source operand from Va to Vb to match VAX vectors.
2. Add Write Intent option for VLD and VGATH.
3. Add BOOT instruction.
4. Add CMPSWLI, CMPSWQI, CMPSWQIP.
5. Add VMERGE and IOTA variants that operate on clear mask bits.
6. Delete BUGCHK, RMAQIP.
7. Add RMALI as a non-privileged instruction.
8. Add WRCC, RDCC instructions.
9. Add LDQP, STQP.
10. Eliminate coprocessor support.
11. Contents of destination vector elements are UNPREDICTABLE when a vector memory management exception occurs.
12. Vector loads/stores to I/O space are UNPREDICTABLE.
13. Change behavior of VCMP in masked mode.
14. Add masked mode for VLD, VST, VGATH, VSCAT.
15. Vc[i]<63:32> is UNPREDICTABLE for all instructions with longword result, except for logicals.
16. Remove Vector Integer Divide.
17. Specify result of CMPF, CMPG, CVTFL, CVTGL, VCVTFL, VCVTGL with floating reserved operand as source.
18. Clarify that strides and offsets can be positive, negative, or zero.
19. Add masked operate format.
20. Revise DRAIN/DRAINM.
21. Change mnemonics to use opcode qualifiers.

Revision 2.0, 24 June 1986.

1. Simplified subsetting rules.
2. Changed VLDL and VGATHL to zero bits <63:32> of the destination vector element.

3. Clarified parallelism allowed between scalar and vector memory references in VLDx, VGATHx, VSTx, and VSCATx instructions.

4. Removed MULH.

5. Copy bits <63:32> of Vb elements to Vc elements for vector Boolean operations rather than making them UNPREDICTABLE.

6. Zero bits <63:32> of Vc elements for 32 bit integer add, subtract and shift vector operations rather than making them UNPREDICTABLE.

7. Added unsigned integer vector compare, VCMPUx.

8. Removed remainder instructions, REM and VREM.

9. Added vector unsigned multiply instruction, VUMULH.

10. Changed FOB to Fault on Low Bit Clear, FLBC.

11. Changed DRAIN from Epicode format to operate format.

12. Added DRAINM instruction to serialize vector and scalar memory references.

13. Changed PROBE to use a one bit mode operand.

14. Changed REI:
    - Test for a non-zero PS<VMM> bit.
    - Dispatch to the Vector Restart Fault exception handler if PS<VRF> is set.
    - Removed Illegal Operand test on <VEN> and <VRF>.

15. Changed SWPCTX to not invalidate the translation buffer or virtual instruction cache when the old and new PTBR addresses are the same. Removed ESP and SSP save and restore.

16. Removed literal operand form from COPWR instruction.

17. Changed all the subtract instructions, except scalar longword subtract (SUB and SUBV), so the Ra operand is the minuend and the Rb operand is the subtrahend. This provides the operation, scalar minus vector. SUB and SUBV were left unchanged because the a-operand provides the literal and these operations are performed in a hardware unit separate from the other subtract operations.

18. Changed all the divide instructions so the Ra operand is the dividend and the Rb operand is the divisor. This provides the operation, scalar divided by vector. All divides were changed because on some implementations the divide hardware will be shared by all divide instructions.

Revision 1.0, 22 December 1985

1. Changed register width from 64 bits to 32 bits.

2. Changed Epicode parameter registers to R4-R9.
3. Changed instruction descriptions to use instruction fields.
4. Changed MOVx mnemonics to LD/ST.
5. Changed REI to match new privileged architecture.
6. Changed Unbiased rounding to VAX rounding.
7. Added RMAQI, Read, Mask, Add Quadword, Interlocked.
8. Added RMAQIP, Read, Mask, Add Quadword, Interlocked, Physical.
9. Added SWIPL, Swap IPL.
10. Added SWASTEN, Swap AST enable.
11. Added SWPCTX, Swap Privileged Context.
12. Added FOB, Fault On Low Bit Set.
13. Added UMULH, Unsigned 32-bit Multiply, Return High bits.
14. Added F_Floating operations.
15. Added floating-point exception error result.
16. Added vector registers and vector instructions.
17. Added Coprocessor instructions.
18. Eliminated sign extended byte and word loads.
19. Eliminated operate format loads and stores.
20. Eliminated Compare address instructions.
21. Eliminated ADDRC, Add and Return Carry.
22. Eliminated SUBRB, Subtract and Return Borrow.
23. Eliminated CMPUEQ, CMPUNE, Compare Unsigned Equality
24. Eliminated Convert Quad to Long,Word,Byte instructions.
25. Eliminated Directed roundings to Plus and Minus Infinity.
26. Eliminated Queue instructions.
27. Eliminated Change Mode instructions.
28. Eliminated USRCHK, User Check.
29. Eliminated Quadword parameter from BUGCHK.
30. Eliminated PROBEPx, Probe Previous Mode Read/Write.
31. Eliminated INTON/INTOFF.

32. Eliminated RDSP/WRTSP, Read and Write Stack Pointer.

33. Eliminated SWIS, SWKS, Switch to Interrupt/Kernel stack.

34. Eliminated PREFETCH.

35. Eliminated MOVCNT, MOVCYT, Move Count/Cycle Time.

Revision 0.0, 5 July 1985

1. First Review Distribution

CHAPTER 5

MEMORY MANAGEMENT

5.1 INTRODUCTION

Memory management consists of the hardware and software which control the allocation and use of physical memory. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time; see Chapter 7, Process Structure. PRISM uses memory protection and multiple address spaces to ensure that one process will not affect other processes or the operating system.

To further improve software reliability, two processor modes provide memory access control for privileged (kernel mode) and non-privileged (user mode) software. Protection is specified at the individual page level for data and instruction access. A page may be inaccessible or may have different accessibility for each processor mode. Accessibility can be read-only, read/write, or no access. Accessible pages can be restricted to have only data or instruction access.

A program uses virtual addresses to access its data and instructions. However, before these virtual addresses can be used to access memory, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. The processor utilizes this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management provides both memory protection and memory mapping mechanisms. The PRISM memory management architecture is designed to meet several goals:

- Provide a large address space for instructions and data.
- Allow programs to run on hardware with physical memory smaller than the virtual memory used.
- Provide convenient and efficient sharing of instructions and data.
- Allow sparse use of a large address space without excessive page table overhead.
- Contribute to software reliability.
- Provide independent execute, read and write access protection.

- Provide an efficient mechanism for controlled entry to privileged operating system functions.

5.2 VIRTUAL ADDRESS SPACE

A virtual address is a 32-bit unsigned integer which specifies a byte location within the virtual address space. The programmer sees a linear array of 4,294,967,296 bytes. The virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size is 8 Kbytes. Future implementations of PRISM may use page sizes ranging up to 64 Kbytes. System software should, therefore, allocate regions with differing protection on 64-Kbyte virtual address boundaries to ensure image compatibility across all PRISM implementations.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables, and saves the inactive (but used) parts of the virtual address space on external storage media.

The operating system must be mapped into the same part of the address space for every process.

5.2.1 Virtual Address Format

The PRISM processor generates a 32-bit virtual address for each instruction and operand in memory. The virtual address consists of two segment number fields, and a Byte Within Page field.

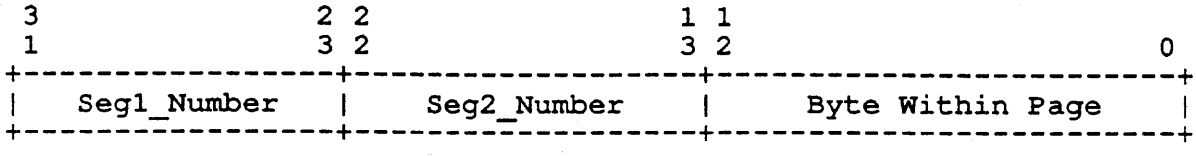


Figure 5-1: Virtual Address Format

The segment number fields, bits <31:13> of a virtual address, specify the virtual page to be referenced. The Byte Within Page field, bits <12:0> of a virtual address, specifies the byte offset within the page. A page contains 8 Kbytes.

5.3 PHYSICAL ADDRESS SPACE

Physical addresses are, at most, 45 bits. A processor may choose to implement a smaller physical address space by not implementing some number of high-order bits. The most significant implemented physical address bit selects memory space when it is 0, and I/O space when it is 1. For example, in a 30-bit physical address space, bit <29> selects memory or I/O space.

5.4 MEMORY MANAGEMENT CONTROL

Memory management is always enabled when the processor is not running Epicode. At processor initialization time, the processor executes Epicode with memory management disabled.

5.5 PAGE TABLE ENTRIES

The processor uses a quadword Page Table Entry (PTE) to translate virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical Page Frame Number.

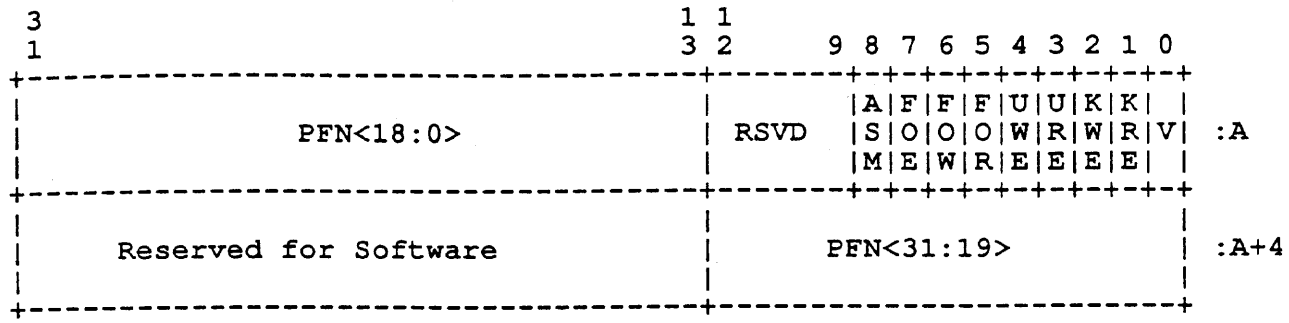


Figure 5-2: Page Table Entry

Fields in the page table entry are interpreted as follows:

Bits	Description
0	Valid (V) - Indicates the validity of the ASM, FOE, FOW, FOR bits and the PFN field. When V is set, the ASM, FOE, FOW, FOR bits and the PFN fields are valid for use by hardware. When V is clear, the PFN field is reserved for use by software. The V bit does not affect the validity of KRE, KWE, URE, and UWE.
1	Kernel Read Enable (KRE) - This bit enable reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V=0.
2	Kernel Write Enable (KWE) - This bit enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V=0.
3	User Read Enable (URE) - This bit enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.
4	User Write Enable (UWE) - This bit enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.

NOTE

If a write enable bit is set and the corresponding read enable bit is not, the operation of the processor is UNDEFINED.

5 Fault On Read (FOR) - When set, a Fault On Read exception occurs on an attempt to read any location in the page.

6 Fault On Write (FOW) - When set, a Fault On Write exception occurs on an attempt to write any location in the page.

7 Fault On Execute (FOE) - When set, a Fault On Execute exception occurs on an attempt to execute an instruction in the page.

8 Address Space Match (ASM) - When set, this PTE matches all Address Space Numbers. ASM must only be set for a virtual page if that page is shared among all processes in the system. If ASM is set in some but not all processes, the address mapping is UNPREDICTABLE.

12:9 Reserved for future use by DIGITAL.

12:10 Reserved for future use by DIGITAL.

44:13 Page Frame Number (PFN) - The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the Byte Within Page bits of the virtual address to obtain the physical address, see Section 5.7. If V is clear, this field may be used by software.

63:45 Reserved for software.

5.5.1 Changes To Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the valid bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation, e.g., setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in an even-odd register pair and then moving the new PTE to the page table using a Store Quadword instruction (STQ or STQP).

Multiprocessing makes the problem more complicated. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must either use the Read Mask and Add Quadword Interlocked (RMAQI) instruction, or use some other form of software

synchronization to modify PTEs that are already valid. Once a processor has modified a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their Translation Buffer. Software must inform other processors of changes to PTEs via the interprocessor interrupt mechanism and an associated software protocol. Since page table entries are shared data structures, modifications to them must follow the rules for shared data specified in Chapter 9.

5.6 MEMORY PROTECTION

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular processor mode. Access to each page is controlled by a protection code that specifies, for each processor mode, whether read or write references are allowed.

The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to index page tables.
- The intended access type (read data, write data, or instruction fetch).
- The current processor mode from the Processor Status.

If the access is allowed and the address can be mapped (the Page Table Entry is valid), the result is the physical address corresponding to the specified virtual address.

For protection checks, the intended access is Read for data loads and instruction fetch, and Write for data stores.

If an operand is an address operand, then no reference is made to memory. Hence, the page need not be accessible nor map to a physical page.

5.6.1 Processor Modes

There are 2 processor modes:

- Kernel
- User

The processor mode of a running process is stored in the Current Mode bit of the Processor Status (PS); see Chapter 6, Exceptions and Interrupts, Section 6.2.

5.6.2 Protection Code

Every page in the virtual address space is protected according to its

use. A program may be prevented from reading, or writing portions of its address space. Associated with each page is a protection code that describes the accessibility of the page for each processor mode. The code allows a choice of read or write protection for each processor mode.

- Each mode's access can be read/write, read-only, or no-access.
- Read and write accessibility are specified independently.
- The protection of each mode can be specified independently.

The protection code is specified by 4 bits in the PTE, see Section 5.5.

The PRISM architecture allows a page to be designated as execute only by setting the read enable bit for the processor mode and by setting the fault on read and write bits in the PTE.

5.6.3 Access Violation Fault

An Access Violation fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection field.

5.7 ADDRESS TRANSLATION

Address translation is performed by accessing entries in a two-level page table structure. The Page Table Base Register (PTBR) contains the physical Page Frame Number of the first-level page table. If part of any page table resides in I/O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

The Page Table Base Register contains the physical Page Frame Number of the highest-level (Segment 1) page table. Bits <31:23> of the virtual address are used to index into the first-level page table to obtain the physical Page Frame Number of the base of the second-level (Segment 2) page table. Bits <22:13> of the virtual address are used to index into the second level page table to obtain the physical Page Frame Number (PFN) of the page being referenced. The PFN is concatenated with virtual address bits <12:0> to obtain the physical address of the location being accessed.

If the first-level PTE is valid, the protection bits are ignored; the protection code in the second-level PTE is used to determine accessibility. If a first-level PTE is invalid, an Access Violation occurs if the PTE<KRE> equals zero. An Access Violation on a first-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

\Note that this mapping scheme does not require multiple contiguous physical pages. There are no length registers. Two pages (16 Kbytes) map 8 Mbytes of virtual address space; 513 pages (approximately 4 Mbytes) map the entire 4-Gbyte address space.\

The algorithm to generate a physical address from a virtual address is shown below:

```
seg1_pte <- ({PTBR * 8192} + {8 * VA<31:23>}) !Read Physical
IF seg1_pte<V> EQ 0 THEN
    IF seg1_pte<KRE> EQ 0 THEN
        {initiate Access Violation fault}
    ELSE
        {initiate Translation Not Valid fault}

seg2_pte <- ({seg1_pte<PFN> * 8192} + {8 * VA<22:13>})  !Read Physical
IF {{{seg2_pte<UWE> EQ 0} AND {write access} AND {PS<CM> EQ 1}} OR
    {{seg2_pte<URE> EQ 0} AND {read or execute access}
                              AND {PS<CM> EQ 1}} OR
    {{seg2_pte<KWE> EQ 0} AND {write access} AND {PS<CM> EQ 0}} OR
    {{seg2_pte<KRE> EQ 0} AND {read  or execute access}
                              AND {PS<CM> EQ 0}}}
    THEN
        {initiate Access Violation fault}
    ELSE
        IF seg2_pte<V> EQ 0 THEN
            {initiate Translation Not Valid fault}

IF {seg2_pte<FOW> EQ 1} AND {write access} THEN
    {initiate Fault On Write fault}
IF {seg2_pte<FOR> EQ 1} AND {read access} THEN
    {initiate Fault On Read fault}
IF {seg2_pte<FOE> EQ 1} AND {execute access} THEN
    {initiate Fault On Execute fault}

Physical_Address <- {seg2_pte<PFN> * 8192} OR VA<12:0>
```

5.8 TRANSLATION BUFFER

In order to save actual memory references when repeatedly referencing the same pages, a hardware implementation may include a translation buffer to remember successful virtual address translations and page states.

When the process context is changed, a new value is loaded into the Address Space Number (ASN) internal processor register with a Swap Privileged Context instruction (SWPCTX); see Chapter 4, Instruction Descriptions, Page 4-98 and Chapter 7, Process Structure. This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers. Additionally, when the software changes any part (except for the Software field) of a valid Page Table Entry, it must also move a virtual address within the corresponding page to the Translation Buffer Invalidate Single (TBIS) internal processor register with the MTPR instruction; see Chapter 8, Internal Processor Registers, Page 8-25.

\Some implementations may invalidate the entire Translation Buffer on an MTPR to TBIS. In general, implementations may invalidate more than the required translations in the TB.\

The entire Translation Buffer can be invalidated by executing a Translation Buffer Flush instruction (TBFLUSH); see Chapter 4, Instruction Descriptions, Page 4-101.

The Translation Buffer must not store invalid PTEs. Therefore, the software is not required to invalidate Translation Buffer entries when making changes for PTEs that are already invalid.

The TBCHK internal processor register is available for interrogating the presence of a valid translation in the Translation Buffer; see Chapter 8, Internal Processor Registers, Page 8-23.

\Hardware implementors should be aware that a single, direct mapped TB has a potential problem when a load/store instruction and its data map to the same TB location. If TB misses are handled in Epicode, there could be an endless loop unless the instruction is held in an instruction buffer or a translated physical PC is maintained by the hardware.\

5.9 ADDRESS SPACE NUMBERS

The PRISM architecture allows a processor to optionally implement 16-bit address space numbers (process tags) to reduce the need for invalidation of cached address translations for process specific addresses when a context switch occurs. The address space number for the current process is loaded by software in the Address Space Number (ASN) internal processor register with a Swap Privileged Context instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

\There are several possible ways of using ASNs. There are several complications in a multiprocessor system. Consider the case where a process that executed on processor-1 is rescheduled on processor-2. If a page is deleted or its protection is changed, the TB in processor-1 has stale data. One solution would be to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. This results in significant overhead in a system with several processors. Another solution would be to have software invalidate all TB entries for a process on a new processor before it can begin execution, if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor. \

5.10 MEMORY MANAGEMENT FAULTS

Five types of faults are associated with memory access and protection:

- Access Violation
- Fault On Read

- Fault On Write
- Fault On Execute
- Translation Not Valid

See Chapter 6, Exceptions and Interrupts, for a detailed description of these faults.

An Access Violation (ACV) fault is taken when the protection field of the second-level PTE that maps the data indicates that the intended page reference would be illegal in the specified processor mode. An Access Violation fault is also taken if the KRE bit is zero in an invalid first level PTE.

A Fault On Read (FOR) fault occurs when a read is attempted with PTE<FOR> set. A Fault On Write (FOW) fault occurs when a write is attempted with PTE<FOW> set. A Fault On Execute (FOE) fault occurs when instruction execution is attempted with PTE<FOE> set.

A Translation Not Valid (TNV) fault is taken when a read or write reference is attempted through an invalid PTE in a first- or second-level page table.

Note that these five faults have distinct vectors in the System Control Block. The Access Violation fault takes precedence over Translation Not Valid, and Fault On Read/Write/Execute. Translation Not Valid, and Fault On Read/Write/Execute are mutually exclusive. Fault On Read and Fault On Write can occur simultaneously in the RMAQI, RMALI, CMPSWLI, and CMPSWQI instructions, in which case the order that the exceptions are taken in is UNPREDICTABLE.

Revision History:

Revision 3.0, 26 April 1988.

1. Eliminate DCV.
2. Clarify ASM.
3. Change access mode to processor mode.
4. Clarify address translation algorithm to show execute access.

Revision 2.0, 24 June 1986.

1. Reduce processor modes to kernel and user only.
2. Eliminate indirect PTE's.
3. Eliminate execute protection.
4. Change protection to independent read and write enables for user and kernel modes.
5. Change PTE format to reduce TB fill time.

Revision 1.0, 22 December 1985.

1. Change virtual address to 32 bits.
2. Simplify PTE format. Eliminate M, and COM in favor of Fault On Read/Write/Execute. Eliminate skip bits in PTE.
3. Eliminate system space.
4. Change page size to 8 Kbytes
5. Change protection change boundary to 64 Kbytes
6. Move exception frames to Chapter 6.

Revision 0.0, Initial Release, 5 July 1985.

CHAPTER 6

EXCEPTIONS AND INTERRUPTS

6.1 INTRODUCTION

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an event occurs, the processor forces a change in control flow from that indicated by the current instruction stream.

Some of the events are relevant primarily to the currently executing process, and normally invoke software in the context of the current process. The notification of such events is termed an exception.

Other events are primarily relevant to other processes, or to the system as a whole, and are therefore serviced in a system-wide context. The notification for these events is termed an interrupt.

Some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time.

6.1.1 Processor Interrupt Priority Level (IPL)

The processor has eight Interrupt Priority Levels (IPL's) divided into four software levels (numbered 0 to 3), and four hardware levels (numbered 4 to 7). User applications and most operating system software run at IPL 0, which may be thought of as process level. Higher numbered interrupt levels have higher priority; i.e., any request at an interrupt level higher than the processor's current IPL will interrupt immediately, but requests at lower or equal levels are deferred.

Interrupt levels 0 to 3 exist solely for use by software. No hardware event can request an interrupt on these levels. Conversely, interrupt levels 4 to 7 exist solely for use by hardware. Software cannot request an interrupt at any of these levels.

6.1.2 Interrupts

The processor arbitrates interrupt requests according to priority. When the priority of an interrupt request is higher than the current processor IPL, the processor will raise the IPL and service the

interrupt request. The interrupt service routine is entered at the IPL of the interrupting source and does not usually change the IPL set by the processor. Interrupt requests can come from I/O Devices, memory controllers, other processors, or the processor itself.

The priority level of one processor does not affect the priority level of other processors. Thus, in a multiprocessor system, interrupt levels alone cannot be used to synchronize access to shared resources. Even the various urgent interrupts, including those exceptions that run at IPL 7, do so on only one processor.

Synchronization with other processors in a multiprocessor system involves a combination of raising the IPL and executing an interlocking instruction sequence. Raising the IPL prevents the synchronization sequence itself from being interrupted on a single processor, while the interlock sequence guarantees mutual exclusion with other processors.

6.1.3 Exceptions

Most exception service routines execute at the current processor IPL in response to exception conditions caused by software. Serious system failures such as a machine check, however, raise the IPL to the highest level (7) to minimize processor interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur. All exceptions are handled in kernel mode.

There are three types of exceptions:

- A fault is an exception condition that occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent re-execution of the instruction will give correct results. Faults are not guaranteed to leave the machine in exactly the same state it was in immediately prior to the fault, but rather in a state such that the instruction can be correctly executed if the fault condition is removed.

- An abort is an exception condition that occurs during an instruction and potentially leaves the registers and memory in an indeterminate state such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone.

- A trap is an exception condition that occurs at the completion of the operation that caused the exception. Since several instructions may be in various stages of execution at any point in time, it is possible for multiple traps to occur simultaneously. The next instruction address that is reported on traps is that of the next instruction that would have issued if the trapping condition had not occurred. This is not necessarily the address of the instruction immediately following the one encountering the trap condition. Therefore, in general, it is difficult to fix up results and continue program execution at the point of the trap. Software can force a trap to be continued more easily without the need for complicated fix-up code. This is accomplished

by placing a Drain (DRAIN) instruction immediately after the instruction whose possible trap is to be made continuable; see Chapter 4, Instruction Descriptions, Page 4-77.

For example:

```
MULG    R4,R6,R8
DRAIN
```

In this example, no further instructions are allowed to issue until the MULG has completed and any possible trap has been initiated.

6.1.4 Contrast Between Exceptions And Interrupts

Generally, exceptions and interrupts are similar. However, there are four important differences:

1. An exception condition is caused by the execution of an instruction while an interrupt is caused by some activity in the system that may be independent of any instruction.

2. The IPL of the processor is usually not changed when the processor initiates an exception, while the IPL is always raised when an interrupt is initiated.

3. Exceptions are always initiated immediately, no matter what the processor IPL is, while interrupts are deferred until the processor IPL drops below the IPL of the requesting source.

4. Some exceptions can be selectively disabled by selecting instructions that do not check for exception conditions. If an exception condition occurs when checking is disabled, the exception will not occur on a subsequent instruction that does check such conditions. If an interrupt request occurs while the processor IPL is equal to or greater than that of the interrupting source, the condition will eventually initiate an interrupt if the interrupt request is still present and the processor IPL is lowered below that of the interrupting source.

6.2 PROCESSOR STATE

Processor state consists of a longword called the Processor Status (PS) and a longword containing the Program Counter (PC), which is the 32-bit virtual address of the next instruction.

When either an exception or interrupt is initiated, the current processor state must be preserved. This is accomplished by automatically pushing the PC, followed by the PS, on the Kernel stack. Subsequently, instruction execution can be continued at the point of the exception or interrupt by executing a Return from Exception or Interrupt (REI) instruction; see Chapter 4, Instruction Descriptions, Page 4-87.

\Initiation of an exception or interrupt causes the PC, followed by the PS, to be pushed on the Kernel stack. This is opposite to VAX which pushes PSL followed by PC. We want to allow for the possibility of future machines being 64-bits with a 32-bit compatibility mode. Pushing PS last allows Epicode to test a mode bit in the PS and determine the format of the PS and PC that were pushed on the stack.\

Process context such as the mapping information is not saved or restored on each interrupt or exception. Instead, it is saved and restored when process context switching is performed. Other processor status is changed even less frequently; see Chapter 7, Process Structure.

The PS can be explicitly stored with the Move Processor Status (MOVPS) instruction; see Chapter 4, Instruction Descriptions, Page 4-81. The PC can be explicitly stored with the Jump to Subroutine (JSR) instruction. All branching instructions also load a new value into the PC; see Chapter 4, Instruction Descriptions, Pages 4-74 and 4-72.

The terms current PS and saved PS are used to distinguish between this status information when it is stored internal to the processor and when copies of it are materialized in memory.

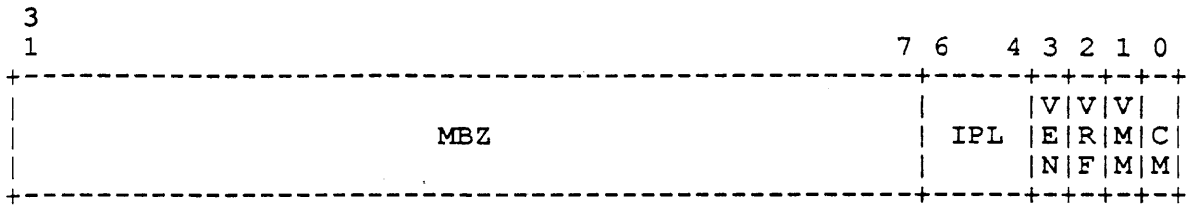


Figure 6-1: Processor Status

Bits	Description
0	Current Mode (CM). The processor mode of the currently executing process as follows: 0 - Kernel 1 - User
1	Virtual Machine Monitor (VMM) - When set, the processor is executing in a virtual machine monitor. When clear, the processor is running in either real or virtual machine mode. \This bit is only meaningful when running with epicode that implements virtual machine capabilities.\
2	Vector Restart Frame (VRF) - This bit can only be set in a PS which has been saved during the initiation of an exception. When set, a vector restart frame has been pushed on the stack prior to the saved PS and PC.
3	Vector Enable (VEN) - This bit controls whether vector instructions can be executed. When this bit is set, vector instructions execute normally. When this bit is clear, an attempt to issue a vector instruction causes a Vector Enable fault.
6:4	Interrupt Priority Level (IPL) - The current processor priority, in the range 0 to 7.
31:7	Reserved to DIGITAL, MBZ.

At bootstrap, the initial value of PS is set to 70 (hex). VRF, VEN, VMM, and CM are clear and IPL is 7.

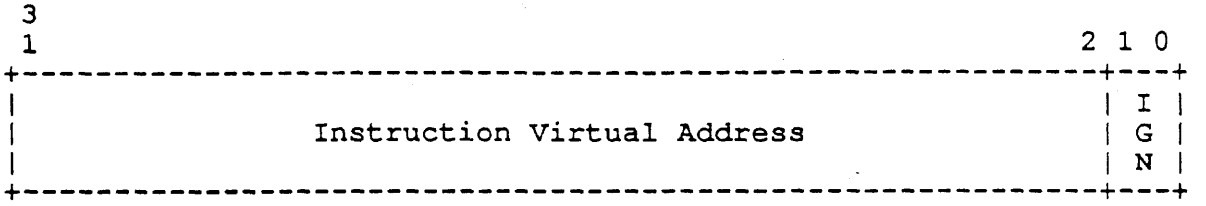


Figure 6-2: Program Counter

All instructions are aligned on longword boundaries and, therefore, hardware can assume zero for the two low-order PC bits.

6.3 INTERRUPTS

In some implementations, several instructions may be in various stages of execution simultaneously. Before the processor can service an interrupt request, all active instructions must be allowed to complete without exception (e.g., an exception could occur in a currently active instruction, in which case the exception would be initiated before the interrupt).

The following events will cause an interrupt:

- Asynchronous System Trap (AST) - IPL 1.
- Software interrupts - IPL 1 to 3.
- Console interrupts - IPL 4.
- I/O Device interrupts - IPL 4 and 5.
- 1 ms Interval Clock interrupt - IPL 6.
- Interprocessor interrupt - IPL 6.
- Power Recovery interrupt - IPL 7.
- Machine Check exception/interrupt - IPL 7.

Each interrupt source has a separate vector location (offset) within the System Control Block (SCB); see Section 6.6. The vector location for architecturally defined interrupts is fixed by the architecture. The vector location contains 2 longwords. The first longword contains the virtual address of the interrupt service routine. The second longword contains a interrupt service routine parameter.

When an interrupt is serviced, the PC, PS, R5, and R4 are pushed on the Kernel stack, the contents of the first longword in the SCB vector location are copied into R4 and the PC, the contents of the second longword in the SCB vector location are copied into R5, and instruction execution is initiated in Kernel Mode.

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Therefore, the instructions, data, and the contents of the interrupt vector for the interrupt service routine must be present in every process at the same virtual address.

Interrupt service routines should follow the discipline of not lowering IPL below their initial level. Lowering IPL in this way could result in a delay in the processing of more urgent interrupts at an intermediate level.

Kernel Mode software may need to raise and lower IPL during certain instruction sequences that must synchronize with possible interrupt conditions (e.g., Power Recovery). This can be accomplished by specifying the desired IPL and executing a Swap IPL instruction (SWIPL) or by executing an REI instruction that restores a PS that contains the desired IPL; see Chapter 4, Instruction Descriptions, Pages 4-100 and 4-87.

6.3.1 Asynchronous System Trap (AST) - Level 1

Asynchronous System Traps are a means of notifying a process of events that are not synchronized with its execution, but which must be dealt with in the context of the process. An Asynchronous System Trap is initiated when an REI instruction restores a PS with a Current Mode that is less privileged than or equal to a mode for which an AST is pending and not disabled; see Chapter 7, Process Structure, Section 7.3.

6.3.2 Software Interrupts - Levels 1 To 3

6.3.2.1 Software Interrupt Summary Register

The architecture provides three priority interrupt levels for use by software (level 0 is also available for use by software but interrupts can never occur at this level). The Software Interrupt Summary Register (SISR) stores a mask of pending software interrupts. Bit positions in this mask which contain a 1 correspond to the levels on which software interrupts are pending.

When the processor IPL drops below that of the highest requested software interrupt, a software interrupt is initiated and the corresponding bit in the SISR is cleared.

The SISR is a read-only internal processor register which may be read by Kernel Mode software by executing a Move From Processor Register instruction specifying SISR (MFPR SISR); see Chapter 8, Internal Processor Registers, Section 8.1.

6.3.2.2 Software Interrupt Request Register

The Software Interrupt Request Register (SIRR) is a write-only internal processor register used for making software interrupt requests.

Kernel Mode software may request a software interrupt at a particular level by executing a Move To Processor Register instruction specifying SIRR (MTPR SIRR); see Chapter 8, Internal Processor Registers, Section 8.1.

If the requested interrupt level is greater than the current IPL, the interrupt will occur before the execution of the next instruction. If, however, the requested level is equal to or less than the current processor IPL, the interrupt request will be recorded in the Software Interrupt Summary Register (SISR) and deferred until the processor IPL drops to the appropriate level.

Note that no indication is given if there is already a request at the specified level. Therefore, the respective interrupt service routine must not assume that there is a one-to-one correspondence between interrupts requested and interrupts generated. A valid protocol for generating this correspondence is:

1. The requester places information in a control block and then inserts the control block in a queue associated with the respective software interrupt level.

2. The requester uses MTPR SIRR to request an interrupt at the appropriate level.

3. The interrupt service routine attempts to remove a control block from the request queue. If there are no control blocks in the queue, the interrupt is dismissed with an REI.

4. If a valid control block is removed from the queue, the requested service is performed and Step 3 is repeated.

6.3.3 Console Interrupts - Level 4

NOTE

\A common console architecture for uniprocessor and multiprocessor systems is currently being defined. The Console architecture is to subject to change.\

Console interrupts are requested, if enabled, as characters are received from and transmitted to the console terminal.

6.3.3.1 Console Receive Control Status

The Console Receive Control Status register (CRCS) is a read/write internal processor register used to enable and disable console receive interrupts. Console receive interrupts are used to synchronize the input of characters from the console terminal.

CRCS may be read by Kernel Mode software by executing a Move From Processor Register instruction specifying CRCS (MFPR CRCS). Kernel Mode software may write CRCS by executing a Move To Processor Register instruction specifying CRCS (MTPR CRCS). See Chapter 8, Internal Processor Registers, Section 8.1.

6.3.3.2 Console Transmit Control Status

The Console Transmit Control Status register (CTCS) is a read/write internal processor register used to enable and disable console transmit interrupts. Console transmit interrupts are used to synchronize the output of characters to the console terminal.

CTCS may be read by Kernel Mode software by executing a Move From Processor Register instruction specifying CTCS (MFPR CTCS). Kernel Mode software may write CTCS by executing a Move To Processor Register instruction specifying CTCS (MTPR CTCS). See Chapter 8, Internal Processor Registers, Section 8.1.

6.3.4 I/O Device Interrupts - Levels 4 And 5

The architecture provides two priority levels for use by I/O Devices.

I/O Device interrupts are requested when a completion, attention, or error condition is present in an I/O Device and the respective interrupt is enabled.

6.3.5 Urgent Interrupts - Levels 6 And 7

The architecture provides two priority levels for use by urgent conditions including serious errors (e.g., Machine Check), interprocessor interrupts, interval timer interrupts, and Power Recovery. Interrupts on these levels are initiated by the processor upon detection of certain conditions. Some of these conditions are not interrupts. For example, Machine Check is usually an exception but it runs at a high priority level.

Interrupt Level 7 is reserved for those conditions that must lock out all processing until handled. This includes the hardware "disaster" Machine Check and Power Recovery; See Section 6.4.6.2.

The Power Recovery interrupt is generated when power is restored after a power failure. The power-down sequence is handled totally in Epicode. After having saved volatile machine state in memory (e.g., scalar registers, vector registers, Epicode registers, writeback cache data, etc.), Epicode gracefully stops system operation in an implementation-dependent manner. When power is restored the system enters a restart sequence. At the end of the sequence, if successful, a Power Recovery interrupt is initiated; see Chapter 11, System Bootstrapping and Console, Section 11.1.3.

Even though the power-down sequence is handled totally in Epicode, it will not be initiated until the processor IPL drops below 7. Thus critical code sequences can block the power-down sequence by raising the IPL to 7. Software, however, must take extra care not to lock out the power-down sequence for an extended period of time.

\The time interval is TBS.\

Interrupt level 6 is reserved for interprocessor and interval timer interrupt requests.

6.3.5.1 Interval Clock Interrupt - Level 6

The 1ms Interval Clock requests an interrupt every millisecond if clock interrupts are enabled.

6.3.5.1.1 Interval Clock Interrupt Enable

The Interval Clock Interrupt Enable register (ICIE) is a read/write internal processor register used to enable and disable Interval Clock interrupts.

ICIE may be read by Kernel Mode software by executing a Move From Processor Register instruction specifying ICIE (MFPR ICIE). Kernel Mode software may write ICIE by executing a Move To Processor Register instruction specifying ICIE (MTPR ICIE). See Chapter 8, Internal Processor Registers, Section 8.1.

6.3.5.2 Interprocessor Interrupt - Level 6

Interprocessor interrupts are provided to enable operating system software running on one processor to interrupt activity on another processor and cause operating system dependent actions to be performed.

If the target processor is the same as the current processor, whether or not an interprocessor interrupt is initiated is UNPREDICTABLE.

6.3.5.2.1 Interprocessor Interrupt Enable Register

The Interprocessor Interrupt Enable register (IPIE) is a read/write internal processor register used to enable and disable interprocessor interrupts. Interprocessor interrupts are used in multiprocessing systems to notify other processors of state changes. When interprocessor interrupts are enabled, a processor can receive interrupts from other processors. Writes to this register may be ignored in a uniprocessor system.

The IPIE may be read by Kernel Mode software by executing a Move From Processor Register instruction specifying IPIE (MFPR IPIE). Kernel Mode software may write IPIE by executing a Move To Processor Register instruction specifying IPIE (MTPR IPIE); see Chapter 8, Internal Processor Registers, Section 8.1, Page 8-12.

Explicit state is not provided by the architecture for software to directly determine whether there was an outstanding interprocessor interrupt when powerfail occurred. It is the responsibility of software to leave sufficient information in memory so that it may determine the proper action on power-up. One such method would be for software to maintain an action or request queue for each processor. On power-up, software would examine the action/request queue for each processor and if the queue is not empty, request an interprocessor interrupt with the respective processor as the target.

6.3.5.3 Interprocessor Interrupt Request Register

The Interprocessor Interrupt Request Register (IPIR) is a write-only internal processor register used for making a request to interrupt a specific processor.

Kernel Mode software may request to interrupt a particular processor by executing a Move To Processor Register instruction specifying IPIR (MTPR IPIR); see Chapter 8, Internal Processor Registers, Section 8.1,, Page 8-13.

Note that, like software interrupts, no indication is given as to whether there is already an interprocessor interrupt pending when one is requested. Therefore, the interprocessor interrupt service routine must not assume there is a one-to-one correspondence between interrupts requested and interrupts generated. A valid protocol similar to the one for software interrupts for generating this correspondence is:

1. The requester places information in a control block and then inserts the control block in a queue associated with the target processor.

2. The requester uses MTPR IPIR to request an interprocessor interrupt on the target processor.

3. The interprocessor interrupt service routine on the target processor attempts to remove a control block from its request queue. If there are no control blocks remaining, the interrupt is dismissed with an REI.

4. If a valid control block is removed from the queue, the specified action is performed and Step 3 is repeated.

6.4 EXCEPTIONS

Exceptions can be grouped into seven categories:

1. Arithmetic traps

2. Data Alignment fault

3. Faults occurring as a consequence of an instruction

4. Memory management faults

5. Serious system failures

6. Vector exceptions

Each exception has a separate vector location (offset) within the System Control Block (SCB); see Section 6.6 below. The vector location contains 2 longwords. The first longword contains the virtual address of the exception service routine. The second longword contains a exception service routine parameter.

When initiating an exception, various parameters are pushed on the

Kernel stack. These parameters represent information that is necessary to process the respective exception. An even number of longwords is always pushed. Minimally, this consists of the processor state (PC and PS), R5, and R4, but can also include such things as virtual addresses and instruction values. If the number of parameters is not an even number of longwords, then a zero longword is pushed to ensure that the stack remains quadword aligned; see Section 6.4.4. After the parameters are pushed on the Kernel stack, the contents of the first longword in the SCB vector location are copied into R4 and the PC, the contents of the second longword in the SCB vector location are copied into R5, and instruction execution is initiated in Kernel Mode.

6.4.1 Arithmetic Traps

An arithmetic trap is an exception that occurs as the result of performing an arithmetic or conversion operation. In general, it is difficult to fix up results and continue from this type of exception. Software can, however, force an arithmetic trap to be continued more easily by placing a DRAIN instruction immediately following an instruction that can cause an arithmetic trap.

If scalar register R0 is specified as the destination of an operation that can cause an arithmetic trap, it is UNPREDICTABLE whether the trap will actually occur, even if the operation would definitely produce an exceptional result.

If a floating reserved operand is specified in an F_ or G_floating operation, then a floating reserved operand with ETYPE<FRS> set is reported for that operation. The one exception to this rule is F_ and G_floating division. For this case, if both a true zero divisor (not a floating reserved divisor) and a floating reserved dividend are specified, it is UNPREDICTABLE whether a floating reserved operand with ETYPE<FRS> set or a floating divide by zero with ETYPE<FDZ> set is reported.

In general it is permissible for an implementation to use a forwarded or bypassed result in a subsequent instruction, even if the result is exceptional. A floating exceptional result is always a reserved operand with the exception type encoded in the ETYPE field. An integer exceptional result is the low 32-bits of the true result.

Exceptional results can be forwarded to the address calculation for load and store instructions (scalar and vector), to the address calculation for jump to subroutine instructions, as the source data for a store instruction (scalar and vector), or as the source data for a conditional branch instruction. This can result in the generation of an inappropriate address, the storing of exceptional results in memory, or an unintended branch. If such an occurrence is possible, software should use a combination of range checking and DRAIN instructions to precisely isolate such programming errors.

Arithmetic traps are initiated in Kernel Mode and push the following information on the Kernel stack:

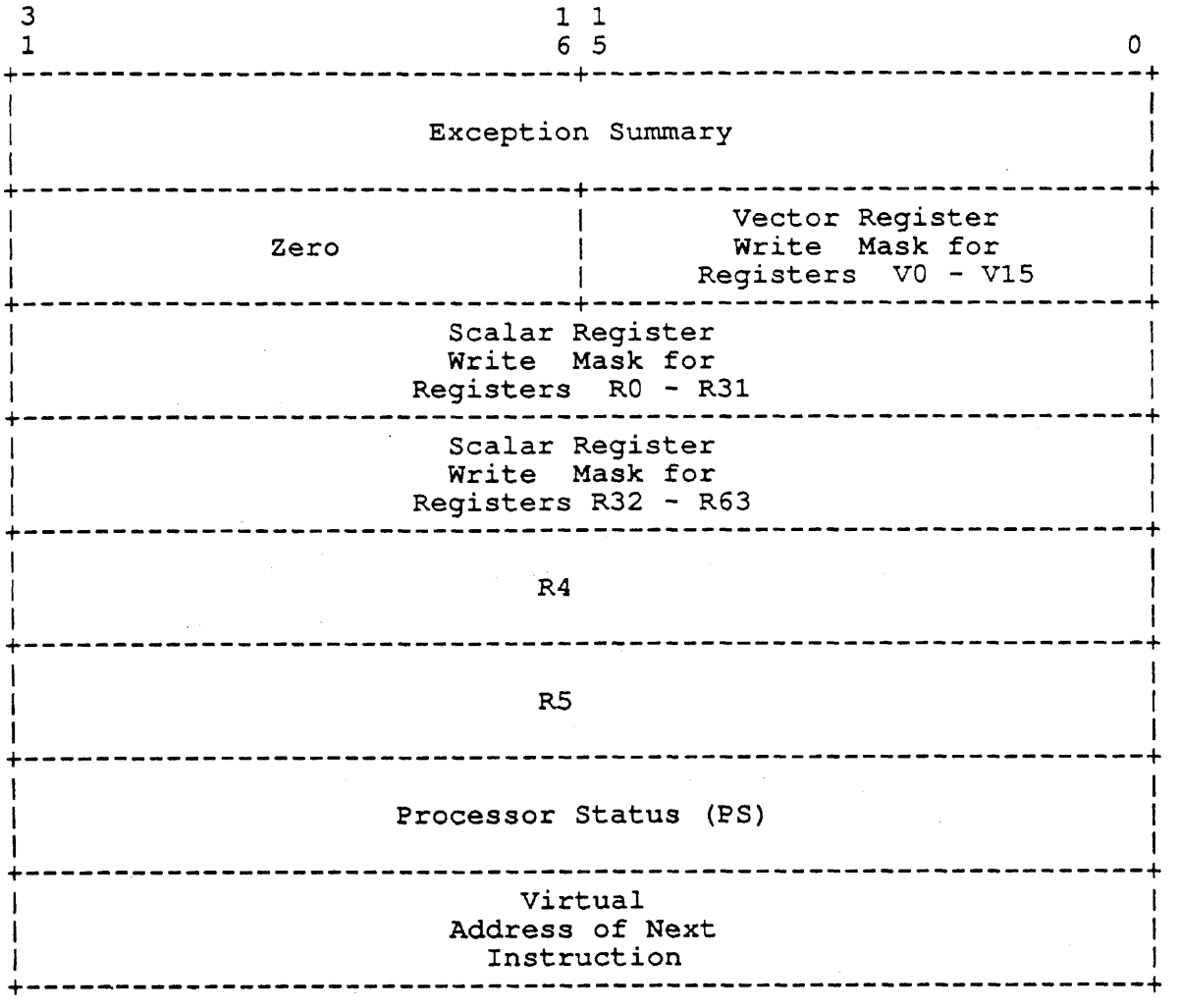


Figure 6-3: Arithmetic Trap Exception Frame

When an arithmetic exception condition is detected, several instructions may be in various stages of execution. These instructions are allowed to complete before the arithmetic exception can be initiated. Some of these instructions may themselves cause further arithmetic exceptions. Thus it is possible for several arithmetic exceptions to occur simultaneously.

The Exception Summary parameter records the various types of arithmetic exceptions that can occur together.

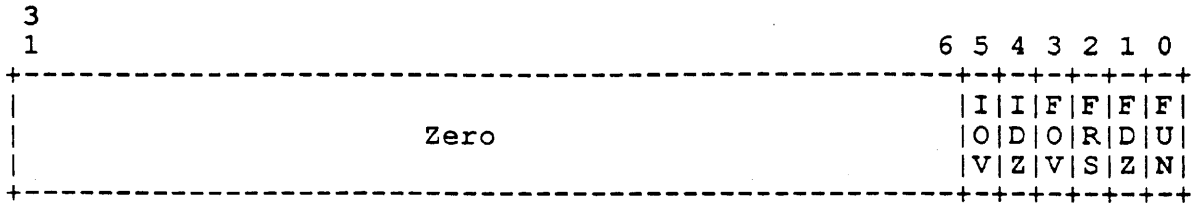


Figure 6-4: Exception Summary

Bit	Description
0	Floating Underflow (FUN) - An F_ or G_floating arithmetic or conversion operation underflowed the destination exponent.
1	Floating Divide by Zero (FDZ) - An attempt was made to perform an F_ or G_floating divide operation with a divisor of zero.
2	Floating Reserved Operand (FRS) - An attempt was made to perform an F_ or G_floating arithmetic, conversion, or comparison operation, and one or more of the operand values were reserved.
3	Floating Overflow (FOV) - An F_ or G_floating arithmetic or conversion operation overflowed the destination exponent.
4	Integer Divide by Zero (IDZ) - An attempt was made to perform an integer divide operation with a divisor of zero.
5	Integer Overflow (IOV) - An integer arithmetic operation or a conversion from F_ or G_floating to integer overflowed the destination precision.

The Vector Register Write Mask parameter records which vector registers were written with one or more elements containing exceptional results. There is a one-to-one correspondence between bits in the Vector Register Write Mask longword and the vector register numbers. The mask records, starting at bit 0 and proceeding right to left to bit 15, which of the vector registers V0 through V15 were written with one or more elements containing an exceptional result.

The Scalar Register Write Mask parameters record which scalar registers were written with exceptional results. There is a one-to-one correspondence between bits in the Scalar Register Write Mask longwords and the scalar register numbers. Thus the first longword records, starting at bit 0 and proceeding right to left, which of the scalar registers R0 through R31 received an exceptional result. The second longword records the same information, again starting at bit 0 and proceeding right to left, for scalar registers R32 through R63. When the exceptional value is a quadword, the bits corresponding to the register numbers of the low and high parts of the result are both set in the appropriate longword mask.

The actual exceptional value written to the destination register depends on the operation being performed and the type of exception:

- For Integer Overflow the low order 32-bits of the true result are written to the destination register.

- The exceptional result written to the destination register for an Integer Divide by Zero is UNPREDICTABLE.
- The result of a floating comparison or conversion from floating to integer is UNPREDICTABLE if any of the floating operands are reserved.
- All floating exceptional values are encoded as reserved operands with an exception type inserted in the low bits of the word containing the exponent; see Chapter 4, Instruction Descriptions, Page 4-51.

6.4.2 Data Alignment Fault

All data must be naturally aligned or an alignment fault may be generated. Natural alignment means that data bytes are on byte boundaries, data words are on word boundaries, data longwords are on longword boundaries, and data quadwords are on quadword boundaries.

6.4.2.1 Scalar Alignment Fault

A Scalar Alignment fault may be generated when an attempt is made to load or store a word, longword, or quadword to/from a scalar register using an address that does not have the natural alignment of the particular data reference.

Scalar Alignment faults are initiated in the Kernel Mode and push the following information on the Kernel Mode stack:

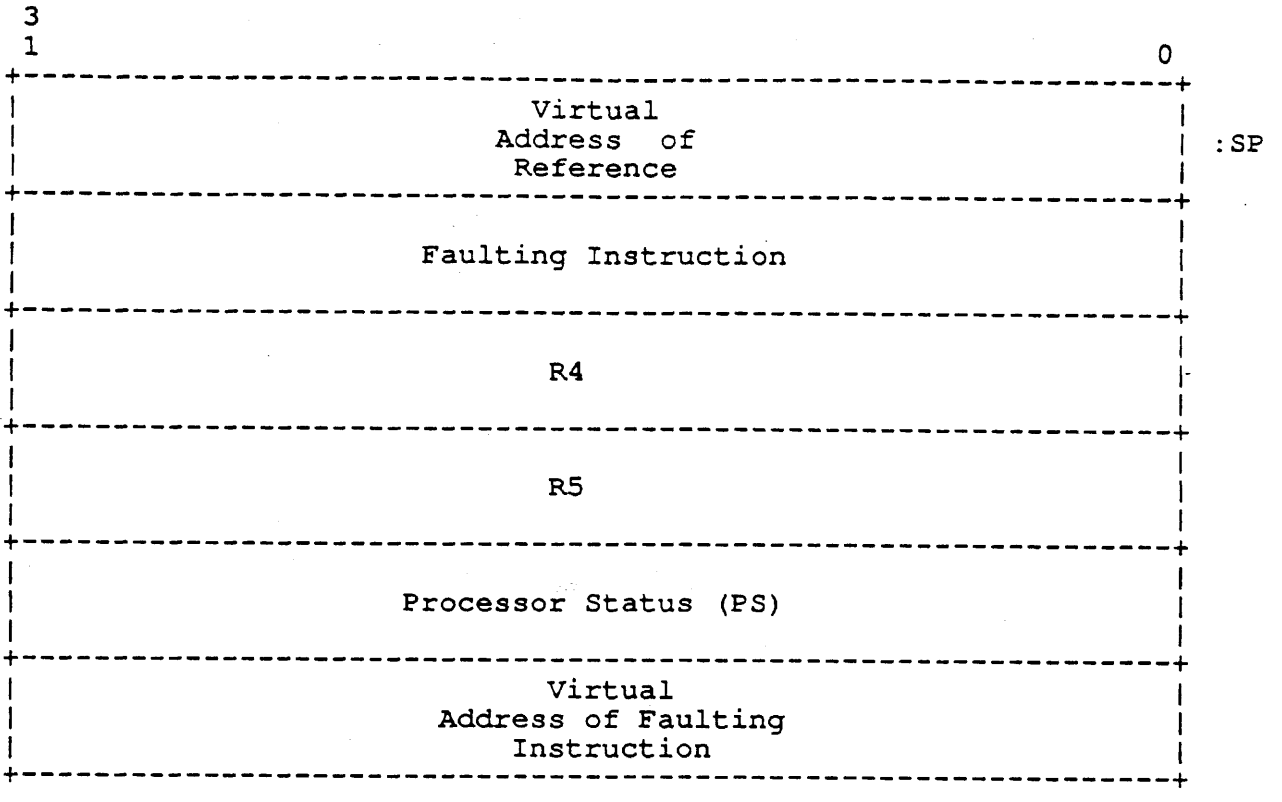


Figure 6-5: Scalar Alignment Fault Exception Frame

The faulting instruction is pushed on the stack so that emulation software can determine the register operands and opcode value. This would not be possible if the instruction was contained in a page that was executable, but not readable.

An implementation may elect to implement scalar data alignment in hardware or Epicode, or force the operating system, or possibly the user (for non-DIGITAL operating system software) to emulate the specified operation by handling this exception.

Emulation software, whether Epicode, an operating system, user code, or hardware may write partial results to memory without probing to make sure all writes will succeed when dealing with unaligned store operations.

If a memory management exception condition occurs while reading or writing part of the unaligned data, the appropriate memory management fault is generated.

Software should avoid data misalignment whenever possible since the emulation performance penalty may be as large as 100 to 1.

6.4.3 Faults Occurring As The Result Of An Instruction

6.4.3.1 Breakpoint Fault

A Breakpoint fault is an exception that occurs when a Breakpoint (BPT) instruction is executed; see Chapter 4, Instruction Descriptions, Page 4-76. Breakpoint faults are intended for use by debuggers and can be used to place breakpoints in a program.

A Breakpoint fault is initiated in Kernel Mode and pushes the following information on the Kernel stack:

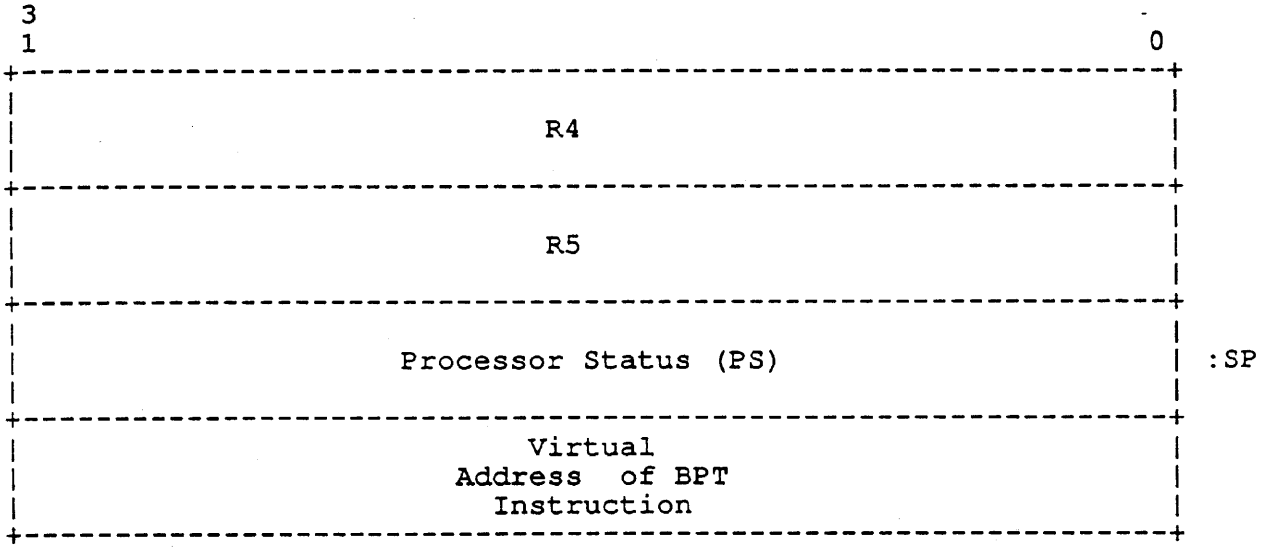


Figure 6-6: Breakpoint Fault Exception Frame

Breakpoint faults are initiated in Kernel Mode so that system debuggers can capture breakpoint faults that occur while the user is executing system code.

6.4.3.2 Fault On Low Bit Clear Fault

A Fault On Low Bit Clear fault is an exception that occurs when a Fault on Low Bit Clear (FLBC) instruction is executed and the low order bit of the specified scalar register is clear; see Chapter 4, Instruction Descriptions, Page 4-73.

Fault On Low Bit Clear faults are initiated in the Kernel Mode and push the following information on the Kernel Mode stack:

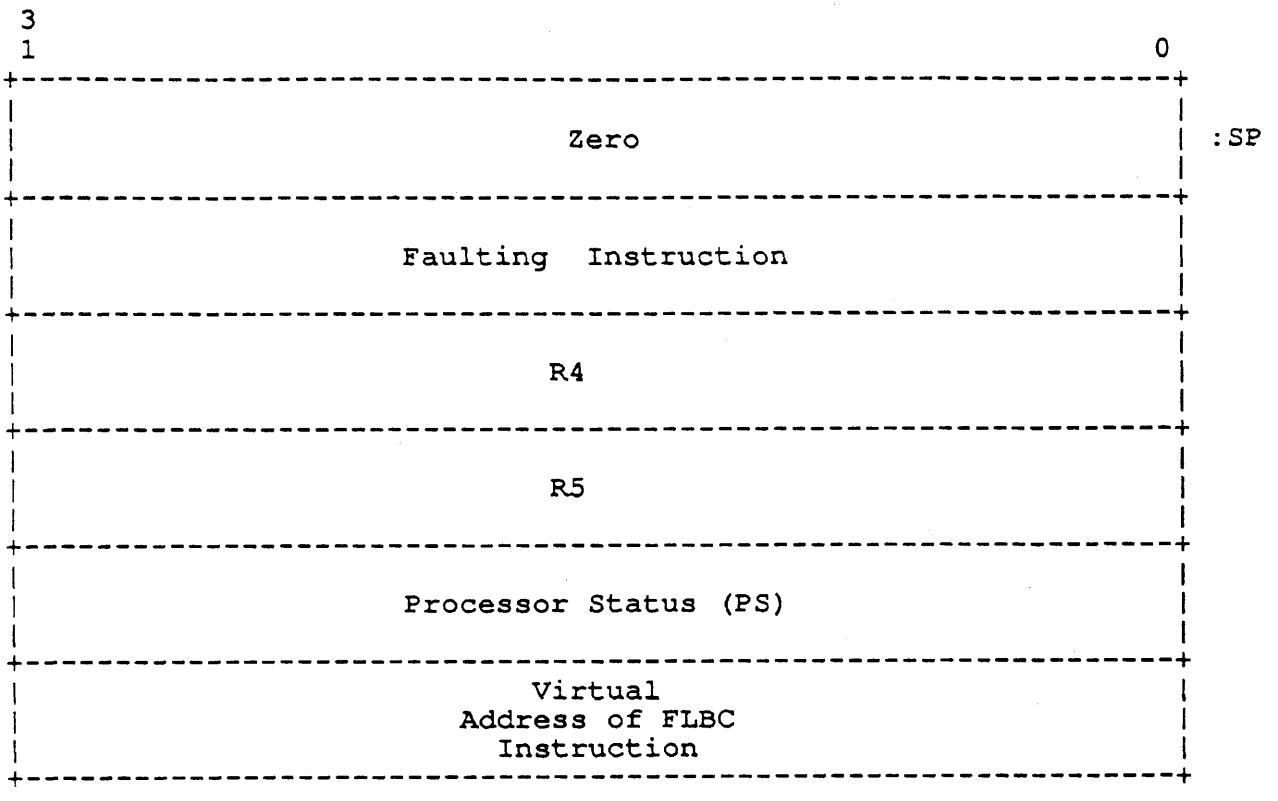


Figure 6-7: Fault On Low Bit Clear Fault Exception Frame

The faulting instruction is pushed on the stack so that software can determine the exact cause of the fault by examining the information encoded in the displacement field. This would not be possible if the instruction was contained in a page that was executable, but not readable.

6.4.4 Illegal Operand Fault

An Illegal Operand fault occurs when an attempt is made to execute an Epicode instruction with operand values that are illegal or reserved for future use by DIGITAL.

Illegal Operand faults are initiated in the Kernel Mode and push the following information on the Kernel Mode stack:

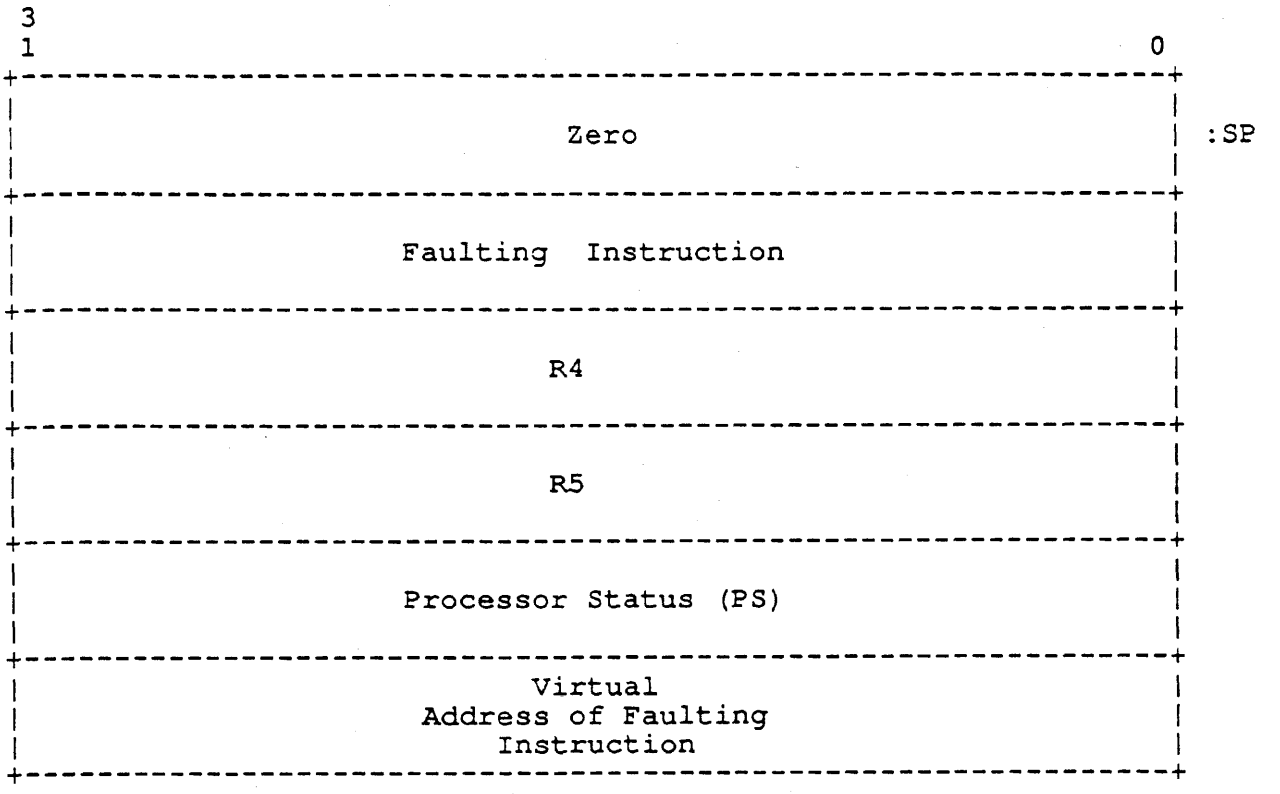


Figure 6-8: Illegal Operand Fault Exception Frame

Illegal operands include:

- An interlock address that is not longword aligned (RMALI, CMPSWLI)
- An interlock address that is not quadword aligned (RMAQI, CMPSWQI)
- An invalid combination of bits in the PS restored by REI
- Unaligned user stack on an REI.

The faulting instruction is pushed on the stack so that software can determine the exact cause of the fault. This would not be possible if the instruction was contained in a page that was executable, but not readable.

All stacks are required to be quadword aligned. It is the responsibility of software to ensure that the initial values for stack pointers are quadword aligned and that subsequent adjustments to the stack pointers are made in quadword increments.

Epicode pops information from the source stack during an REI instruction. Epicode always pops an even number of longwords from the subject stack, thus preserving quadword alignment.

\Quadword alignment is maintained to ensure that a 64-bit architecture can compatibly handle exceptions, interrupts, and the REI instruction. It is also advantageous for Epicode to be able to use quadword instructions to construct exception frames and to read the PS and PC

from the stack on an REI.\

An Illegal Operand fault occurs during the execution of an REI instruction when Epicode attempts to remove the processor state from the User stack and the stack is not quadword aligned.

An unaligned Kernel stack causes a Kernel Stack Not Valid halt; see Section 6.4.6.1.

6.4.4.1 Privileged Instruction

A Privileged Instruction fault is an exception that occurs when an attempt is made to execute a privileged instruction while the current mode is User. Privileged operations can only be executed in Kernel Mode.

Privileged Instruction faults are initiated in the Kernel Mode and push the following information on the Kernel Mode stack:

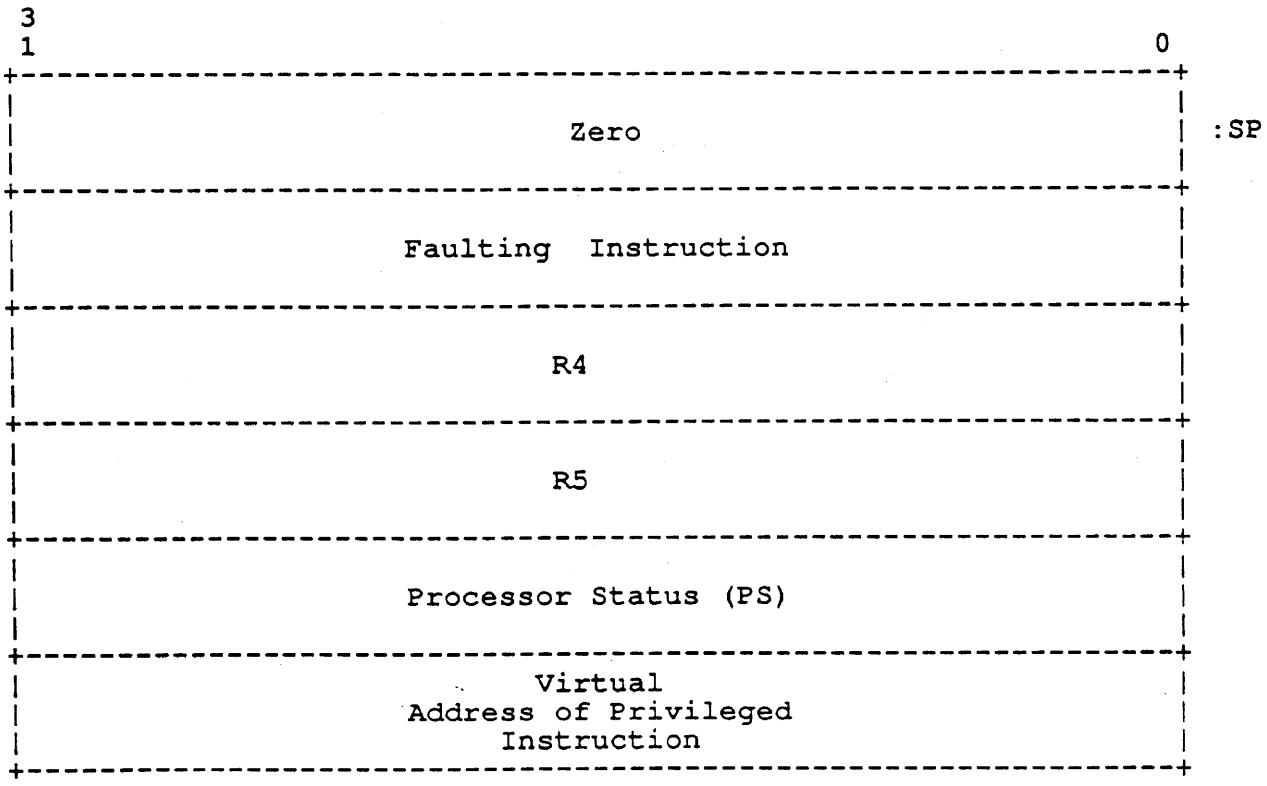


Figure 6-9: Privileged Instruction Fault Exception Frame

The faulting instruction is pushed on the stack so that software can determine the exact cause of the fault. This would not be possible if the instruction was contained in a page that was executable, but not readable.

6.4.4.2 Reserved Opcode Fault

A Reserved Opcode fault is an exception that occurs when an attempt is made to execute an opcode that is reserved to DIGITAL or a subsetted opcode that requires emulation on the host implementation.

Reserved Opcode faults are initiated in the Kernel Mode and push the following information on the Kernel Mode stack:

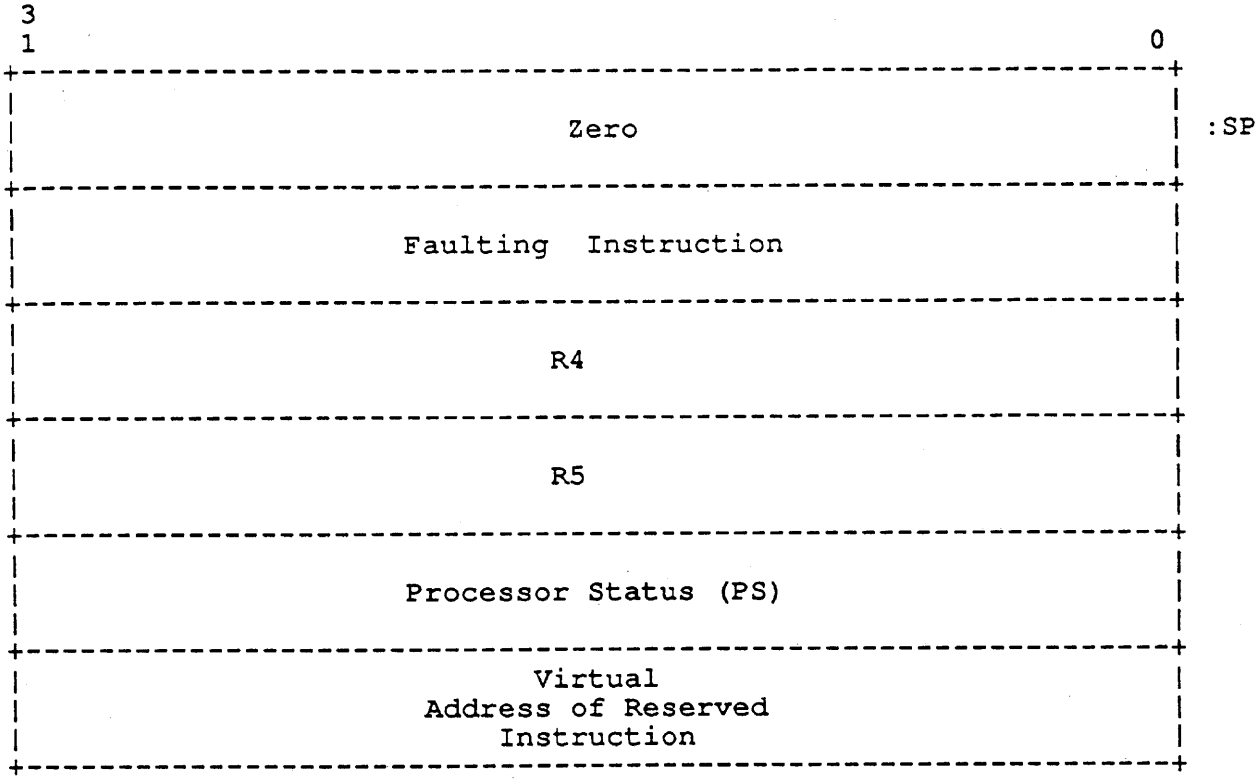


Figure 6-10: Reserved Opcode Fault Exception Frame

The faulting instruction is pushed on the stack so that software can determine the exact cause of the fault. This would not be possible if the instruction was contained in a page that was executable, but not readable.

6.4.4.3 Vector Enable

A Vector Enable fault is generated if an attempt is made to execute a vector instruction when vector instructions are disabled (PS<VEN> is clear), even if a vector processor is not present.

Vector Enable faults are initiated in Kernel Mode and push the following information on the Kernel stack:

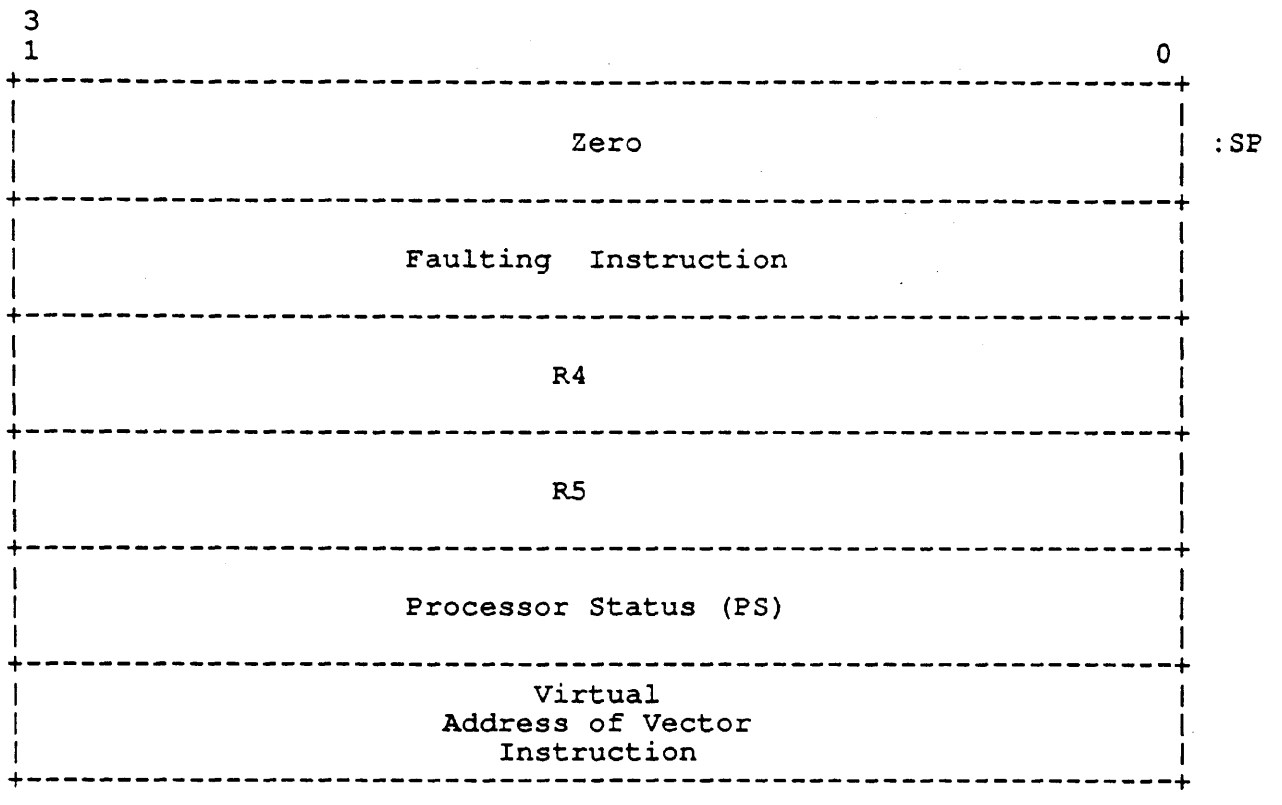


Figure 6-11: Vector Enable Fault Exception Frame

Vector Enable faults can be used to avoid unnecessary saving and restoring of vector registers during context switches without introducing security holes (i.e. passing information from one process to another via the vector registers).

6.4.5 Memory Management Faults

Memory management faults occur when a virtual address translation encounters an exception condition. This can occur as the result of instruction fetch or during a scalar load or store operation.

Memory management faults are generated in Kernel Mode and push the following information on the Kernel stack:

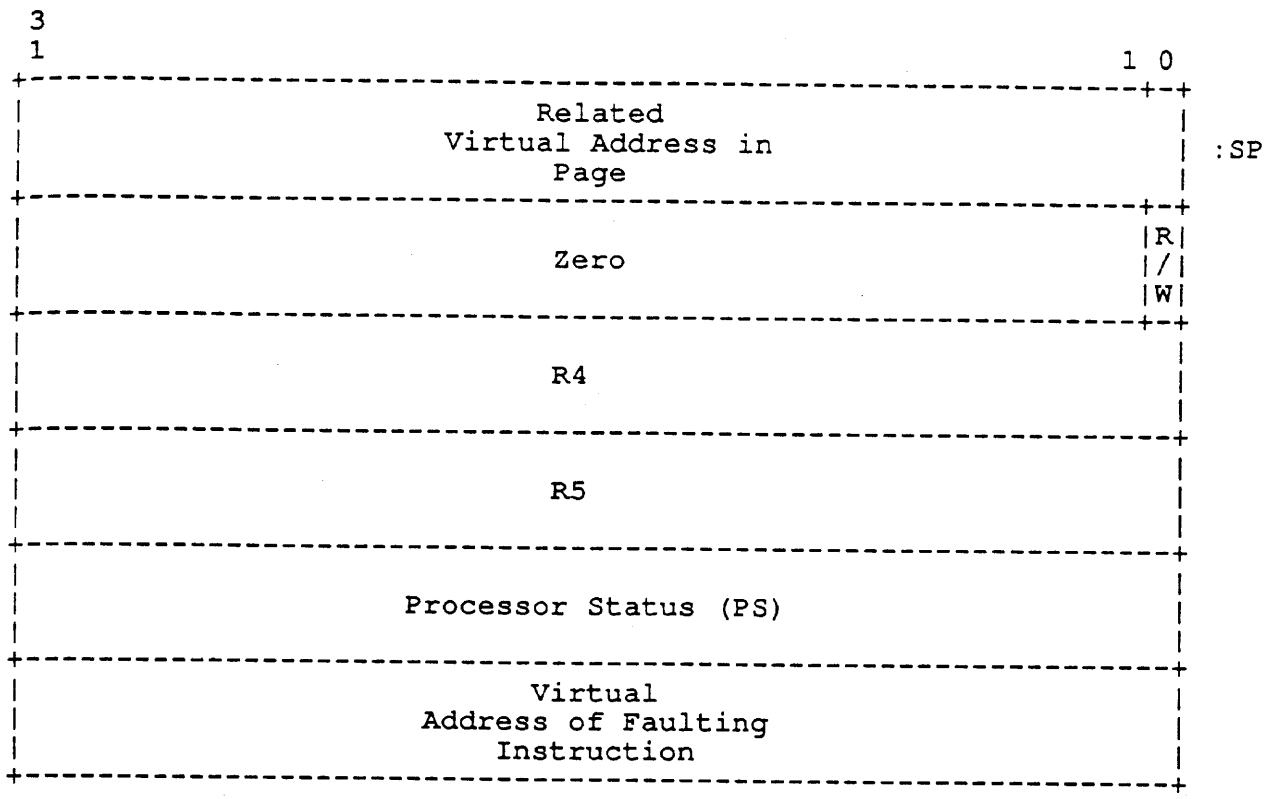


Figure 6-12: Memory Management Fault Exception Frame

The first parameter is a virtual address in the page encountering the fault condition, but not necessarily the exact virtual address.

The second parameter indicates whether the reference was a read (0) or a write (1).

The virtual address of the faulting instruction is the virtual address of the scalar load or store instruction that encountered the fault condition.

Chapter 5, Memory Management, describes the memory management architecture of PRISM in more detail.

6.4.5.1 Access Violation

An Access Violation fault is an exception indicating that an attempted access to a virtual address was not allowed in the current mode.

Access violations usually indicate program errors, but in some cases, such as automatic stack expansion, can mean implicit operating system functions.

Access Violation faults take precedence over Translation Not Valid, Fault On Read, Fault On Write, and Fault On Execute faults.

Access violations take precedence over Translation Not Valid faults for two important reasons:

1. A malicious user could degrade system performance by causing spurious page faults to pages for which no access is allowed.

2. The page fault rate on inaccessible pages could be used as a low bandwidth timing channel to pass critical information and compromise system integrity.

6.4.5.2 Translation Not Valid

A Translation Not Valid fault is an exception that indicates that an attempted access was made to a virtual address whose Page Table Entry (PTE) was not valid.

Software may use Translation Not Valid faults to implement virtual memory capabilities.

6.4.5.3 Fault On Execute

A Fault On Execute fault is an exception that indicates that an attempted instruction stream access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Execute bit set.

Software may use Fault On Execute faults to implement processor mode changes and protected entry to Kernel Mode, and for collecting page usage statistics.

6.4.5.4 Fault On Read

A Fault On Read fault is an exception that indicates that an attempted read access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Read bit set.

Software may use Fault On Read faults to implement watchpoints and for collecting page usage statistics.

6.4.5.5 Fault On Write

A Fault On Write fault is an exception that indicates that an attempted write access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Write bit set.

Software may use Fault On Write faults to maintain modified page information, to implement copy on write and watchpoint capabilities, and for collecting page usage statistics.

6.4.6 Serious System Failures

6.4.6.1 Kernel Stack Not Valid Halt

A Kernel Stack Not Valid halt is an exception that indicates that the Kernel stack was not valid, was unaligned, or a memory error occurred when Epicode attempted to push parameter information during the initiation of an interrupt or exception. Immediately upon detecting this condition the processor enters the restart sequence; see Chapter 11, System Bootstrapping and Console, Section 11.2.2.

6.4.6.2 Machine Check Abort

A Machine Check abort indicates that the processor detected an internal machine error. Common machine check conditions are cache parity errors and internal bus errors.

Machine Check aborts raise IPL to 7 and are initiated in Kernel Mode. The following information is pushed on the Kernel stack:

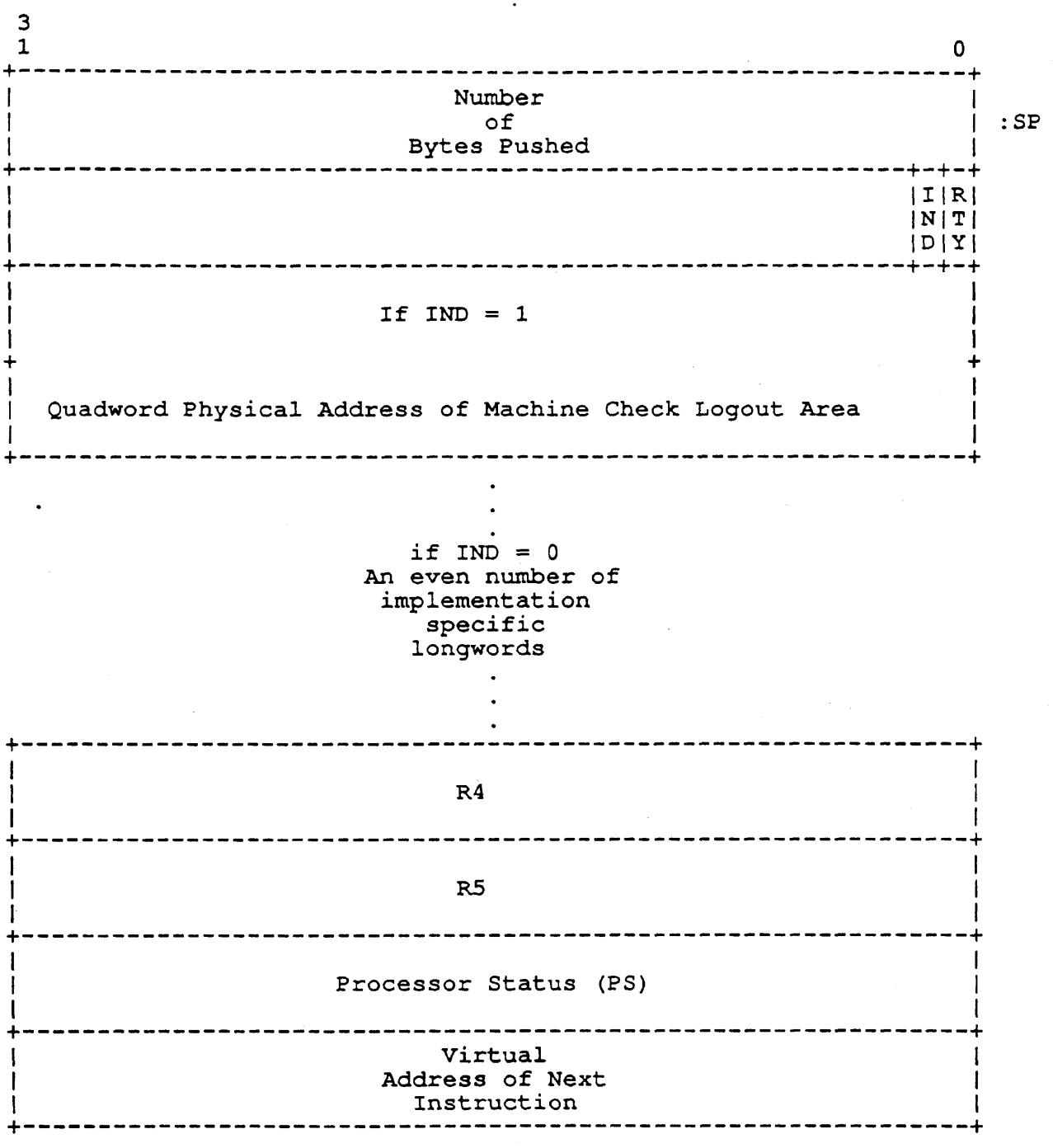


Figure 6-13: Machine Check Abort Exception Frame

Implementation-specific information is either pushed on the stack as longwords or stored in a machine check logout area. An even number of informational longwords are pushed in order to keep the stack quadword aligned. A longword containing a retry (RTY) and indirect (IND) flags followed by the number of parameter bytes are then pushed. The number of parameter bytes does not include the processor state (PS and PC), R4 and R5, but does include the count and flags longwords.

Software must decide, on an implementation-specific basis, depending on the parameters provided, if operations should be aborted. If retry

is possible, Epicode is responsible for taking the appropriate action before setting the RTY flag. If RTY is 1, the error is recoverable and the instruction can be retried.

Epicode sets an internal machine check in progress flag before initiating the machine check exception. This flag is cleared by the operating system machine check handler by writing a 1 to the MCES IPR (see Page 8-14) before it exits. If a second Machine Check is detected while machine check in progress is set, a Double Error abort is generated and the processor enters the restart sequence; see Chapter 11, System Bootstrapping and Console.

6.4.7 Vector Exceptions

Vector instructions perform arithmetic, logical, comparison, and load/store operations on vector registers which consist of more than one element; see Chapter 4, Instruction Descriptions. If an arithmetic exception condition is encountered during a vector operation, it is not reported until the entire vector has been processed. Memory management and alignment exceptions, however, must be reported before the vector operation completes and sufficient state must be saved so the appropriate vector load/store operation can be restarted by software after the exception condition has been corrected.

Several vector and/or scalar operations may be in progress simultaneously, and therefore it is possible to incur more than one memory management, alignment, and/or arithmetic exception condition concurrently.

A vector restart frame is pushed on the Kernel stack for each vector related memory management or alignment exception condition. If an arithmetic exception condition is concurrently present, then exception information for the arithmetic exception is pushed after having pushed the requisite number of vector restart frames.

The following information is pushed on the Kernel stack for each vector restart frame:

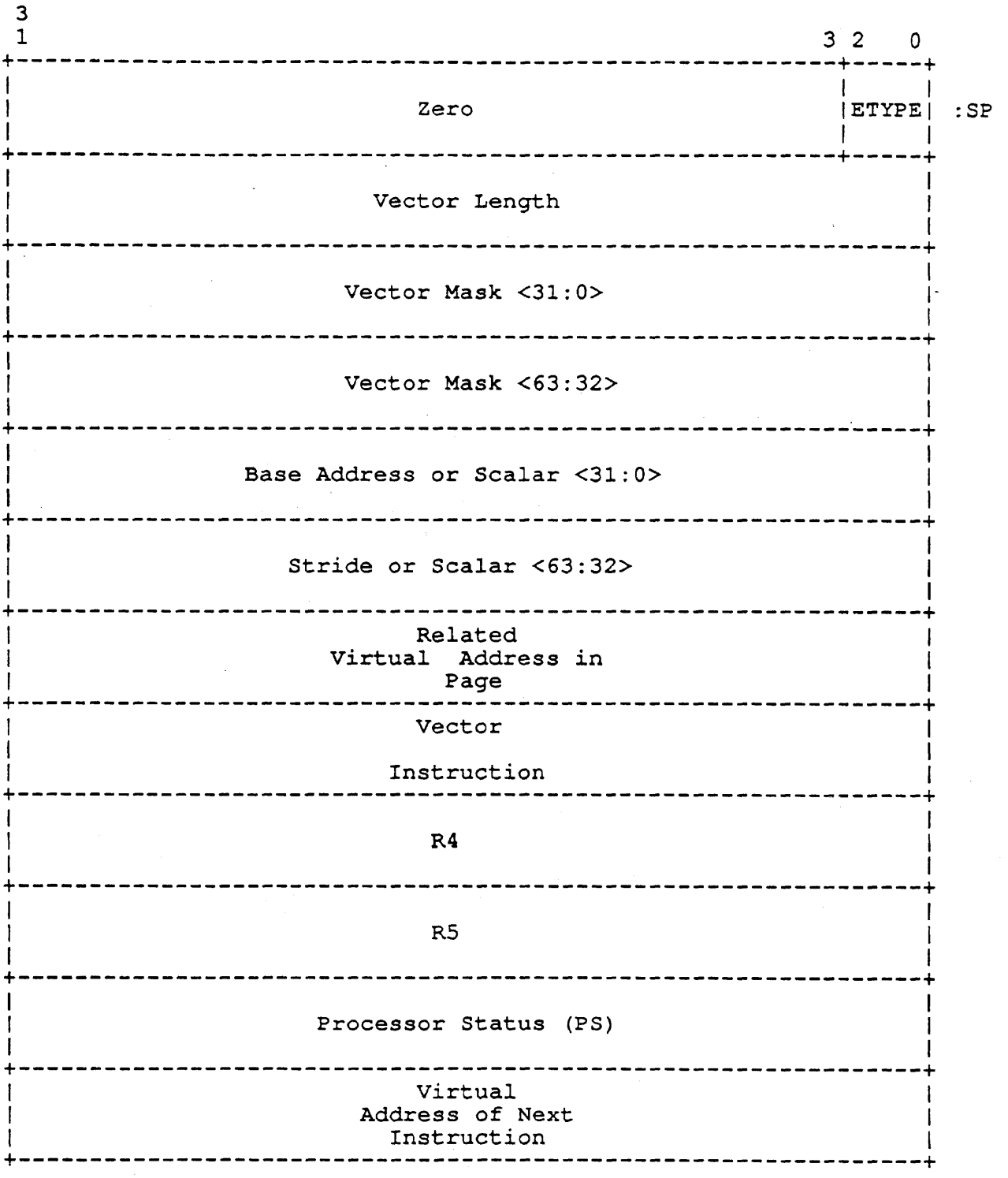


Figure 6-14: Vector Restart Frame

Bits Description

2:0 Exception Type (ETYPE) - The type of exception described by this vector restart frame. Exception types are:

0 - Access Violation
1 - Fault On Read
2 - Fault On Write
3 - Translation Not Valid
4 - Vector Alignment
5 - Instruction Pending

Instruction pending refers to any vector operations that were issued and either not actually started or suspended before completion in a manner that allows them to be restarted from the beginning.

The information pushed on the Kernel stack for an arithmetic exception is described in Section 6.4.1.

If more than one exception frame is pushed on the Kernel stack (i.e. more than one vector restart frame or a vector restart frame and an arithmetic frame), then the current PC followed by the current PS is pushed for each frame. The first exception frame is pushed with the Vector Restart Frame (VRF) bit clear in the saved PS. Subsequent exception frames are pushed with Vector Restart Frame (VRF) set in the saved PS.

If an arithmetic exception frame has been pushed on the Kernel stack, then an arithmetic trap is initiated. If no arithmetic exception frame has been pushed on the Kernel stack, then a vector memory access trap is initiated. Both these exceptions are generated in Kernel Mode.

It is the responsibility of operating system software to process vector memory access traps and complete the corresponding vector load/store instruction as necessary. It is envisioned that operating system software will copy the exception information from the Kernel stack to the previous mode stack (if the previous mode was Kernel, then no copy need take place) and then reflect the exception to a handler in the previous mode. The handler can examine the Exception Type (ETYPE) and take whatever action is appropriate. For example, a Translation Not Valid exception could be handled by first executing a scalar load/store instruction that referenced the related virtual address and then either re-executing the offending instruction (must put all scalar operands back in the original registers) or suitably building or dispatching to a vector instruction that will accomplish the desired load or store operation.

\A vector load or store instruction can require up to 131 pages of memory to be resident in order to complete the instruction. One page is required to hold the instruction, and up to 64 pages may be required to hold the vector data. This in turn can require up to 66 page table pages (one segment 1 page table page, 64 segment 2 page table pages for the vector data, and one segment 2 page table page for the instruction). Therefore, operating system software must guarantee a minimum available working set size of 131 pages for programs that use vector instructions.\

After having processed a vector memory access trap, software should

remove the exception parameters from the stack and execute an REI instruction. If the REI encounters a PS with Vector Restart Frame (VRF) set, then a vector restart fault is initiated: see Section 6.4.7.1. Note that attempting to continue after an arithmetic trap in which the PS has VRF set will also initiate a vector restart fault.

6.4.7.1 Vector Restart Fault

Execution of an REI instruction with Vector Restart Frame (VRF) set in the PS specified by the current stack pointer, causes a vector restart fault to be initiated. The exception is initiated by popping the saved PS and PC from the Current Mode stack, restoring the saved PS and loading the PC with the contents of the vector restart fault vector in the SCB. This is possible since the vector restart frames were originally pushed on the Kernel stack with an identical PC and a PS differing only in whether the Vector Restart Frame (VRF) bit was set or clear.

6.5 SERIALIZATION OF EXCEPTIONS AND INTERRUPTS

It is a goal of the architecture to allow and promote parallel instruction execution. This means that at any point in time there may be several instructions in various stages of execution. When an exception or interrupt condition is detected, all active instructions must be completed before the exception or interrupt can actually be initiated.

In order to accomplish this, instruction issuing is stopped until all instructions in progress have completed. At this point it is possible for multiple exception and interrupt events to be present in which case arithmetic traps take precedence over vector access traps, which take precedence over all other faults, which take precedence over interrupts.

Thus the priority of initiation is:

1. Arithmetic traps
2. Vector access traps
3. All other exceptions (faults)
4. Highest priority interrupt

If an arithmetic trap and a fault condition are both present, any machine state that may have been altered by the fault condition must be sufficiently restored before the arithmetic trap is initiated. Generally, no state may have been altered, but some implementations may need to ensure that subsequent scalar register writes after a memory management fault are backed up or not allowed to occur.

If an exception and an interrupt condition are both present, the exception is initiated. The interrupt will be initiated when conditions permit. This may be on the first instruction of the exception service routine if the exception did not raise IPL (e.g., Machine Check).

In cases where multiple exceptions are possible in a single instruction (e.g., Data Alignment and Translation Not Valid), the order in which the exceptions are detected is UNPREDICTABLE.

6.6 SYSTEM CONTROL BLOCK (SCB)

The System Control Block (SCB) is a quadword aligned region of physically contiguous memory containing vectors by which exceptions and interrupts are dispatched to the appropriate service routines. The address of the SCB is held in an internal processor register and may be loaded by executing a Move To Processor Register instruction specifying the System Control Block Base (MTPR SCBB); See Chapter 8, Internal Processor Registers, Section 8.1.

A vector is a quadword in the SCB that is examined by Epicode when an exception or interrupt is initiated. A unique vector is defined for each interrupt and exception.

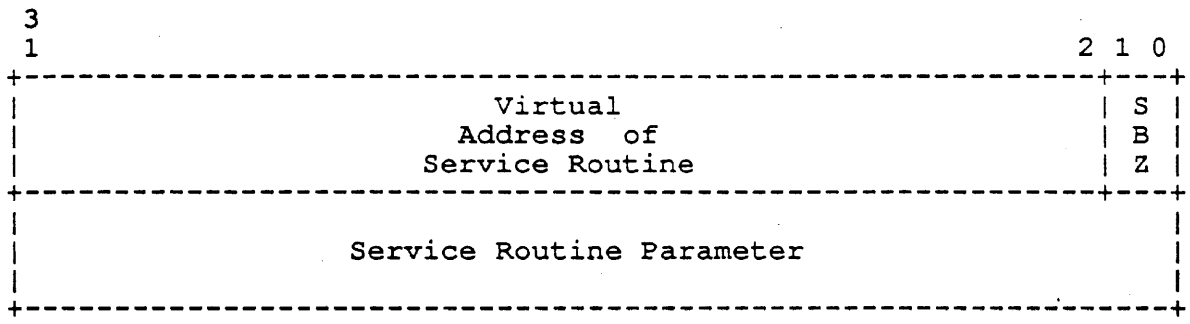


Figure 6-15: System Control Block Vector

If Epicode reads a service routine address for which bits <1:0> are not zero, the resultant operation is UNDEFINED.

Table 6-1: System Control Block Vector Assignments

Vector (hex)	Name	Type	Longwords Stacked	Notes
00	Unused			Reserved to DIGITAL.
08	Machine Check	Abort	*	Implementation specific number of longwords pushed on stack.
10	Fault On Bit	Fault	6	Faulting instruction pushed on stack.
18	Vector Enable	Fault	6	
20	Scalar Alignment	Fault	6	Faulting instruction and virtual address of reference pushed on stack.
28	Access Violation	Fault	6	Virtual address and type of reference pushed on stack.
30	Translation Not Valid	Fault	6	Virtual address and type of reference pushed on stack.
38	Fault On Read	Fault	6	Virtual address and type of reference pushed on stack.
40	Fault On Write	Fault	6	Virtual address and type of reference pushed on stack.
48	Fault On Execute	Fault	6	Virtual address and type of reference pushed on stack.
50	Arithmetic Trap	Trap	8	Exception summary and vector and scalar register write masks pushed on stack.
58	Vector Access	Trap	12	Vector Load/store information is pushed on stack.

Table 6-1: System Control Block Vector Assignments (Continued)

Vector (hex)	Name	Type	Longwords Stacked	Notes
60	Vector Restart	Fault	12	Vector restart frame previously pushed on stack.
68	Software Level 1	Int	4	IPL is raised to 1.
70	Software Level 2	Int	4	IPL is raised to 2.
78	Software Level 3	Int	4	IPL is raised to 3.
80	AST Interrupt	Int	4	IPL is raised to 1.
88	Privileged Instruction	Fault	6	Faulting instruction pushed on stack.
90	Illegal Operand	Fault	6	Faulting instruction pushed on stack.
98	Unused			Reserved to DIGITAL.
100-D8	Bus/Memory Errors	Int	*	Reserved to DIGITAL. Implementation specific number of longwords pushed on stack.
E0	Breakpoint	Fault	4	
E8	Unused			Reserved to DIGITAL.
F0	Reserved Opcode	Fault	6	Faulting instruction pushed on stack.
F8	Power Recovery	Int	4	IPL is raised to 7.
100	Interprocessor Int	Int	4	IPL is raised to 6.
108-178	Unused			Reserved to DIGITAL.
180	Interval Clock	Int	4	IPL is raised to 6.
188-1E8	Unused			Reserved to DIGITAL.
1F0	Console Receive	Int	4	IPL is raised to 4.
1F8	Console Transmit	Int	4	IPL is raised to 4.
200-7F8	Unused			Reserved to DIGITAL.
800-FF8	I/O Devices	Int	4	I/O Device specific interrupt vectors. IPL raised to 4 or 5.

6.7 STACKS

At any point in time, the processor is in one of two modes (Kernel or User). There is a stack pointer associated with each mode. When the processor changes from one of these modes to another, SP (R1) is saved in an Epicode-dependent location for the old state (Epicode may save privileged context in internal registers or in the process privileged context area; see Chapter 7, Process Structure, Section 7.2), and the new SP is loaded from an Epicode-dependent location.

The Current Mode (CM) field of PS specifies which of the two architecturally defined stack pointers is currently in use, as follows:

Mode	Stack
0	Kernel (KSP)
1	User (USP)

6.7.1 Stack Writability

In response to various exceptions and interrupts, Epicode pushes information on the Kernel stack. Epicode may write this information without first probing to ensure that all such writes to the Kernel stack will succeed. If a memory management exception occurs while pushing information, a Kernel Stack Not Valid abort occurs.

6.7.2 Stack Residency

The User stack does not need to be resident. Software running in Kernel Mode can bring in or allocate stack pages as Translation Not Valid faults occur. However, since this activity is taking place in Kernel Mode, the Kernel stack must be resident.

Translation Not Valid, Access Violation, Fault On Read, and Fault On Write faults occurring on Kernel Mode references to the Kernel stack are considered serious system failures from which recovery is not possible. If any of these faults occur, the processor enters the restart sequence; see Chapter 11, System Bootstrapping and Console.

It is not necessary for the Kernel stack to be resident for processes other than the current one, but it must be resident before the process is selected to run by operating system software.

6.7.3 Stack Alignment

All stacks must be quadword aligned. It is the responsibility of software to ensure that stacks are quadword aligned.

Epicode pushes parameters on the Kernel stack in response to exceptions and interrupts. All information pushed is a multiple of quadwords. Thus, if the initial value of a stack pointer is quadword aligned and all adjustments to the respective stack pointer leave it quadword aligned, the stack will remain quadword aligned.

6.7.4 Initiate Exception Or Interrupt

Exceptions and interrupts are initiated by Epicode with interrupts disabled. When an exception or interrupt is initiated, the associated SCB vector is read to determine the address of the service routine. Epicode then attempts to push the PC followed by the PS, R5, and R4 and in the case of exceptions, other parameters if required, on the Kernel stack. During the attempt to push this information, several exceptions can occur. These are:

- Stack Alignment
- Translation Not Valid
- Access Violation
- Fault On Write

If any of the above exceptions occur, a Kernel Stack Not Valid abort is initiated and the processor enters the restart sequence; see Chapter 11, System Bootstrapping and Console.

After the parameters are pushed on the Kernel stack, the contents of the first longword in the SCB vector location are copied into R4 and the PC, the contents of the second longword in the SCB vector location are copied into R5, and instruction execution is initiated in Kernel Mode.

Instruction Issue Model

```
check_for_exception_or_interrupt:

IF NOT {exception or interrupt pending} THEN
    BEGIN
    {fetch next instruction}
    {decode and execute instruction}
    END
ELSE
    BEGIN
    {wait for instructions in progress to complete}
    IPR_SP[PS<CM>] <- SP
    new_sp <- KSP
    IF new_sp<2:0> NE 0 THEN
        {initiate Kernel stack not valid halt}
    IF {exception pending} THEN
        BEGIN
        {back up implementation specific state if necessary}
        IF {vector exception} AND {NOT {machine check}} THEN
            BEGIN
            new_ipl <- PS<IPL>
            tmp <- PS
            FOR i <- 1 TO {number of vector exceptions}
                BEGIN
                PUSH(PC, tmp)
                PUSH(R5, R4)
                PUSH(instruction[i], related_address[i])
                PUSH(stride[i] or scalar<63:32>,
                        initial_base[i] or scalar<31:0>)
                PUSH(vector_mask_hi[i], vector_mask_lo[i])
                PUSH(vector_length[i], exception_type[i])
                tmp<VRF> <- 1
                END
            IF {arithmetic exception} THEN
                BEGIN
                PUSH(PC, tmp)
                PUSH(R5, R4)
                PUSH(write_mask_R63_R32, write_mask_R31_R0)
                PUSH(write_Mask_V15_V0, summary)
                vector <- {arithmetic exception SCB offset}
                END
            ELSE
                vector_offset <-{vector memory access exception SCB offset}
            END
        ELSE
            BEGIN
            IF {machine check} THEN
                new_ipl <- 7
            ELSE
                new_ipl <- PS<IPL>
            PUSH(PC, PS)
            PUSH(R5, R4)
            FOR i <- {number of parameters} / 2 TO 1 BY - 1
                BEGIN
                PUSH(parameter[{i * 2 } + 1], parameter[i * 2])
                END
            IF {{number of parameters} MOD 2} EQ 1 THEN
                PUSH(parameter[1], 0)
            vector_offset <- {exception SCB offset}
```

```
            END
        END
    ELSE
        BEGIN
        new_ipl <- {interrupt source IPL}
        PUSH(PC, PS)
        PUSH(R5, R4)
        vector_offset <- {interrupt SCB offset}
        END
    PS<CM> <- 0
    PS<IPL> <- new_ipl
    SP <- new_sp
    PC <- (SCBB + vector_offset)
    R5 <- (SCBB + vector_offset + 4)
    R4 <- PC
    END
GOTO check_for_exception_or_interrupt


PROCEDURE PUSH(first, last)

BEGIN
IF ACCESS(new_sp - 8, 0) THEN
    BEGIN
    (new_sp - 4) <- first
    (new_sp - 8) <- last
    new_sp <- new_sp - 8
    RETURN
    END
ELSE
    {initiate Kernel stack not valid halt}
END
```

6.7.5 Epicode Interrupt Arbitration

It is envisioned that most, if not all, implementations will provide hardware to check for pending interrupts. This includes software and AST interrupts as well as those caused by the console terminal, Interval Clock, I/O Devices, interprocessor interrupts, and powerfail.

Certain implementations, however, may find it more cost effective to implement parts of the interrupt arbitration in Epicode. The console terminal, Interval Clock, I/O Device interrupts, interprocessor interrupts, and powerfail must be monitored by hardware, and when proper enabling conditions are present, cause an interrupt to be initiated. Software and AST interrupts, however, can be implemented totally in Epicode.

The following sections describe the Epicode instructions that require special checks to implement these capabilities. In all cases, the interrupt is initiated before the execution of the next instruction. In a system that implements interrupts totally in hardware, an identical behavior must be provided.

6.7.5.1 MTPR AST Request Register

Writing the ASTRR internal processor register (see Chapter 8, Internal Processor Registers, Section 8.1) requests an AST for one of the two processor modes. This may request an AST on a formerly inactive level and thus cause an AST interrupt.

The logic required to check for this condition is:

```
ASTSR<mode> <- 1
IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> EQ 0} THEN
    {initiate AST interrupt at IPL 1}
```

6.7.5.2 MTPR Software Interrupt Request Register

Writing the SIRR internal processor register (see Chapter 8, Internal Processor Registers, Section 8.1) requests a software interrupt at one of the four software interrupt levels. This may cause a formerly inactive level to cause a software interrupt.

The logic required to check for this condition is:

```
SISR<level> <- 1
IF level GT PS<IPL> THEN
    {initiate software interrupt at IPL level}
```

6.7.5.3 Return From Exception Or Interrupt

The Return from Exception or Interrupt instruction (see Chapter 4, Instruction Descriptions, Page 4-87) writes both the Current Mode and IPL fields of the PS; see Section 6.2. This may enable a formerly disabled AST or software interrupt to occur.

The logic required to check for this condition is:

```
PS<CM> <- (SP)<CM>
PS<IPL> <- (SP)<IPL>
IF RIGHT_SHIFT(SISR, PS<IPL> + 1) NE 0 THEN
    {initiate software interrupt at IPL of high bit set in SISR}
tmp <- NOT LEFT_SHIFT(10(bin), PS<CM>)
IF {{tmp AND ASTEN AND ASTSR}<1:0> NE 0} AND {PS<IPL> EQ 0} THEN
    {initiate AST interrupt at IPL 1}
```

6.7.5.4 Swap AST Enable

Swapping the AST enable state for the Current Mode results in writing the ASTEN internal processor register (see Chapter 8, Internal Processor Registers, Section 8.1). This may enable a formerly disabled AST to cause an AST interrupt.

The logic required to check for this condition is:

```
tmp <- R4<0>
R4 <- ZEXT(ASTEN<PS<CM>>)
ASTEN<PS<CM>> <- tmp
IF ASTEN<PS<CM>> AND ASTSR<PS<CM>> AND (PS<IPL> EQ 0)
    {initiate AST interrupt at IPL 1}
```

6.7.5.5 Swap Interrupt Priority Level

Swapping the Interrupt Priority Level (IPL) writes the IPL field of the Processor Status (PS); see Section 6.2. This may enable a formerly disabled AST or software interrupt to occur.

The logic required to check for this condition is:

```
tmp <- R4<2:0>
R4 <- ZEXT(PS<IPL>)
PS<IPL> <- tmp
IF RIGHT_SHIFT(SISR, PS<IPL> + 1) NE 0 THEN
    {initiate software interrupt at IPL of high bit set in SISR}
IF ASTEN<0> AND ASTSR<0> AND (PS<IPL> EQ 0) THEN
    {initiate AST interrupt at IPL 1}
```

6.7.6 Processor State Transition Table

Table 6-2: Processor State Transitions

Initial State	Final State: User IPL=0	Kernel IPL=0	Kernel IPL>0	Program Halt
USER IPL=0		Exc	Int Exc SWASTEN	NP
KERNEL IPL=0	REI*		REI SWIPL Int Exc MTPR* SWASTEN	HALT
KERNEL IPL>0	REI*	REI* SWIPL*		HALT

* - An REI that increases mode or lowers IPL, or a SWIPL that lowers IPL, or a MTPR ASTRR or MTPR ASTEN, can cause an interrupt request at IPL 1.

Exc - State change caused by an exception.

Int - State change caused by an interrupt.

NP - State not possible.

Revision History:

Revision 3.0, 26 April 1988

1. Push faulting instruction on stack for vector enable fault.
2. Allow vector restart frame to include any unexecuted vector instruction.
3. Remove Stack Alignment abort.
4. Remove Bug Check fault.
5. Add retry bit and indirect bit to machine check frame.
6. Clarify forwarding of exceptional results.
7. All exceptions go to Kernel mode.
8. Revise SCB vector assignments to make console and IP interrupt vector offsets match VAX SCB offsets.
9. Make SCB vectors quadwords containing service routine address and parameter.
10. Push R5 and R4 on kernel stack on exceptions and interrupts. Copy SCB vector into R4. Copy service routine parameter into R5.

Revision 2.0, 24 Jun 1986

1. Minor edits and clarifications for revision 2.0.
2. Change references to I/O Port Controllers to I/O Devices.
3. Change PS to contain one bit for mode and delete all references to Supervisor and Executive modes.
4. Rename the Vector Exception Frame (VEF) bit in the PS to the Vector Restart Frame (VRF) bit.
5. Change the interval clock interval from 10ms to 1ms and change the interrupt priority from 5 to 6.
6. Clarify floating reserved operand exceptions and the use of forwarded exceptional values.
7. Delete vector alignment abort exceptions, they are now covered by new exceptions called Vector Restart and Vector Memory Access.
8. Change Fault On Bit fault to Fault On Low Bit Clear Fault.
9. Change the privileged instruction fault frame to include the privileged instruction.
10. Change memory management exceptions to apply only to scalar load and store instructions. Vector memory management and alignment exceptions are covered by Vector Memory Access and

Restart exceptions.

11. Delete all references to the Supervisor and Executive mode stacks.

12. Delete the vector alignment SCB vector and replace with the vector restart exception vector.

13. Change instruction issue model to conform to the new Vector Restart and Vector Memory Access exceptions.

14. Change state transition table to include only Kernel and User Modes.

Revision 1.0, 22 December 1985

1. General rewrite of chapter to better organize information and to reflect the change from a 64- to a 32-bit architecture.

2. Change the number of IPLs from 32 to 8.

3. Removal of all types of traps except arithmetic traps. There is now only one kind of trap.

4. Renamed PSQ to PS and PC.

5. Previous mode, interrupt stack, and interrupt disable were removed from the PS to simplify the privileged architecture.

6. Added vector fault to the definition of PS for saved copies of PS. This bit is similar in functionality to First Part Done (FPD) on VAX.

7. Added vector enable to the definition of PS. This bit enables the use of vector instructions and enables optimization of the saving and restoring of vector registers for processes that do not use them without introducing security holes.

8. Added Vector Enable fault.

9. Changed PS to a longword and PC to a longword.

10. Added I/O Port Controller interrupts as part of adding the I/O architecture.

11. Removed much information that was duplicated in other places and inserted a reference to the proper definition.

12. Revised arithmetic traps to reflect the agreed upon handling at the August 23 technical review.

13. Added Fault On Bit fault and dropped User Check trap.

14. Added Fault On Read, Fault On Write, and Fault On Execute faults as part of the simplification of memory management.

15. Dropped the separate fault for emulated instructions and combined with reserved opcode.

16. Change Bug Check trap to fault.

17. Added vector exception information and an explanation of how vector arithmetic and memory management faults are handled.

18. Grossly simplified serialization rules.

19. Added section on instruction issue and how it pertains to exceptions and interrupts.

20. Added section on Epicode interrupt arbitration for instructions that alter the state such that an AST or software interrupt may be generated.

21. Updated state transition table to reflect simplified privileged architecture.

Revision 0.0, July 5, 1985

1. First review distribution.

CHAPTER 7

PROCESS STRUCTURE

7.1 PROCESS DEFINITION

A process is the basic entity that is scheduled for execution by the processor. A process represents a single thread of execution and consists of an address space and both hardware and software context.

The hardware context of a process is defined by:

- 64 scalar registers
- 16 vector registers
- Vector Length register (VL)
- Vector Count register (VC)
- Vector Mask register (VM)
- Processor Status (PS)
- Program Counter (PC)
- 2 stack pointers
- Asynchronous System Trap Enable register (ASTEN)
- Asynchronous System Trap Summary Register (ASTSR)
- Process Page Table Base Register (PTBR)
- Address Space Number (ASN)
- Cycle Count Register (CCR)

The software context of a process is defined by operating system software and is system dependent.

A process may share the same address space with other processes or have an address space of its own. There is, however, no separate address space for system software, and therefore, the operating system must be mapped into the address space of each process at identical virtual addresses; see Chapter 5, Memory Management.

In order for a process to execute, its hardware context must be loaded into the scalar registers, vector registers, and internal processor

registers. While a process is executing, its hardware context is continuously updated. When a process is not being executed, its hardware context is stored in memory.

Saving the hardware context of the current process in memory, followed by loading the hardware context for a new process, is termed context switching. Context switching occurs as one process after another is scheduled by the operating system for execution.

7.2 HARDWARE PRIVILEGED PROCESS CONTEXT

The hardware context of a process is defined by a privileged part which is context switched with the Swap Privileged Context instruction (SWPCTX) (see Chapter 4, Instruction Descriptions, Page 4-98) and a nonprivileged part which is context switched by operating system software.

When a process is not executing, its privileged context is stored in a quadword aligned memory structure called the Hardware Privileged Context Block (HWPCB).

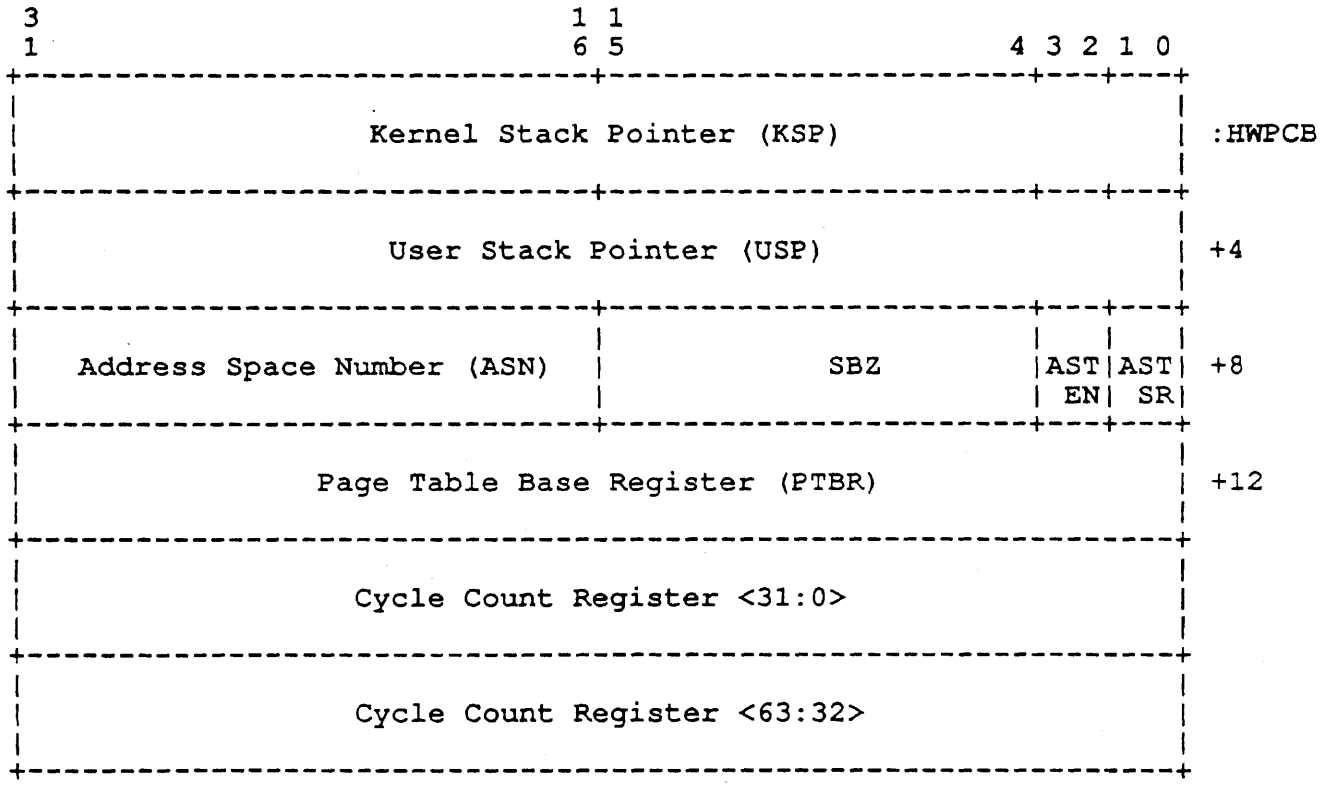


Figure 7-1: Hardware Privileged Context Block

The Hardware Privileged Context Block (HWPCB) for the current process is specified by the Privileged Context Block Base register (PCBB); see Chapter 8, Internal Processor Registers, Page 8-15.

If ASNs are not implemented, the ASN field of the HWPCB Should Be Zero (SBZ).

The Swap Privileged Context instruction (SWPCTX) saves the privileged context of the current process into the HWPCB specified by PCBB, loads a new value into PCBB, and then loads the privileged context of the new process into the appropriate hardware registers.

The new value loaded into PCBB, as well as the contents of the Privileged Context Block, must satisfy certain constraints or an UNDEFINED operation results:

1. The physical address loaded into PCBB must be quadword aligned, and describe six contiguous longwords that are neither in I/O space nor in non-existent memory.

2. The value of PTBR must be the Page Frame Number of an existent page that is neither in I/O space nor in non-existent memory.

It is the responsibility of the operating system to save and load the nonprivileged part of the hardware context.

The SWPCTX instruction returns ownership of the current HWPCB to operating system software and passes ownership of the new HWPCB from the operating system to the processor. Any attempt to read or write a HWPCB while ownership resides with the processor has UNPREDICTABLE results.

7.3 ASYNCHRONOUS SYSTEM TRAPS (AST)

Asynchronous System Traps (ASTs) are a means of notifying a process of events that are not synchronized with its execution, but which must be dealt with in the context of the process with minimum delay.

Asynchronous System Traps (ASTs) interrupt process execution and are controlled by the AST Enable (ASTEN) and AST Summary (ASTSR) internal processor registers; see Chapter 8, Internal Processor Registers, Pages 8-4 and 8-6.

The AST Enable register (ASTEN) contains an enable bit for each of the two processor modes. When the bit corresponding to a processor mode is set, ASTs for that mode are enabled. The AST enable bit for a processor mode may be changed by executing a Swap AST Enable instruction (SWASTEN); see Chapter 4, Instruction Descriptions, Page 4-89.

The AST Summary Register (ASTSR) contains a pending bit for each of the two processor modes. When the bit corresponding to a processor mode is set, an AST is pending for that mode. The AST pending bit for a processor mode may be set by requesting an AST for the respective mode.

Kernel mode software may request an AST for a particular processor mode by executing a Move To Processor Register instruction specifying ASTRR (MTPR ASTRR); see Chapter 8, Internal Processor Registers, Page 8-5.

Hardware or Epicode monitors the state of ASTEN, ASTSR, PS<CM>, and PS<IPL>. If PS<IPL> is zero, and there is an AST pending and enabled

for a processor mode that is less than or equal to PS<CM> (i.e. an equal or more privileged processor mode), an AST interrupt is initiated at IPL 1. ASTs that are pending and enabled for a less privileged processor mode are not allowed to interrupt execution in a more privileged processor mode.

7.3.1 A Software Model For AST Processing

It is intended that ASTs represent a single level of interrupt for each of the two processor modes. Therefore, operating system software should not allow nested ASTs to occur within a single mode. One way to accomplish this is for operating system software to keep track of the processor modes for which an AST is currently in progress and not request further ASTs for these processor modes until processing of the respective ASTs has been completed.

In the following discussion it is assumed that the operating system maintains a per process mask that contains one bit for each of the processor modes for which an AST is currently active. When an AST is delivered to a particular processor mode, the corresponding bit in the active mask is set. Later, when AST processing is completed, the operating system clears the respective bit and checks if any ASTs have been queued at the particular level but not requested.

The operating system must also keep track of the processor mode which is to receive an AST when the event associated with the AST is completed. Typically, such an event is the completion of an asynchronous I/O request or the expiration of a timer. The simplest way to do this is to construct an AST control block when the original request is received and record in the control block the processor mode and address of the AST routine that is to be executed.

A simple model for uniprocessor AST delivery:

1. The completion of an event for which an AST has been requested causes operating system software to place an AST control block in a queue associated with the target process. The AST queue is ordered by processor mode with more privileged entries at the front of the queue.

2. If the target process is currently executing and an AST is not currently in progress for the specified processor mode, an AST is requested for the corresponding processor mode by executing a MTPR ASTRR instruction. If the target process is not currently executing and an AST is not currently in progress for the specified processor mode, an AST is requested by setting the bit corresponding to the specified processor mode in the saved ASTSR of the target process.

3. Hardware or Epicode monitors the state of ASTEN, ASTSR, PS<CM>, and PS<IPL>. If PS<IPL> is zero and there is an AST pending and enabled for an processor mode that is less than or equal to PS<CM> (i.e. an equal or more privileged processor mode), an AST interrupt is initiated at IPL 1.

4. The AST delivery interrupt service routine is entered at IPL 1 in Kernel mode and attempts to remove an AST control block from the process AST queue. The AST queue must be scanned

from the front, looking for an entry that specifies a processor mode that is less than or equal to the current mode of the saved PS (a processor mode that is equal to or more privileged than the previous processor mode) and for which ASTs are enabled and not active (i.e. there is not already an AST in progress for the mode). If an appropriate entry is located, then it is removed from the queue and the bit corresponding to the destination processor mode is set in the active mask. An appropriate PS and PC are constructed on the Kernel stack and an REI is executed which begins execution of the AST routine. If an appropriate AST control block cannot be located, the AST interrupt is simply dismissed. (It is possible for this condition to arise in the special case where an AST interrupt is initiated, clearing the corresponding pending bit in ASTSR, and before operating system software sets the appropriate bit in the active mask, another AST for the same processor mode is requested.)

5. At the conclusion of processing an AST, the AST routine calls the operating system to exit from the AST. The operating system clears the appropriate bit in the active mask and checks to see if another AST has been queued for the specified processor mode. If another AST has been queued, an AST is requested by executing an MTPR ASTRR specifying the appropriate processor mode.

7.4 PROCESS CONTEXT SWITCHING

Process context switching occurs as one process after another is scheduled for execution by operating system software. Context switching requires the hardware context of one process to be saved in memory followed by the loading of the hardware context for another process into the hardware registers.

The privileged hardware context is swapped with the Swap Privileged Context instruction (SWPCTX). Other hardware context must be saved and restored by operating system software.

The sequence in which process context is changed is important since the SWPCTX instruction changes the environment in which the context switching software itself is executing. Also, although not enforced by hardware, it is advisable to execute the actual context switching software in an environment which cannot be context switched (i.e. at an IPL high enough that rescheduling cannot occur).

The SWPCTX instruction is the only method provided for loading certain internal processor registers. The SWPCTX instruction always saves the privileged context of the old process and loads the privileged context of a new process. Therefore, a valid HWPCB must be available to save the privileged context of the old process as well as load the privileged context of the new process.

At system initialization, a valid HWPCB is constructed in the Restart Parameter Block (RPB) for each processor; see Chapter 11, System Bootstrapping and Console, Section 11.1.1.2. Thereafter, it is the responsibility of operating system software to ensure a valid HWPCB when executing a SWPCTX instruction.

7.4.1 A Software Model For Process Context Switching

The following context switching code represents a model by which operating system software can switch context from one process to another.

Certain assumptions are made regarding the entry and exit conditions of this code. At entry it is assumed that the code is executing in Kernel mode at IPL 2 and that the continuation PC and PS have already been saved on the Kernel stack. At exit, the execution of the new process is to be continued by an REI instruction.

```
swap_process_context:
    sub     #4*4,sp,sp              ; allocate room to save registers
    stq     r4,8(sp)                ; save scalar registers R4 and R5
    stq     r2,(sp)                 ; save scalar registers R2 and R3
    mfpr    prbr                    ; read processor base register into R4
    ldl     prb$l_swpcb(r4),r2      ; get address of current software PCB
    stq     r6,swpcb$l_r6(r2)       ; save scalar registers R6 and R7
    stq     r8,swpcb$l_r8(r2)       ; save scalar registers R8 and R9
    stq     r10,swpcb$l_r10(r2)     ; save scalar registers R10 and R11
        .
        .
        .
    stq     r58,swpcb$l_r58(r2)     ; save scalar registers R58 and R59
    stq     r60,swpcb$l_r60(r2)     ; save scalar registers R60 and R61
    stq     r62,swpcb$l_r62(r2)     ; save scalar registers R62 and R63
    ldl     16(sp),r4               ; get saved PS
    srl     #ps$v_ven,r4,r3         ; shift PS<VEN> to low bit
    blbc    r3,10$                  ; if low bit clear, not using vectors
    rdvc    r4                      ; read vector count register
    rdvl    r5                      ; read vector length register
    stq     r4,swpcb$l_vc(r2)       ; save vector count and length registers
    rdvml   r4                      ; read low half of vector mask register
    rdvmh   r5                      ; read high half of vector mask register
    stq     r4,swpcb$l_vml(r2)      ; save vector mask register
    add     #64,r0,r4               ; set vector length to 64 elements
    wrvl    r4                      ;
    lda     swpcb$q_v0(r2),r2       ; get base address of vector save area
    vstq    #8,r2,v0                ; save vector register V0
    lda     64*8(r2),r2             ; get address of next vector save area
    vstq    #8,r2,v1                ; save vector register V1
    lda     64*8(r2),r2             ; get address of next vector save area
    vstq    #8,r2,v2                ; save vector register V2
        .
        .
        .
    lda     64*8(r2),r2             ; get address of next vector save area
    vstq    #8,r2,v13               ; save vector register V13
    lda     64*8(r2),r2             ; get address of next vector save area
    vstq    #8,r2,v14               ; save vector register V14
    lda     64*8(r2),r2             ; get address of next vector save area
    vstq    #8,r2,v15               ; save vector register V15
10$:                                ;
;
; Execute operating system dependent code to select new process.
;
; Exit with:
;
```

```
;       R2 - address of new process software PCB.
;

    mfpr    prbr                    ; read processor base register
    stl     r2,prb$l_swpcb(r4)      ; set address of new software PCB
    ldq     swpcb$q_hwpcb(r2),r4 ; get physical address of hardware PCB
    swpctx                          ; swap privileged context

;
; The privileged context has been swapped at this point and thus
; a new address space is in effect as is a new Kernel stack pointer
; and saved PC and PS.
;

    ldl     16(sp),r4               ; get saved PS
    srl     #ps$v_ven,r4,r3         ; shift PS<VEN> to low bit
    blbc    r3,20$                  ; if low bit clear, not using vectors
    or      #1,r0,r4
    mtpr    ven                     ; enable vector instructions
    add     #64,r0,r4               ; set vector length to 64 elements
    wrvl    r4                      ;
    lda     swpcb$q_v0(r2),r3       ; get base address of vector save area
    vldq    #8,r3,v0                ; load vector register V0
    lda     64*8(r3),r3             ; get address of next vector save area
    vldq    #8,r3,v1                ; load vector register V1
    lda     64*8(r3),r3             ; get address of next vector save area
    vldq    #8,r3,v2                ; load vector register V2
        .
        .
        .
    lda     64*8(r3),r3             ; get address of next vector save area
    vldq    #8,r3,v13               ; load vector register V13
    lda     64*8(r3),r3             ; get address of next vector save area
    vldq    #8,r3,v14               ; load vector register V14
    lda     64*8(r3),r3             ; get address of next vector save area
    vldq    #8,r3,v15               ; load vector register V15
    ldq     swpcb$l_vc(r2),r4       ; get saved vector count and length
    wrvc    r4                      ; write vector count register
    wrvl    r5                      ; write vector length register
    ldq     swpcb$l_vml(r2),r4      ; get saved vector mask
    wrvml   r4                      ; write low half of vector mask register
    wrvmh   r5                      ; write high half of vector mask register
20$:
    ldq     swpcb$l_r6(r2),r6       ; load scalar registers R6 and R7
    ldq     swpcb$l_r8(r2),r8       ; load scalar registers R8 and R9
    ldq     swpcb$l_r10(r2),r10     ; load scalar registers R10 and R11
        .
        .
        .
    ldq     swpcb$l_r58(r2),r58 ; load scalar registers R58 and R59
    ldq     swpcb$l_r60(r2),r60 ; load scalar registers R60 and R61
    ldq     swpcb$l_r62(r2),r62 ; load scalar registers R62 and R63
    ldq     (sp),r2             ; load scalar registers R2 and R3
    ldq     8(sp),r4            ; load scalar registers R4 and R5
    add     #4*4,sp,sp          ; deallocate register save area
    rei                         ; resume process execution
```

Revision History:

Revision 3.0, 26 April 1988

1. Add Cycle Count Register to HWPCB.

Revision 2.0, 24 Jun 1986

1. Change format of HWPCB to contain only the Kernel and User stack pointers. Reduce the size of the ASTEN and ASTSR fields in the HWPCB from four to two bits.

2. Remove all references to four access modes and replace with references to two access modes.

3. Change context switch code to explicitly set the vector length register to 64 elements.

Revision 1.0, 22 December 1985

1. Chapter rewritten to reflect simplified privileged architecture.

2. Removed all explicit assumptions about how operating system software uses the hardware process structure.

3. Removed references to PSW, ASTLVL, and the interrupt stack.

4. Added new definition of hardware context and defined the Hardware Privileged Context Block (HWPCB).

5. Revised the AST section and added a software model of AST processing.

6. Deleted the section on Process Structure Interrupts.

7. Combined the sections on saving and loading process context into a single section on swapping context.

Revision 0.0, July 5, 1985

1. First review distribution.

CHAPTER 8

INTERNAL PROCESSOR REGISTERS

8.1 INTERNAL PROCESSOR REGISTERS

This chapter describes the PRISM Internal Processor Registers (IPRs). These registers are read and written with Move From Processor Register (MFPR) and Move To Processor Register (MTPR) instructions; see Chapter 4, Instruction Descriptions, Pages 4-95 and 4-96.

These instructions accept input operands from, and write results to, the scalar registers R4, R5, and R6. Prior to execution of an MTPR/MFPR, required input operands must be loaded into scalar registers R4 and R5. In certain cases no input operands are required. MFPR returns the IPR contents in one or more of the scalar registers R4, R5, and R6.

Internal Processor Registers may or may not be implemented as actual hardware registers. An implementation may choose any combination of Epicode and hardware to produce the architecturally specified functionality.

Internal Processor Registers are only accessible from Kernel mode.

Table 8-1: Internal Processor Register (IPR) Summary

Register Name	Mnemonic	Access	R4	R5	R6
Address Space Number	ASN	R	number		
AST Enable	ASTEN	R/W	mask		
AST Request Register	ASTRR	W	mode		
AST Summary Register	ASTSR	R	mask		
Console Receive Ctrl. Status	CRCS	R/W	enable		
Console Receive Data Buffer	CRDB	R	char		
Console Transmit Ctrl. Status	CTCS	R/W	enable		
Console Transmit Data Buffer	CTDB	W	char		
Interval Clock Int. Enable	ICIE	R/W	enable		
Interprocessor Int. Enable	IPIE	R/W	enable		
Interprocessor Int. Request	IPIR	W	number		
Machine Check Error Summary	MCES	R/W	value		
Privileged Context Block Base	PCBB	R	address	address	
Processor Base Register	PRBR	R/W	value		
Page Table Base Register	PTBR	R	frame		
System Control Block Base	SCBB	R/W	address	address	
System Identification	SID	R	ident	value	
Software Int. Request Register	SIRR	W	level		
Software Int. Summary Register	SISR	R	mask		
System Serial Number	SSN	R	serial		
Trans. Buffer Check	TBCHK	R	number	address	status
Trans. Buffer Invalidate Single	TBIS	W	number	address	
Time Of Year	TOY	R/W	time		
User Stack Pointer	USP	R/W	address		
Vector Enable	VEN	W	value		
Who-Am-I	WHAMI	R	number		

NOTE

\A common console architecture for uniprocessor and multiprocessor systems is currently being defined. The Console registers are to subject to change.\

Address Space Number (ASN)

Access:

Read

Operation:

R4 <- ZEXT(ASN<15:0>)

Value at System Initialization:

Zero

Format:

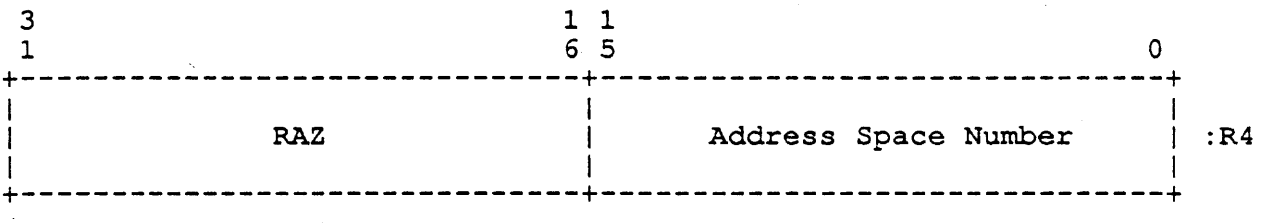


Figure 8-1: Address Space Number Register (ASN)

Description:

Address Space Numbers (ASNs) are used to further qualify Translation Buffer references; see Chapter 5, Memory Management. The current ASN may be read by executing an MFPR instruction specifying ASN.

As processes are scheduled for execution, the ASN for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapter 4, Instruction Descriptions, Page 4-98 and Chapter 7, Process Structure.

AST Enable (ASTEN)

Access:

Read/Write

Operation:

```
R4 <- ZEXT(ASTEN<1:0>)          !Read
ASTEN<1:0> <- R4<1:0>           !Write
```

Value at System Initialization:

Zero

Format:

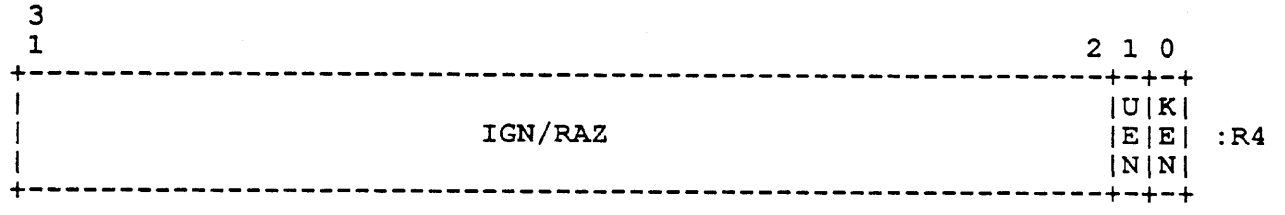


Figure 8-2: AST Enable Register (ASTEN)

Description:

The AST Enable register records the AST enable state for each processor mode: Kernel (KEN) and User (UEN). The current AST enable state may be read and written by executing MFPR and MTPR instructions that specify ASTEN.

\ASTEN is not present in the VAX architecture. It was added to the PRISM architecture to allow software (especially nonprivileged software) to enable and disable ASTs efficiently for the current mode via the SWASTEN instruction. It is anticipated that, with multitasking, it will become extremely important to be able to enable and disable ASTs in an efficient manner in shareable runtime support routines.\

As processes are scheduled for execution, the state of AST Enable for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction. The Swap AST Enable (SWASTEN) instruction can be used to change the enable state for the current processor mode; See Chapter 4, Instruction Descriptions, Pages 4-98 and 4-89, and Chapter 7, Process Structure.

AST Request Register (ASTRR)

Access:

Write

Operation:

ASTRR <- R4<0>

Value at System Initialization:

Not Applicable

Format:

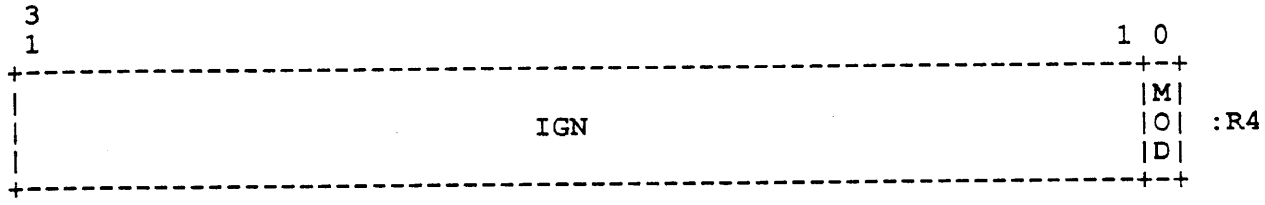


Figure 8-3: AST Request Register (ASTRR)

Description:

An AST may be requested for a particular processor mode by executing an MTPR instruction that specifies ASTRR as its destination. Processor mode encodings are those used in the Processor Status (PS); see Chapter 6, Exceptions and Interrupts, Section 6.2.

An MTPR ASTRR sets the bit corresponding to the specified processor mode in the AST Summary Register; see Page 8-6. If proper enabling conditions are present, an AST interrupt is initiated prior to issuing the next instruction; see Chapter 6, Exceptions and Interrupts, Section 6.7.5.

AST Summary Register (ASTSR)

Access:

Read

Operation:

R4 <- ZEXT(ASTSR<1:0>)

Value at System Initialization:

Zero

Format:

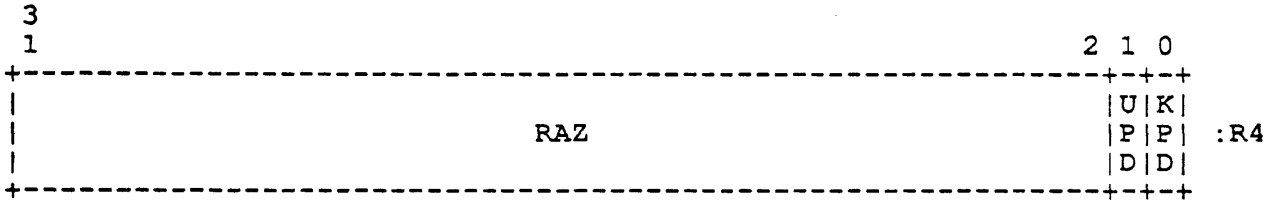


Figure 8-4: AST Summary Register (ASTSR)

Description:

The AST Summary Register records the AST pending state for each processor mode: Kernel (KPD) and User (UPD). The current AST pending state may be read by executing an MFPR instruction specifying ASTSR.

As processes are scheduled for execution, the pending AST state for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapter 4, Instruction Descriptions, Page 4-98 and Chapter 7, Process Structure.

MTPR ASTRR requests an AST at a particular processor mode and sets the corresponding pending bit in ASTSR; see Page 8-5.

When the processor IPL is 0, and proper enabling conditions are present, an AST interrupt is initiated at IPL 1 and the corresponding processor mode bit in ASTSR is cleared; see Chapter 6, Exceptions and Interrupts, Section 6.7.5.

Console Receive Control Status (CRCS)

Access:

Read/Write

Operation:

```
R4 <- CRCS                          ! Read
CRCS<0> <- R4<0>                    ! Write
```

Value at System Initialization:

Zero

Format:

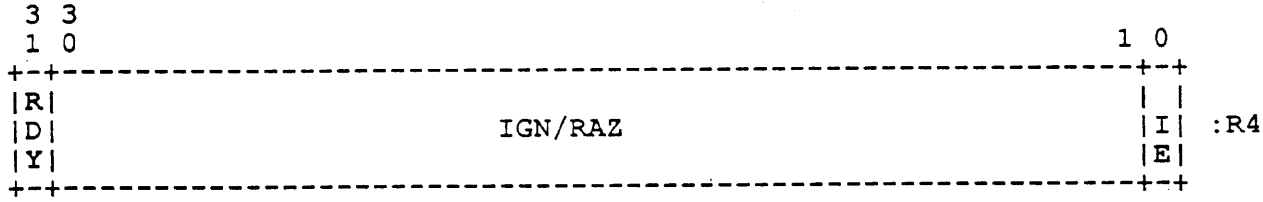


Figure 8-5: Console Receive Control Status Register (CRCS)

Description:

The Console Receive Control Status register provides access to console input status and controls whether interrupts are generated when characters are received from the console terminal; see Chapter 11, System Bootstrapping and Console, Section 11.2.

The Console Receive Control Status register may be read and written by executing MFPR and MTPR instructions that specify CRCS. When CRCS is written, 1 enables console receive interrupts and 0 disables interrupts; see Chapter 6, Exceptions and Interrupts, Section 6.3.3.1. Reading CRCS returns the current interrupt enable (IE) status and whether a character is ready (RDY) to be read from the Console Receive Data Buffer (CRDB); see Page 8-8.

Character ready (RDY) is set when a character is received from the console. If interrupts are enabled (IE set) when RDY is set, a console receive interrupt is latched and will be generated when conditions permit. If RDY is cleared (e.g. during the receipt of another character) before the interrupt is initiated, then it is UNPREDICTABLE whether the interrupt will actually be taken. However, if RDY is again set, another interrupt will be latched. If IE is cleared after an interrupt has been latched but before it has been initiated, then the latched request is cleared and the interrupt will not be taken. When the state of interrupt enable (IE) transitions from disabled (0) to enabled (1) and a character is available (RDY is set), it is UNPREDICTABLE whether a console receive interrupt is generated.

Console Receive Data Buffer (CRDB)

Access:

Read

Operation:

R4 <- CRDB

Value at System Initialization:

Undefined

Format:

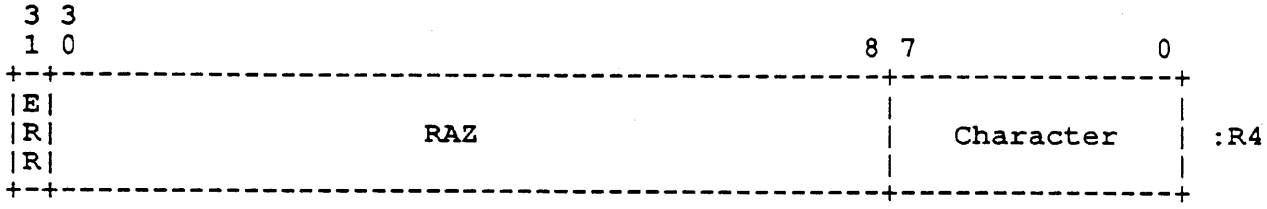


Figure 8-6: Console Receive Data Buffer Register (CRDB)

Description:

The Console Receive Data Buffer register allows characters to be read from the console by executing an MFPR instruction specifying CRDB; see Chapter 11, System Bootstrapping and Console, Section 11.2.

CRDB may be read when a character is ready for input (CRCS<RDY> is set); see Page 8-7. If CRDB is read when a character is not ready for input (CRCS<RDY> is clear), the result is UNPREDICTABLE.

Reading CRDB returns an error indication (ERR) and an 8-bit ASCII character. ERR is set if an error, such as data overrun or loss of carrier, is detected while the character is being received.

Reading CRDB clears CRCS<RDY>.

Console Transmit Control Status (CTCS)

Access:

Read/Write

Operation:

```
R4 <- CTCS                          ! Read
CTCS<0> <- R4<0>                    ! Write
```

Value at System Initialization:

10000000 (hex)

Format:

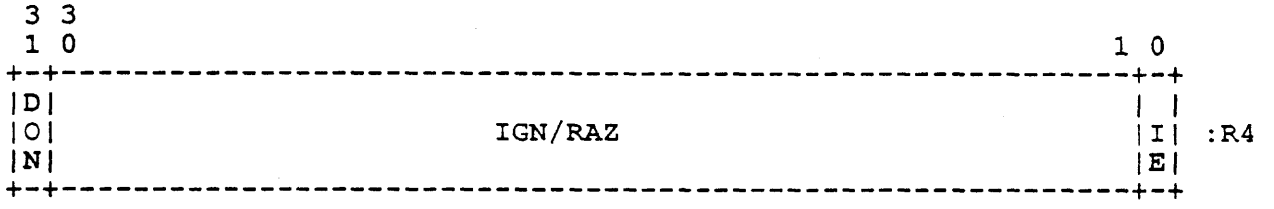


Figure 8-7: Console Transmit Control Status Register (CTCS)

Description:

The Console Transmit Control Status register provides access to console output status and controls whether interrupts are generated when characters have been transmitted to the console; see Chapter 11, System Bootstrapping and Console, Section 11.2.

The Console Transmit Control Status register may be read and written by executing MFPR and MTPR instructions that specify CTCS. When CTCS is written, 1 enables console transmit interrupts and 0 disables interrupts; see Chapter 6, Exceptions and Interrupts, Section 6.3.3.2. Reading CTCS returns the current interrupt enable (IE) status and whether a character can be transmitted (DON) to the Console Transmit Data Buffer (CTDB); see Page 8-10. Character done (DON) is cleared when a character is written to CTDB and set when the character has been transmitted to the console. If interrupts are enabled (IE set) when DON is set, a console transmit interrupt is latched and will be generated when conditions permit. If DON is cleared (e.g. another character is written to CTDB) before the interrupt is initiated, it is UNPREDICTABLE whether the interrupt will actually be taken. However, if DON is again set, another interrupt will be latched. If IE is cleared after an interrupt has been latched but before it has been initiated, then the latched request is cleared and the interrupt will not be taken.

When the state of interrupt enable transitions from disabled (0) to enabled (1) and a character has finished transmission (DON is set), it is UNPREDICTABLE whether a console transmit interrupt is generated.

Console Transmit Data Buffer (CTDB)

Access:

Write

Operation:

CTDB <- R4<7:0>

Value at System Initialization:

Not Applicable

Format:

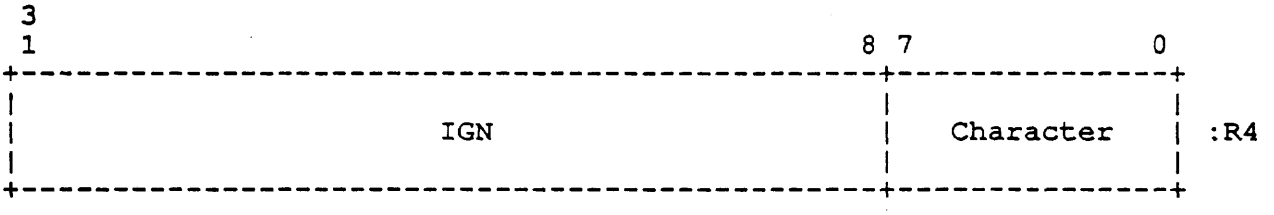


Figure 8-8: Console Transmit Data Buffer Register (CTDB)

Description:

The Console Transmit Data Buffer register allows 8-bit ASCII characters to be written to the console by executing an MTPR instruction specifying CTDB; see Chapter 11, System Bootstrapping and Console, Section 11.2.

CTDB may be written when any previously written characters have been transmitted (CTCS<DON> is set); see Page 8-9. If CTDB is written when a character is currently being transmitted (CTCS<DON> is clear), the result is UNPREDICTABLE.

Writing CTDB clears CTCS<DON>.

Interval Clock Interrupt Enable (ICIE)

Access:

Read/Write

Operation:

```
R4 <- ZEXT(ICIE<0>)              ! Read

ICIE <- R4<0>                    ! Write
```

Value at System Initialization:

Zero

Format:

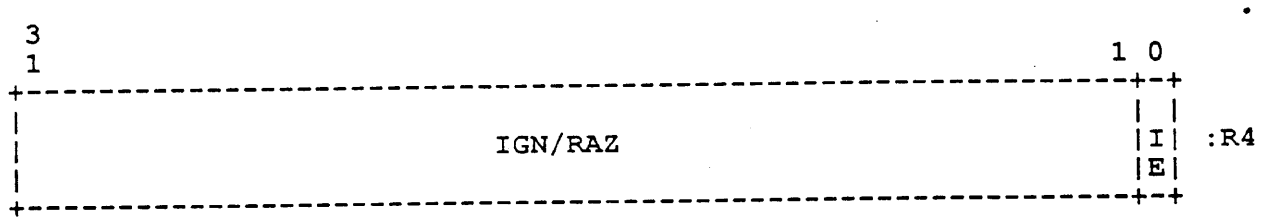


Figure 8-9: Interval Clock Interrupt Enable Register (ICIE)

Description:

The Interval Clock provides the capability to regularly interrupt the processor at 1 millisecond intervals. The interval clock has an accuracy of .0025% or better (approximately 65 seconds per month). The Interval Clock Enable register controls whether clock interrupts are enabled or disabled.

The Interval Clock Interrupt Enable register may be read and written by executing MFPR and MTPR instructions that specify ICIE. When ICIE is written, 1 enables clock interrupts and 0 disables interrupts. After enabling Interval Clock interrupts, the first interrupt may occur in less than 1 millisecond.

Interval Clock interrupts are initiated at IPL 6; see Chapter 6, Exceptions and Interrupts, Section 6.3.5.1.

Interprocessor Interrupt Enable (IPIE)

Access:

Read/Write

Operation:

```
R4 <- ZEXT(IPIE<0>)              ! Read

IPIE <- R4<0>                    ! Write
```

Value at System Initialization:

Zero

Format:

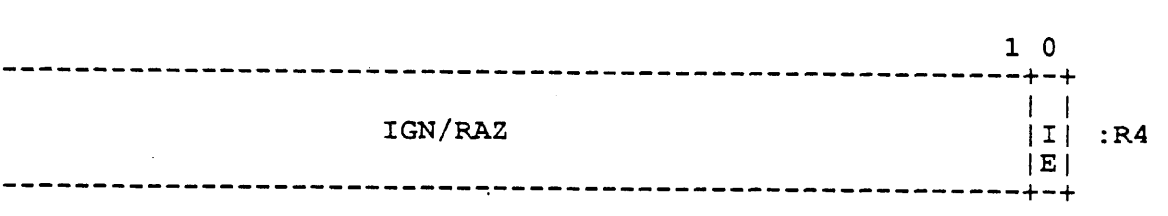


Figure 8-10: Interprocessor Interrupt Enable Register (IPIE)

Description:

The PRISM architecture provides the capability for one processor to interrupt another processor via an IPR; see Page 8-13. The Interprocessor Interrupt Enable register controls whether interprocessor interrupts are enabled or disabled.

The Interprocessor Interrupt Enable register may be read and written by executing MFPR and MTPR instructions that specify IPIE. When IPIE is written, 1 enables interprocessor interrupts and 0 disables interrupts.

An interprocessor interrupt is initiated when interprocessor interrupts are enabled, an interprocessor interrupt request has been received from another processor, and the current IPL is less than 6.

Interprocessor interrupts are initiated at IPL 6; see Chapter 6, Exceptions and Interrupts, Section 6.3.5.2.1.

The value returned in bit <0>, when this register is read on a uniprocessor, is UNPREDICTABLE.

Interprocessor Interrupt Request (IPIR)

Access:

Write

Operation:

IPIR <- R4

Value at System Initialization:

Not applicable

Format:

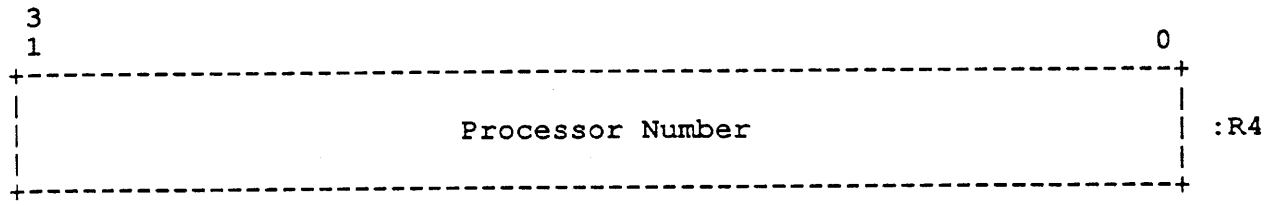


Figure 8-11: Interprocessor Interrupt Request Register (IPIR)

Description:

An interprocessor interrupt can be requested on a specified processor by executing an MTPR instruction specifying IPIR. The interrupt request is recorded on the target processor and is initiated when proper enabling conditions are present; see Page 8-12.

If the target processor is the same as the current processor, whether or not an interprocessor interrupt is initiated is UNPREDICTABLE.

Machine Check Error Summary Register (MCES)

Access:

Read/Write

Operation:

```
R4 <- MCES                                    ! Read

if {R4<0> EQ 1} then MCES<0> <- 0             ! Write
```

Value at System Initialization:

Zero

Format:

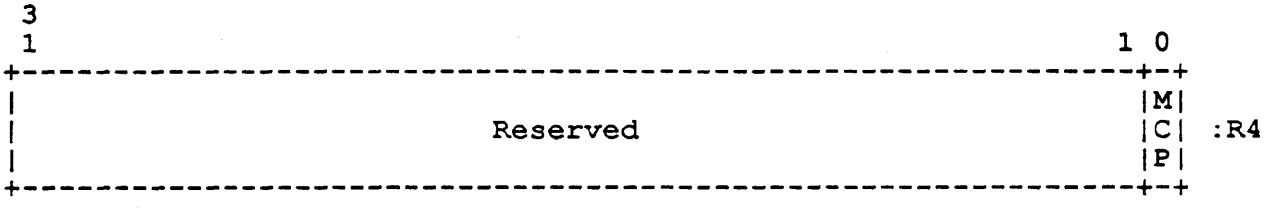


Figure 8-12: Machine Check Error Summary Register (MCES)

Description:

Machine Check in Progress (MCP) is set by the hardware when a machine check occurs. Writing a 1 to this bit clears it. The MCP bit is cleared by the operating system machine check handler before it exits. Reserved bits may be used to report implementation-specific errors.

This IPR is used to detect double machine checks.

Privileged Context Block Base (PCBB)

Access:

Read

Operation:

QR4 <- ZEXT(PCBB)

Value at System Initialization:

See Chapter 11, System Bootstrapping and Console.

Format:

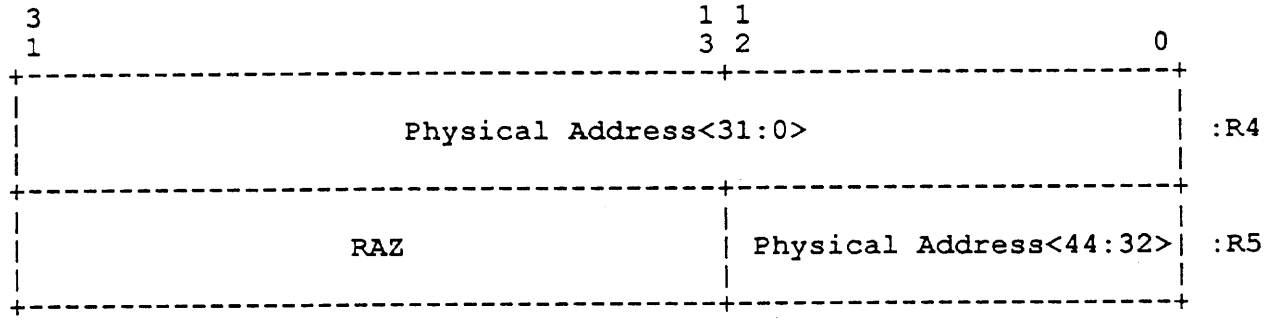


Figure 8-13: Privileged Context Block Base Register (PCBB)

Description:

The Privileged Context Block Base register contains the physical address of the privileged context block for the current process. It may be read by executing an MFPR instruction specifying PCBB.

PCBB is written by the Swap Privileged Context (SWPCTX) instruction; see Chapter 4, Instruction Descriptions, Page 4-98 and Chapter 7, Process Structure.

Processor Base Register (PRBR)

Access:

Read/Write

Operation:

```
R4 <- PRBR                    ! Read

PRBR <- R4                    ! Write
```

Value at System Initialization:

Undefined

Format:

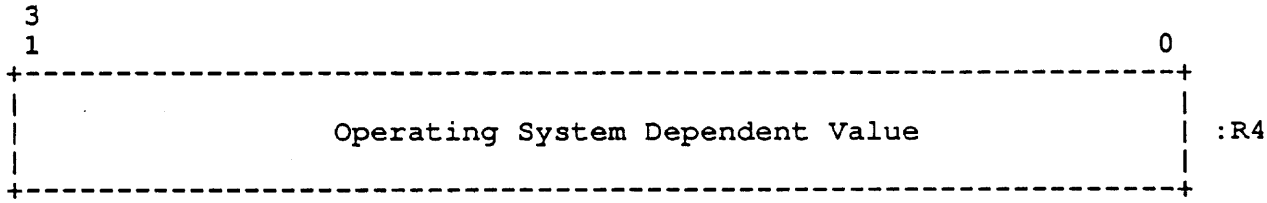


Figure 8-14: Processor Base Register (PRBR)

Description:

In a multiprocessor system, it is desirable for the operating system to be able to locate a processor-specific data structure in a simple and straightforward manner. The Processor Base Register provides a longword of operating system-dependent state that can be read and written via MFPR and MTPR instructions that specify PRBR.

Page Table Base Register (PTBR)

Access:

Read

Operation:

R4 <- PTBR

Value at System Initialization:

See Chapter 11, System Bootstrapping and Console

Format:

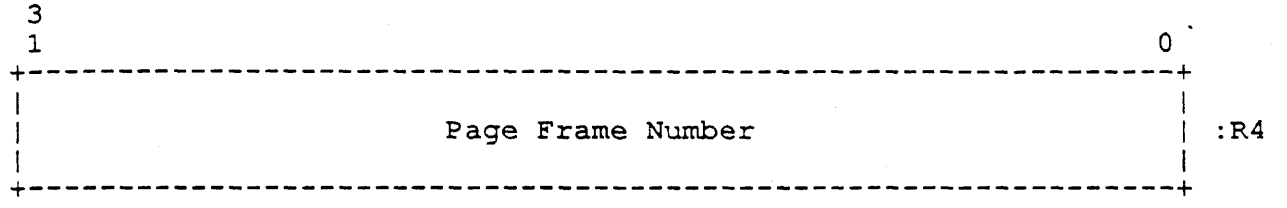


Figure 8-15: Page Table Base Register (PTBR)

Description:

The Page Table Base Register contains the page frame number of the first-level page table for the current process. It may be read by executing an MFPR instruction specifying PTBR; see Chapter 5, Memory Management.

As processes are scheduled for execution, the PTBR for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapter 4, Instruction Descriptions, Page 4-98 and Chapter 7, Process Structure.

System Control Block Base (SCBB)

Access:

Read/Write

Operation:

```
QR4 <- ZEXT(SCBB)                ! Read

SCBB <- QR4                      ! Write
```

Value at System Initialization:

Undefined

Format:

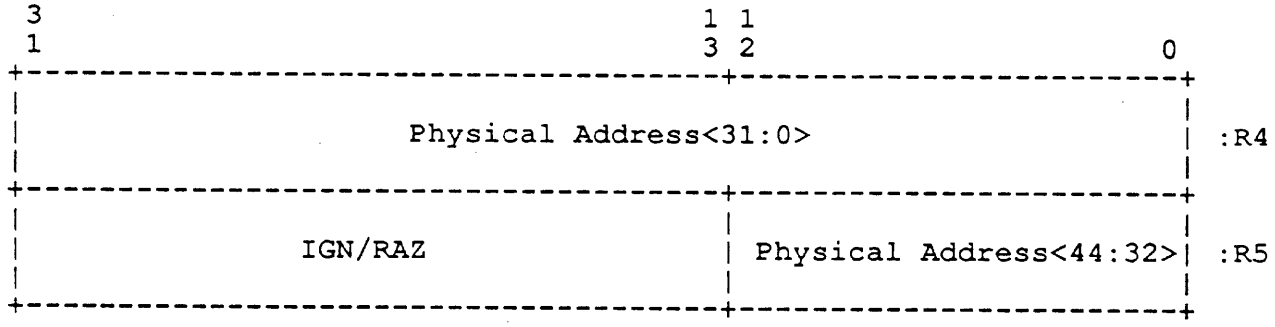


Figure 8-16: System Control Block Base Register (SCBB)

Description:

The System Control Block Base register holds the physical address of the System Control Block which is used to dispatch exceptions and interrupts and may be read and written by executing MFPR and MTPR instructions that specify SCBB; see Chapter 6, Exceptions and Interrupts, Section 6.6.

When SCBB is written, the specified physical address must be the quadword aligned address of a contiguous 8 Kbyte block which is neither in I/O space nor non-existent memory. Otherwise, UNDEFINED operation will result.

System Identification (SID)

Access:

Read

Operation:

QR4 <- SID

Value at System Initialization:

System Identification

Format:

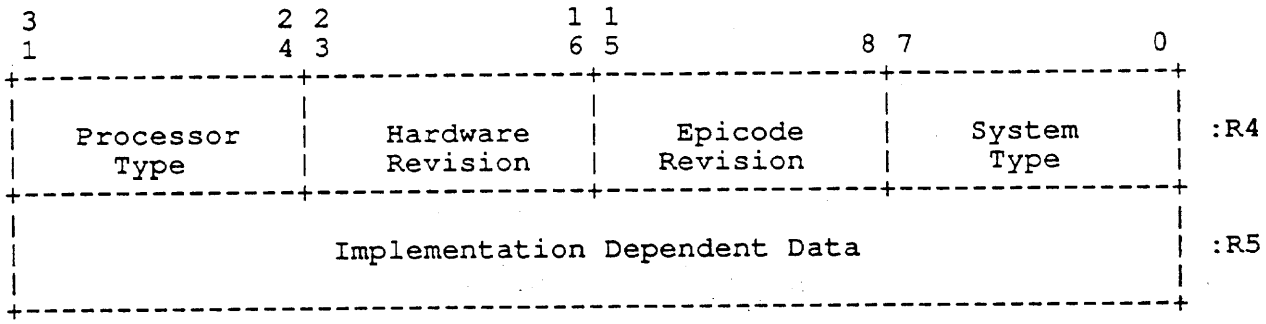


Figure 8-17: System Identification Register (SID)

Description:

The System Identification register provides information about the processor type, hardware and Epicode revision levels, system type, and implementation dependent information.

The System Identification register may be read by executing an MFPR instruction specifying SID.

Software Interrupt Request Register (SIRR)

Access:

Write

Operation:

SIRR <- R4<1:0>

Value at System Initialization:

Not applicable

Format:

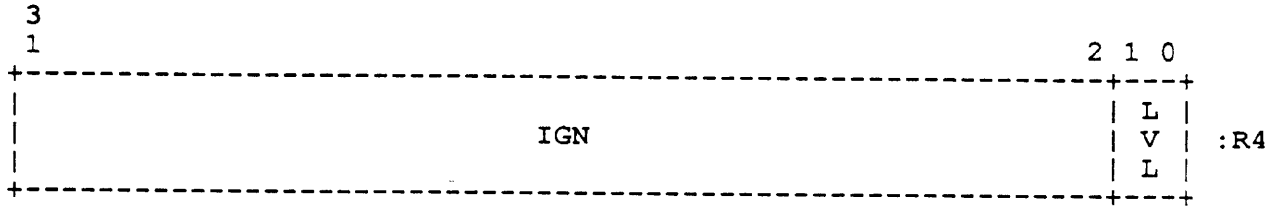


Figure 8-18: Software Interrupt Request Register (SIRR)

Description:

A software interrupt may be requested for a particular Interrupt Priority Level (IPL) by executing an MTPR instruction specifying SIRR. Software interrupts may be requested at levels 0, 1, 2, and 3 (requests at level 0 are ignored).

An MTPR SIRR sets the bit corresponding to the specified interrupt level in the Software Interrupt Summary Register; see Page 8-21. If proper enabling conditions are present, a software interrupt is initiated prior to issuing the next instruction; see Chapter 6, Exceptions and Interrupts, Sections 6.3.2 and 6.7.5.

Software Interrupt Summary Register (SISR)

Access:

Read

Operation:

R4 <- ZEXT(SISR<3:0>)

Value at System Initialization:

Zero

Format:

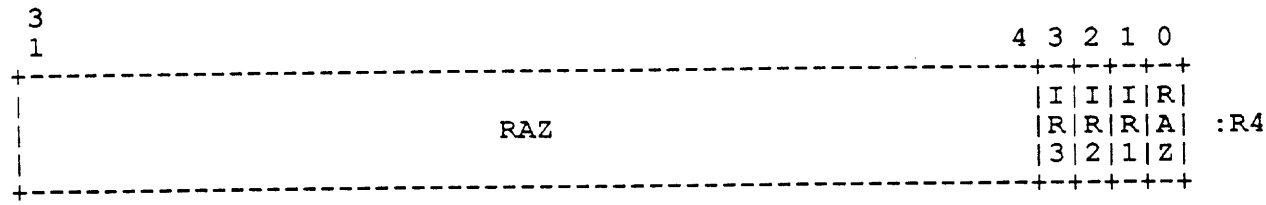


Figure 8-19: Software Interrupt Summary Register (SISR)

Description:

The Software Interrupt Summary Register records the interrupt pending state for each of the interrupt levels 1, 2, and 3. The current interrupt pending state may be read by executing an MFPR instruction specifying SISR.

MTPR SIRR requests an interrupt at a particular interrupt level and sets the corresponding pending bit in SISR; see Page 8-20.

When the processor IPL falls below the level of a pending request, an interrupt is initiated and the corresponding bit in SISR is cleared; see Chapter 6, Exceptions and Interrupts, Sections 6.3.2 and 6.7.5.

System Serial Number (SSN)

Access:

Read

Operation:

```
IF {implemented} THEN
    R4 <- SSN
ELSE
    R4 <- 0
```

Value at System Initialization:

System serial number or zero

Format:

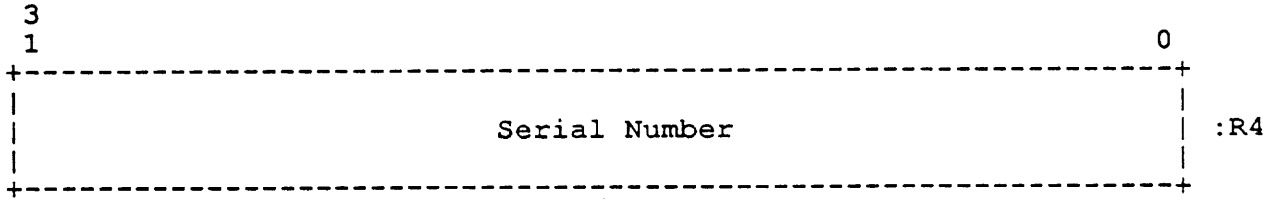


Figure 8-20: System Serial Number Register (SSN)

Description:

The System Serial Number register provides access to the system serial number by executing an MFPR instruction specifying SSN.

Implementation of serial numbers is optional. If implemented, the serial number is returned. Otherwise, zero is returned (zero is an invalid serial number).

Translation Buffer Check (TBCHK)

Access:

Read

Operation:

```
R6 <- 0
IF {implemented} THEN
    R6<0> <- {entry in TB using R4<15:0>, R5}
ELSE
    R6<31> <- 1
```

Value at System Initialization:

Correct results are always returned

Format:

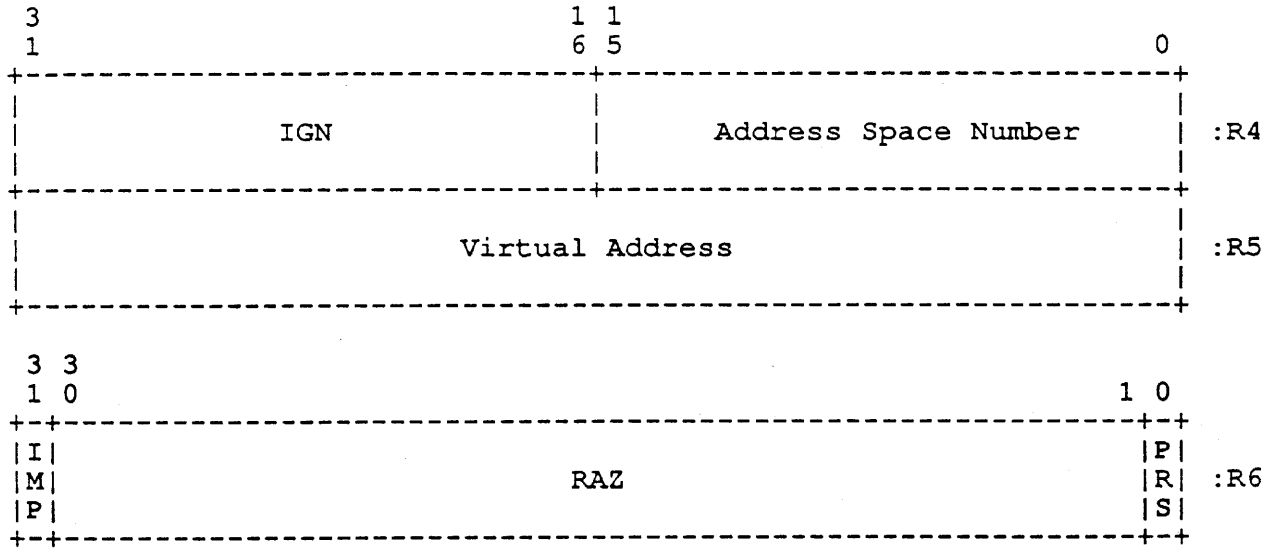


Figure 8-21: Translation Buffer Check Register (TBCHK)

Description:

The Translation Buffer Check register provides the capability to determine if a virtual address is present in the Translation Buffer by executing an MFPR instruction specifying TBCHK; see Chapter 5, Memory Management.

A virtual address and Address Space Number (ASN) are specified as input (if ASNs are not implemented, ASN is ignored). The virtual address can be any address within the desired page. The value read contains an indication of whether the function is implemented and whether the virtual address is present in the Translation Buffer.

If the function is not implemented, a value is returned with bit <31> set and bit <0> clear. Otherwise, a value is returned with bit <31> clear and bit <0> indicates whether the virtual address is present (1) or absent (0) in the Translation Buffer.

The TBCHK register can be used by system software for working set management.

Translation Buffer Invalidate Single (TBIS)

Access:

Write

Operation:

{Invalidate single TB entry using R4<15:0>, R5}

Value at System Initialization:

Not applicable

Format:

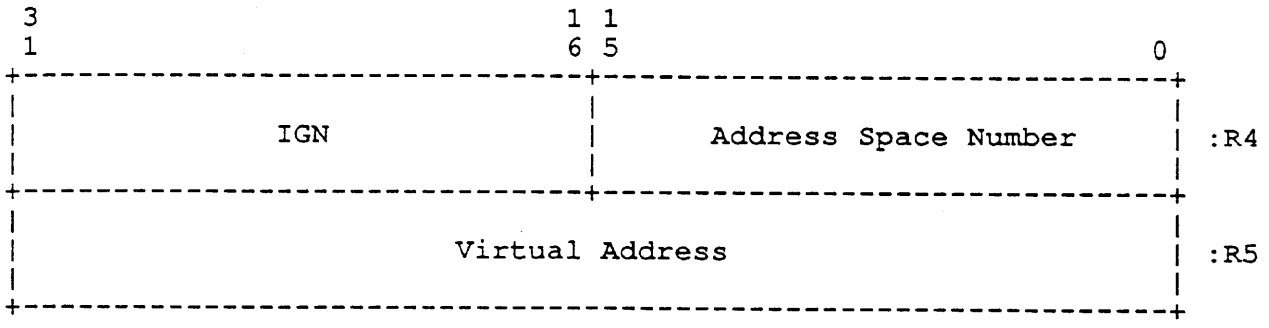


Figure 8-22: Translation Buffer Invalidate Single Register (TBIS)

Description:

The Translation Buffer Invalidate Single register provides the capability to invalidate a single entry in the Translation Buffer by executing an MTPR instruction specifying TBIS; see Chapter 5, Memory Management.

A virtual address and Address Space Number (ASN) are specified as input (if ASNs are not implemented, ASN is ignored). The virtual address can be any address within the desired page.

Time Of Year (TOY)

Access:

Read/Write

Operation:

R4 <- TOY ! Read

TOY <- R4 ! Write

Value at System Initialization:

Correct time or invalid time indication

Format:

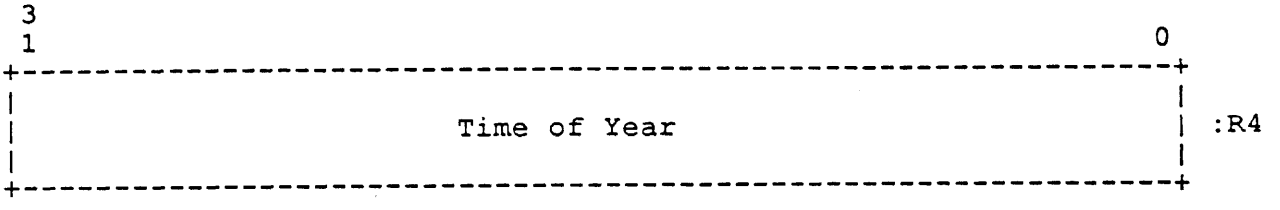


Figure 8-23: Time of Year Register (TOY)

Description:

The Time Of Year register provides the capability to read and write the current time from a battery backed-up source by executing MFPR and MTPR instructions that specify TOY. Access to this register may be very slow (e.g., many milliseconds).

The Time-of-Year clock consists of one longword register. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source with at least .0025% accuracy (approximately 65 seconds per month). The least significant bit of the counter represents a resolution of 10 milliseconds. Thus the counter cycles to 0 after approximately 497 days.

The counter has an optional battery back-up power supply sufficient for at least 100 hours of operation, and the clock does not gain or lose any ticks during transition to or from standby power. The battery is recharged automatically. If the battery has failed, so that the time is not accurate, then the register is cleared upon powerup. The register stays at 0 until the software writes a non-zero value into it. It counts only when it contains a non-zero value.

User Stack Pointer (USP)

Access:

Read/Write

Operation:

```
R4 <- USP                          ! Read

USP <- R4                          ! Write
```

Value at System Initialization:

Undefined

Format:

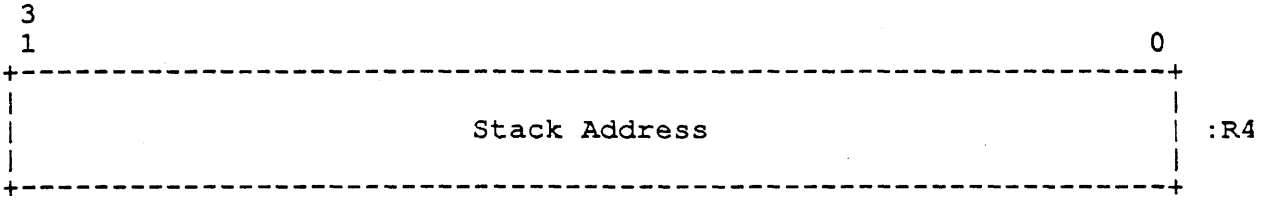


Figure 8-24: User Stack Pointer (USP)

Description:

This register allows the stack pointer for User mode (USP) to be read and written via MFPR and MTPR instructions that specify USP.

The current stack pointer may be read and written directly by specifying scalar register SP (R1).

No internal processor register is provided to read and write the Kernel stack pointer. MxPR instructions can only be executed from Kernel mode, and while in Kernel mode, the current (Kernel mode) stack pointer can be directly read and written.

As processes are scheduled for execution, the two stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapter 4, Instruction Descriptions, Page 4-98 and Chapter 7, Process Structure.

Vector Enable Register (VEN)

Access:

Write

Operation:

PS<VEN> <- R4<0>

Value at System Initialization:

Not Applicable

Format:

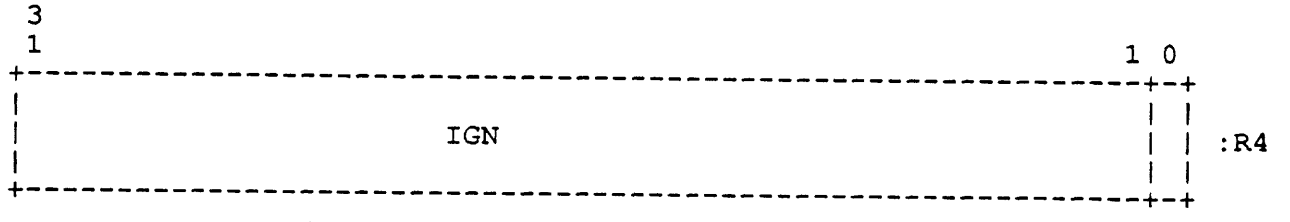


Figure 8-25: Vector Enable Register (VEN)

Description:

This register is used to set or clear the Vector Enable bit in the Processor Status (PS<VEN>); see Chapter 6, Exceptions and Interrupts, Section 6.2. The VEN register can be read with a MOVPS instruction.

Who-Am-I (WHAMI)

Access:

Read

Operation:

R4 <- WHAMI

Value at System Initialization:

Processor number

Format:

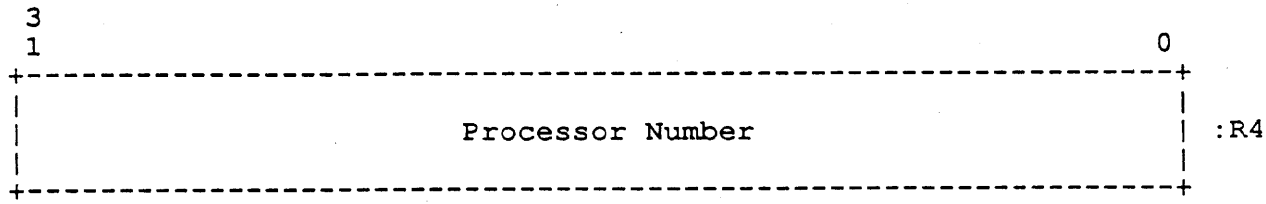


Figure 8-26: Who-Am-I Register (WHAMI)

Description:

The Who-Am-I register provides the capability to read the current processor number by executing an MFPR instruction specifying WHAMI. The processor number returned is in the range 0 to the number of processors in the configuration minus 1.

The current processor number is useful in a multiprocessing system to index arrays that store per processor information. Such information is operating system dependent.

Revision History:

Revision 3.0, 26 April 1988

1. Make ASTEN read/write.
2. Add Machine Check in Progress register.
3. Delete CCYCR as an IPR.
4. Rename Processor ID to System ID.
5. Add Vector Enable IPR.
6. Change TOY clock to be a 32-bit binary counter with 10 ms resolution.

Revision 2.0, 24 June 1986.

1. Change number of stack pointers to 2 and delete ESP and SSP internal processor registers.
2. Delete TBIASN internal processor register.
3. Add Cycle Count Register (CCYCR) internal processor register.
4. Change interval timer interval to 1ms and to interrupt at IPL 6.
5. Change ASTEN, ASTRR, and ASTSR to supply bits for Kernel and User mode.
6. Clarify console receive and transmit interrupts.
7. Clarify range of process number returned by WHAMI.

Revision 1.0, 22 December 1985

1. Removed the following Internal Process Registers:
    1. ISP - Interrupt Stack Pointer
    2. KSP - Kernel Stack Pointer
    3. PBR - Process Page Table Base Register
    4. SBR - System Page Table Base Register
    5. IPL - Interrupt Priority Level
    6. ASTLVL - AST Level
    7. ASNSIZ - Address Space Number Size
    8. PME - Performance Monitor Enable
    9. PAGSIZ - Page Size

10. BOOTFLAGS - Bootstrap Flags

2. Added the following Internal Processor Registers:
    1. CRCS - Console Receive Control Status
    2. CRDB - Console Receive Data buffer
    3. CTCS - Console Transmit Control Status
    4. CTDB - Console Transmit Data Buffer
    5. PTBR - Page Table Base Register
    6. PCBB - Privileged Context Block Base
    7. ASTRR - AST Request Register
    8. ASTSR - AST Summary Register
    9. ASTEN - AST Enable Register

3. Changed the following Internal Processor Register names:
    1. ICCS changed to ICIE
    2. CPUSN changed to PRSN
    3. CPUBR changed to PRBR

4. Changed parameter registers to R4, R5, R6.
5. Changes to reflect new 32 bit register sizes.
6. PTBR changed from address to page frame number.
7. Added system type to SID.
8. Eliminated zero default in ASN parameters.
9. Corrected accuracy of timer and clock.
10. Removed duplicate material and added pointers to other chapters.

Revision 0.0, July 5, 1985

1. First review distribution.

CHAPTER 9

SYSTEM ARCHITECTURE AND PROGRAMMING IMPLICATIONS

9.1 INTRODUCTION

Portions of the PRISM architecture have implications for programming and the system structure of implementations. Architectural implications considered in the following sections are:

- Data sharing and synchronization
- Separation of procedures and data
- Translation Buffers
- Caches
- Stacks

To meet the requirements of the PRISM architecture, software and hardware implementors must take these issues into consideration.

9.2 MEMORY, MULTIPROCESSING, AND INTERPROCESSOR COMMUNICATION

A PRISM system comprises one or more PRISM processors and one or more I/O devices which all share a common memory. The PRISM architecture specifies how these elements communicate with each other. It specifies how data is shared between the processors and I/O devices and how the processors and I/O devices signal each other. It includes rules for the ordering of write operations and interrupts and for mutual exclusion to avoid data corruption by simultaneous reads and writes from multiple processors.

Many contemporary I/O devices really behave like any other PRISM processor in an MP system in that they communicate control information through a shared data structure in memory. This type of I/O device is termed a "smart" I/O device. "Simple" I/O devices have no shared control structure in memory and are typically controlled by writing several device CSRs to describe a transfer and a "GO" bit in the device to initiate it.

"Smart" I/O devices are really part of the PRISM MP processor set and must use the same method of synchronizing with a PRISM processor as two or more PRISM processors use to synchronize with each other. When the term "processor" is used without qualification, it refers to both

PRISM processors and "smart" I/O devices.

9.2.1 The Ordering Of Writes And Interrupts -

PRISM systems use the ordering of events as the mechanism which programs and hardware devices running in parallel can use to coordinate their operations. There is no method in a PRISM system for two or more processors or I/O devices to coordinate based on some time standard held in common. The coordination is primarily based on the ordering of interlocked operations.

This section specifies how programs and hardware devices in a multiprocessing system see the effects of data written from different processors or I/O devices. The rules apply to both memory-space and I/O-space references unless otherwise stated. A write takes effect for some observer when that observer, running in some processor or I/O device, can issue a read which returns the updated value.

1. When only one processor or I/O device is doing both the reading and writing to memory, a read returns the value of the most recent write to the same location. The value read from a memory-space location never written is UNPREDICTABLE.

2. When a single processor or I/O device makes a series of non-interlocked writes to different locations in memory, other observers may see the writes take effect in any order.

   For example, a program in one processor writes into location X and later writes into location Y; a second program in another processor reads location Y and then reads location X. Depending on the relative timing of the two programs, the second program may read old X and old Y, new X and old Y, new X and new Y, or old X and new Y.

3. When a single processor makes a series of I/O space references (reads or writes) to the same I/O device via the same bus or bus adaptor, the references are made in the same order as the program initiated them.

4. A processor write to I/O space provides no guarantee that its previous memory writes are visible to the I/O device should the device access PRISM memory. To provide such a guarantee, an interlocked instruction must be executed before the device references memory. This is described in more detail later in the chapter.

5. When two or more processors or I/O devices each write (without a synchronizing signal, see the next section), the observed order of the writes is UNPREDICTABLE and the observed final value is UNPREDICTABLE. Observers in different processors or I/O devices may see the writes take effect in different orders and the observed order of writes may differ from trial to trial. In addition, since there is no synchronizing signal between the processors, there is no way for the processors to determine when the final values become valid. Also, in the absence of a synchronizing signal, an observer is not guaranteed to have a coherent view of memory and therefore may not see the final value.

For example, consider two programs in two processors, P0 and P1, each writing to its own variable and reading from the other's variable. The programs are not synchronized.

P0	P1
writes X	writes Y
reads Y	reads X

Each processor can read either the new or old copy of the other's variable. Which is read depends on unpredictable timing. There are four combinations, all of which can occur:

P0 reads	P1 reads
new Y	new X
old Y	new X
new Y	old X
old Y	old X

Note particularly the last combination: both programs can read the old value of the other's variable and come to different conclusions about the write order. P0 will conclude that X was written but not Y; P1 will conclude that Y was written but not X.

In Dekker's mutual exclusion algorithm [Dijkstra E.W.(1965): Co-operating sequential processes; in "Programming Languages," F. Genuys (Ed.); Academic Press, New York, pp. 43-112.], the lock variables are the X and Y above. The algorithm relies on processors having the same view of the lock variables, and therefore it will not work on PRISM systems. For mutual exclusion, the PRISM architecture requires that interlocked instructions be used.

6. When two or more processors each make interlocked writes to the same physical address, different observers using interlocked accesses see the interlocked writes take effect in the same order. If the interlocking references map to different physical addresses, the observed write order is as if they were not interlocked, and rule 5 applies.

Consider the rule 5 example modified by adding interlocked instructions.

P0	P1
writes X	writes Y
RMAQI on J	RMAQI on K
reads Y	reads X

If J and K reference the same physical location, then either P0 or P1 will read a new value. The two RMAQI instructions are ordered relative to each other although which one executes first is unpredictable. If P0's RMAQI occurs before P1's RMAQI, then P0's new value of X will be read by P1, although P0 may still read either the old or the new value of Y. If P1's RMAQI occurs before P0's RMAQI, then P1's new value of Y will be read by P0, although P1 may still read either the old or the new value of X.

If J and K reference different physical locations, then the example is identical to that for rule 5: it is possible for both P0 and P1 to read old values.

7. When a processor writes and then requests an interrupt of a target processor, it is UNPREDICTABLE whether the write takes effect in the target before or after the target initiates the interrupt.

8. "Simple" I/O devices operate under the following rules:

   1. No caching of memory data for the "simple" device is allowed. This does not prohibit prefetching output data or write buffering input data during a single I/O transfer operation.

   2. On the PRISM processor - Before software writes to an I/O space CSR on the "simple" device that causes the device to access memory, all prior memory writes must be made visible to the "simple" device. To accomplish this, software must execute an interlocked instruction on a common control variable before it writes to I/O space, causing the device to access memory.

   3. On the "simple" I/O device - Before the device signals an interrupt, all of its prior memory writes must be made visible to the PRISM processors. To accomplish this, I/O adapters and I/O devices must ensure that all prior memory writes by the device have "completed" either before the device interrupt is signalled to a PRISM processor or before the requested interrupt vector is delivered to the PRISM processor. In this context, "completed" means that all of the I/O device writes to memory are guaranteed to be visible to any processor once it performs an interlocked operation on a common control variable.

      For example, on some implementations "completing" an I/O write to memory ensures that any necessary write updates or invalidates have been distributed to other caches in the system. But it does not mean that those caches have processed the updates or invalidates yet. In addition, software on the PRISM processor responding to the interrupt must execute an interlocked instruction on a common control variable before it attempts to read memory data. This ensures that any updates or invalidates that are pending at the PRISM processor's cache are processed.

   4. The PRISM architecture makes no guarantees about data visibility between two PRISM processors when a "simple" I/O device is used as a signaling mechanism between the PRISM processors. Interlocked instructions must be executed on a common control variable for this communication to work.

      For example, PRISM processor P0 writes memory, starts I/O, the I/O completes and signals PRISM processor P1 with an interrupt, and P1 then references memory. No guarantees are made about the ability of P1 to see P0's memory writes. In order for this to work, P0 and P1 must

execute interlocked operations on a common control variable. Typically, software on P0 would execute an interlocked operation before writing the device CSRs in order to acquire control of the resource. To service the interrupt, software on P1 would execute an interlocked operation on the same control variable thereby ensuring data visibility between PRISM processors.

9.2.2 Memory And Shared Data

It is the intent of the PRISM architecture that access to shared memory data be synchronized using interlocked instructions. The interlocked instructions accomplish two things. First, they provide a means for implementing mutual exclusion by ensuring that one and only one processor can access the interlock control variable at a time, preventing multiple simultaneous accesses from corrupting the control variable. And second, they provide the mechanism the hardware uses to make modified data visible to other processors and "smart" I/O devices.

NOTE

\In the VAX architecture, many instructions provide noninterruptable read-modify-write sequences to memory variables. In the VAX, most of the data sharing is more an issue for hardware implementors and a few system programmers. Most programmers never regard data sharing as an issue. In the PRISM architecture, programmers will have to pay more attention to synchronizing access to shared data. One of the major areas this may show up in is AST routines. In the VAX, a programmer can use an ADDL2 to update a variable shared between a "MAIN" routine and an AST routine if running on a single processor. In the PRISM architecture, a programmer will have to deal with AST routines as if they could be run on different processors. \

9.2.2.1 Interprocessor Signaling And Data Visibility

A program must issue an interlocked instruction to memory space as the signal to ensure that all observers can read the signal itself and all the previously-written shared data. If a program does not issue this signal, it is UNPREDICTABLE whether another observer will read the old or new shared data.

The normal interlocked software mechanism for handling shared data accomplishes the required interprocessor signaling. This signaling mechanism is outlined below: (One way to acquire and release ownership of the software control variable is described in Section 9.2.3, Page 9-11.)

```
{Acquire ownership of software control-          (1)
 variable using an interlocked instruction}

{Issue DRAINM instruction if reading any        (2)
 memory data with a vector instruction}

    .
    .  Read and/or write shared data            (3)
    .  using non-interlocked instructions
    .

{Issue DRAINM instruction if any memory data    (4)
 was written with a vector instruction}

{Release ownership of software control-         (5)
 variable using an interlocked instruction}
```

Notes -

1. A user of shared data first acquires ownership of the memory area containing the shared data by performing an interlocked operation on the same control variable that is used by other users of the shared data. In addition to synchronizing access to the shared data, this interlocked instruction guarantees that the user has a coherent view of the shared data using scalar memory reference instructions. For example, on some implementations it ensures that all write updates or invalidates that are pending at the processor's scalar cache are processed before the interlock operation is performed. Note that this only affects the scalar processor cache and thus only ensures a coherent view of memory as seen by scalar memory reference instructions.

2. If the user is going to read shared memory data with vector instructions, software must insert a DRAINM instruction after the interlock instruction and before the first vector load or gather instruction to ensure a coherent view of shared data using vector memory reference instructions. For example, on some implementations the DRAINM would ensure that all write updates or invalidates that are pending at the vector cache are processed before a subsequent vector instruction attempts to reference the data.

3. At this point the user has a coherent view of memory and can read or write the shared data using the normal scalar and vector load/store instructions.

4. If the user wrote shared memory data with vector instructions, software must insert a DRAINM instruction after the last vector store or scatter instruction and before the interlock instruction to ensure that all vector store instructions have finished executing and their writes have "completed". See the next item for a definition of "completed".

5. The user releases ownership of the shared data by performing another interlocked operation on the control variable. The interlocked operation updates the control variable and forces

all previous scalar writes by this user to be "completed". In this context, "completed" means that all previous writes by this processor are guaranteed to be visible to the next processor that performs an interlocked operation on the same control variable for the purpose of acquiring access to shared data. For example, on some implementations it ensures that any necessary write updates or invalidates have been distributed to other caches in the system. But it does not mean that those caches have processed the updates or invalidates yet.

NOTE

\Implementation note: some hardware scheme must be implemented to ensure that the signals listed above produce the correct software effect. This is a particular concern when write-buffers or write-back caches are involved. Draining a write-buffer or cache before the signal is sent will work. Implementing a "snoopy" write-buffer or cache, which monitors and responds to requests from other processors or I/O devices, can also work. Other hardware schemes are possible.\

The preceeding description is based on a mutual exclusion or critical section viewpoint. Another model consists of problem decomposition or data partitioning in which a problem is split among multiple processors. An example would be a loop in which each processor does a different set of iterations. In this case, the work assignments (loop iterations) would be acquired through the use of interlocked instructions, such as RMAQI, and each processor would work on different data elements so there is no real data sharing between the processors. However, after all the loop iterations are completed it is necessary for the processors to synchronize on a common control variable if subsequent execution requires a processor's results to be visible to any other processor.

There are two functions involving the console that are also interprocessor signals.

1. Processor entry to console mode as the result of a HALT (from kernel mode or from the front console panel) or an error halt. The processor is making a major state change after which it is no longer a cooperating member of the system. \ A possible hardware implementation is to drain all internal buffering when either signal occurs. \

2. Any console function that changes shared memory state (e.g. deposit).

9.2.2.2 Atomicity And Corruption

A common memory allows data to be shared among the processors and I/O devices of a PRISM system. Multiple simultaneous accesses may cause some data to become "corrupted" while other "atomic" accesses keep the

data intact. The following two definitions precisely describe atomicity and corruption:

- Atomicity -- The access of a shared datum in memory by one processor or I/O device is classified as atomic (indivisible) or non-atomic (divisible) with respect to an access of the same datum by another processor or I/O device. A memory access is atomic when any other memory access to the same datum may occur before or after but not during the first access. A memory access is non-atomic when another memory access to the same datum may occur before, during, or after the first access.

- Corruption -- A shared datum accessed from memory non-atomically can become corrupted: the datum when written can become any byte-wise combination of all the data that is concurrently being written to it; and when read, the value of the datum may be any byte-wise combination of its original value with any new values that are concurrently being written to it.

  Corruption can occur in two ways: from the non-atomic access of two or more writers, in which case the corrupted value will remain in the memory location until some program reinitializes it; or from the non-atomic accesses of a single writer and one or more readers, in which case only the read accesses may be corrupted, and subsequent read accesses may return an uncorrupted value. A datum (or its value) cannot become corrupted if it is not shared, if it is always read and written by atomic accesses, or if it is accessed by multiple readers and no writers.

To support shared data, a PRISM system meets these requirements:

1. The memory ensures that the granularity of access for independent modification by any processor or I/O device is the longword. This does not imply a maximum reference size of one longword but only that independent modifying accesses to adjacent longwords produce the same results regardless of the order of execution. Systems may choose to do masked writes (less than longword) in the cache by reading the needed longword from memory, merging it in the cache, and then writing the longword back to memory, thereby only supporting longword writes to the main memory system.

   For example, suppose locations 0 and 4 contain the values 5 and 6. Suppose one processor executes BYTE STORE of a 6 to location 0 and another executes BYTE STORE of a 7 to location 4. Then, regardless of the order of execution, including effectively simultaneous execution, the final contents of locations 0 and 4 must be 6 and 7.

   As a second example, suppose locations 0 and 1 contain the values 5 and 6. Suppose one processor does a BYTE STORE of a 6 in memory at location 0. Also, suppose a second processor does a BYTE STORE of a 7 in memory at location 1. After both processors finish execution of the sequences the results are UNPREDICTABLE. Locations 0 and 1 may contain 6 and 7, or 6 and 6, or 5 and 7.

NOTE

\A system may also build a VAX-style memory system with masked writes to the main memory. The longword granularity of sharing is being included to allow simpler and cheaper systems to be built. But since some PRISM systems may use a common memory system with a given VAX implementation we are not going to disallow reusing the existing memory subsystems.\

2. When executing an instruction, a PRISM processor always reads and writes certain instruction operands from memory by atomic accesses. These operands, termed atomic operands, are the following: an aligned longword, and an aligned quadword. An atomic operand is the only kind of shared datum for which a PRISM processor inherently provides atomic memory access.

   For example, if an implementation designed a PRISM memory system that only provided longword access, it must guarantee that aligned quadwords are read and written atomically to prevent a quadword datum from being corrupted if one processor is performing a store quadword operation while another processor is simultaneously performing a load quadword operation on the same location.

3. A processor is not required to provide atomic memory access for any other operands. This includes, but is not limited to the following: a byte, a word, any datum that is not naturally aligned (such as an unaligned longword or quadword), the HWPCB saved and restored by the Swap Privileged Context instruction, or the group of elements comprising the data referenced by a vector load, vector store, vector gather or vector scatter operation.

9.2.3 Using Interlocks To Prevent The Corruption Of Shared Data

PRISM processors do not provide atomic memory access to all data. To prevent the corruption of data accessed non-atomically, readers and writers must access them in a mutually exclusive manner -- there must be either only one writer or any number of readers to shared data at any one time. Failure to use mutual exclusion when multiple processors or I/O devices are reading or writing with non-atomic accesses may produce UNPREDICTABLE results.

Five interlock instructions are provided for mutual exclusion methods that test and set a memory-resident control variable, such as a lock or semaphore, through hardware-implemented atomic memory accesses: CMPSWLI, CMPSWQI, CMPSWQIP, RMALI, and RMAQI. A control variable is a bit string in shared memory whose value indicates whether access to the associated shared data is allowed. Software, by using interlock instructions, sets a control variable to any of the various values which indicate whether the shared data is accessible or inaccessible to other processors and I/O devices.

CMPSWLI, CMPSWQI, and CMPSWQIP perform a compare and swap operation on a virtual longword-aligned, virtual quadword-aligned, or physical quadword-aligned control variable respectively. RMALI and RMAQI perform a read, mask and add operation on a longword-aligned or quadword-aligned control variable respectively.

Each of these instructions accesses a control variable using an atomic sequence of operations termed an interlocked sequence. A processor wishing to begin this sequence requests access to the interlock grain, which is an implementation-dependent region of physical memory containing the control variable. When the access has been granted, the interlocked sequence starts and the interlock grain is locked (made inaccessible) to the interlocked sequences of all other processors or I/O devices. When the sequence is completed, the interlock grain is unlocked making it accessible to the interlocked sequences of other processors and I/O devices. Thus, any two interlocked sequences performed on the same interlock grain are always atomic with respect to each other -- that is, they will always interlock. Note that interlocking occurs at the level of physical memory, and only when control variables map to the same interlock grain will actual interlocking be guaranteed. The size of the interlock grain is implementation dependent, therefore to ensure correct operation software must use a common interlock control variable, beginning at the same byte address for all users of the shared data, and not rely on the interlock grain size of a given implementation.

The following apply to interlocks:

1. Non-interlocked instructions are not necessarily blocked from reading or writing a locked memory region; whether or not they are blocked is implementation dependent.

2. Memory hardware within a PRISM system determines the size of an interlock grain. An interlock grain may vary in size from a byte to all of memory and may even be discontiguous -- for example, every 2**16th byte may be locked. The interlock grain need only include the first byte of the control variable. Therefore, memory hardware need only ensure that interlocked sequences interlock when the starting byte addresses of their control variables map to the same byte in physical memory. It is UNPREDICTABLE whether interlocked sequences will interlock if they reference control variables with different starting-byte physical addresses.

   For example, suppose a programmer chooses to use a quadword in the following way: The low longword will be a counter accessed with RMAQI, and the high longword will contain a one-bit lock accessed with RMALI. Then, even though the control variable for the RMAQI is the entire quadword, the programmer must not expect that the RMALI to the upper longword will interlock with the RMAQI to the quadword since only byte 0 of the quadword need be locked for the RMAQI and byte 4 for the RMALI.

3. It is possible in PRISM implementations that interlocked sequences on two different control variables will unintentionally interlock because both control variables lie within the same interlock grain. For example, if the size of the interlock grain for a given PRISM processor

implementation is one page (8192 bytes), then an interlocked sequence performed on a control variable within a page will lock out all other interlocked sequences which are performed on the same page. Software must not rely on such implementation specific interlock sequences.

4. A processor may read more bytes than necessary when reading a control variable for an interlocked sequence, but it only writes back the following: for RMALI and CMPSWLI, the aligned longword that contains the control variable; and for RMAQI, CMPSWQI and CMPSWQIP, the aligned quadword that contains the control variable.

5. Software must not place data that it will write by non-interlocked instructions within a range of bytes that is written back when a control variable is modified. This is because the write may occur while an interlocked sequence is being performed on the control variable. In this case, the data may be overwritten during the sequence when the control variable is modified. Thus, it is UNPREDICTABLE whether a non-interlocked instruction updates any byte contained within a range of bytes that is written when a control variable is modified.

   In particular, software may not change an interlock control variable with a PRISM "store" instruction. For example, if a second processor tested the interlock control variable using a RMAQI or RMALI instruction with an addend of zero and a mask of all ones and the lock owner's store occurred between the RMAxI read and write, the state of the lock would remain unchanged.

6. For performance reasons, software should avoid looping with an interlocked instruction when waiting for access to shared data. Recall that non-interlocked instructions are not necessarily blocked from reading control variables in a locked interlock grain. On most PRISM processors, checking for the availability of shared data is most efficiently done with non-interlocked instructions. If the non-interlocked instruction indicates the shared datum is available, then an interlocked instruction can confirm the result; otherwise, the program can continue looping on the non-interlocked instruction or do something else. Note that another processor can gain control of the data in the time between the non-interlocked and interlocked checks for data availability.

   Testing a lock control variable with a non-interlocked instruction is more efficient in an MP system because several processors may be trying to acquire the lock and once the processors have loaded the control variable into their individual caches they can test the control variable without impacting the memory system by continually locking and unlocking the interlock grain. Once the owner of the lock control variable releases the lock, hardware guarantees that all other processor caches will eventually see the update or invalidate produced by the owner.

The following code example illustrates this:

```
10$:    ornot   #1,r0,r5        ; mask for rmali
        or      #1,r0,r6        ; addend for rmali
        lda     ctlvar,r4       ; get address of control variable
        rmali                   ; read, mask, add longword
        blbc    r4,30$          ; if lbc, we have ownership

20$:    ldl     ctlvar,r4       ; get control variable
        blbs    r4,20$          ; wait for lock to clear
        beq     r0,10$          ; retry the lock

30$:
```

9.3 SEPARATION OF PROCEDURE AND DATA

The PRISM architecture encourages separation of procedure (instructions), read-only data, and writable data. PRISM procedures may NOT write data that is to be subsequently executed as an instruction without an intervening IFLUSH instruction. If no IFLUSH occurs between a procedure writing data and a subsequent attempt to execute that data as instructions, the results are UNPREDICTABLE.

9.4 TRANSLATION BUFFER, VIRTUAL I AND D CACHES

A system may choose to include a Translation Buffer (TB), a Virtual Instruction Cache (Virtual I Cache), or a Virtual Data Cache (Virtual D Cache). The contents of these caches and/or translation buffers may become invalid, depending upon what operating system activity is being performed. The following table shows what needs to be invalidated for given operating system functions.

Table 9-1: TB/Cache Invalidation

OS Function	TB	Virtual I Cache	Virtual D Cache
Remove from Working Set	Invalidate	-	Invalidate
Delete virtual address	Invalidate	Invalidate	Invalidate
Change PTE<READ_PROT>, PTE<FOE>	Invalidate	Invalidate	
Change PTE<PROT>, PTE<FOR>, PTE<FOW>	Invalidate		Invalidate
Change I-Stream (e.g. processor writes)	-	Invalidate	-
I/O writes new I-Stream	-	Invalidate	-

Assumptions on the above table:

- The D Cache watches I/O and processor writes.
- The I Cache does not watch I/O or processor writes.

Note the Translation Buffer Invalidate instructions (TBFLUSH, MTPR TBIS) only operate on a Translation Buffer and Virtual D Cache, while the IFLUSH instructions only operate on the Virtual I Cache.

9.5 CACHES AND WRITE-BUFFERS

A hardware implementation may include mechanisms to reduce memory access time by making local copies of recently used or expected to be used memory contents or by buffering writes to complete at a later time. Caches and write-buffers are examples of these mechanisms. A cache must be implemented in such a way that its existence is transparent to software (except for timing and error reporting/control/recovery and modification to the I-stream). For example, software does not have to worry about flushing caches before or after I/O transfers.

The following requirements must be met by all cache/write-buffer implementations. All processors and I/O peripherals must provide a coherent view of memory consistent with the rules described in the previous sections for PRISM processors, "smart" I/O devices and "simple" I/O devices.

1. Caches that buffer write data must be able to detect a later write from a "simple" I/O device and invalidate or update their write.

2. A processor must guarantee that all of its previous writes are visible to all other processors and "smart" I/O devices before the write of an interlocked read-modify-write is performed. These writes are only guaranteed to be visible to those other processors or "smart" I/O devices after they perform an interlocked operation on the same control-variable for the purpose of aquiring access to the shared data.

3. A processor must guarantee that a data store to a location followed by a data load from the same location must read the updated value.

4. A processor must guarantee that all of its previous writes are visible to all other processors and I/O devices before a HALT instruction completes. A processor must guarantee that its caches are coherent with the rest of the system before continuing from a HALT.

5. A processor must guarantee that across a powerfail/recovery sequence that the memory system remains coherent. Data can not be lost that was written by the processor before the powerfail and the cache must be in a valid state before normal instruction processing is continued after power is restored.

NOTE

The SWPCTX instruction does not flush pending writes. Therefore, the operating system must perform an interlocked operation to a common lock variable after saving the process state to ensure that all of a process's state is visible to all other processors in a multiprocessor system before the process can be continued on a different processor.

There are many different ways to implement caches. Three different ways currently being used at DIGITAL are write-through, write-back, and write-buffers with a write-through cache. Each method has different problems meeting the PRISM requirements for a cache. The notes following each requirement explain what that requirement means to different implementations.

1. Processor writes to memory followed by a peripheral output transfer must output the updated data.

   - Write-through - In a system with a write-through cache the memory is written as soon as any write is done so the cache need not be able to present its data in place of the memory system.

   - Write-back - In a system with a write-back cache the cache must watch the memory bus and have a mechanism for presenting the correct data when an I/O device accesses a

location that it has cached.

- Write-buffer - In a system with a write-buffer the write-buffer must either watch the memory bus and have a mechanism for presenting the correct data when an I/O device accesses a location that it has buffered or it must purge its contents on all interlocked sequences.

2. Completing a peripheral input transfer followed by the program reading of the memory must read the input value.

   - Write-through - In a system with a write-through cache the cache must watch the memory bus and have a mechanism for either updating or invalidating locations that are written by an I/O device or another processor.

   - Write-back - In a system with a write-back cache the cache must watch the memory bus and have a mechanism for either updating or invalidating locations that are written by an I/O device or another processor.

   - Write-buffer - In a system with a write-buffer the write-buffer must either watch the memory bus and have a mechanism for invalidating pending writes when an I/O device writes a location that it has buffered or it must purge its contents on all interlocked sequences.

3. A write followed by a HALT on the same processor, followed by a read on another processor, must read the updated value.

   - Write-through - In a multiprocessor system with a write-through cache the memory is written as soon as any write is done so there are no additional requirements.

   - Write-back - In a multiprocessor system with a write-back cache, the cache must either continue to watch the memory bus for reads and present the correct data when the other processor accesses a location that it has cached or the cache must propagate all dirty locations to memory before completing execution of a HALT.

   - Write-buffer - In a multiprocessor system with write-buffer all buffered writes must be written to memory before completing execution of a HALT.

4. A HALT on one processor, followed by a write on a second processor, followed by a continue on the first processor, followed by a read on the first processor, must read the updated value.

   - Write-through - In a multiprocessor system with a write-through cache, the cache must either continue to watch the memory bus for writes to locations it has cached, or the cache must invalidate all entries before continuing execution from the HALT.

- Write-back - In a multiprocessor system with a write-back cache, the cache must either continue to watch the memory bus for writes to locations it has cached, or the cache must invalidate all entries before continuing execution from the HALT.

- Write-buffer - In a multiprocessor system with write-buffer all buffered writes must be written to memory before completing execution of a HALT.

5. A write followed by a power failure, followed by restoration of power, followed by a read, must read the updated value provided that the duration of the power failure does not exceed the maximum non-volatile period of the main memory.

   - Write-through - In a system with a write-through cache the cache power supply must be backed up or the cache must be invalidated on restoration of power.

   - Write-back - In a system with a write-back cache either the cache power supply must be backed up or the cache must be written back to main memory on powerfail and the cache invalidated on restoration of power.

   - Write-buffer - In a system with a write-buffer either the write-buffer power supply must be backed up or the write-buffer must be written back to main memory on powerfail and the write-buffer initialized to empty on restoration of power.

   NOTE

   An implementation may choose not to provide powerfail recovery.

6. In multiprocessor systems, access to memory data shared between processors must be interlocked by software executing one of the interlocked instructions on a common control variable. A cache or write-buffer must ensure that all previous writes from the issuing processor are visible to all users of the memory system before the interlocked sequence completes. Alternatively, a cache or write buffer must make all previous writes visible to the memory system and the memory system must ensure that these writes are made visible to all users before they acquire ownership of the shared data by executing an interlocked instruction on the common control variable. The order of writes to memory may be different from the original order of writes by the processor.

   - Write-through - In a system with a write-through cache the memory is written as soon as any write is done and other caches that may have a stale copy of the data must either remain coherent with the memory or become coherent as part of the interlocked operation.

   - Write-back - In a system with a write-back cache it must either remain coherent with all the other caches or

become coherent as part of the interlocked operation.

- Write-buffer - In a system with a write-buffer the write-buffer must purge all its pending writes before the interlocked operation completes.

NOTE

\In a multiprocessor system with caches, the interlocked instructions must cause the data being accessed to be coherent across all processors sharing it. This implies some form of global locking at some granularity. The simplest could be a single global lock that is required to perform any interlocked operation. For performance reasons an implementor may choose to have more locks that interlock access to a subset of all memory. \

A control variable must be acquired with an interlocked instruction before entering a critical section that accesses shared data. A control variable may be tested with either interlocked or non-interlocked instructions. However, a control variable must be acquired and released with an interlocked instruction only.

7. Access to I/O space must not be cached or buffered. Interlocked access to I/O space addresses gives UNPREDICTABLE results.

8. A cache may prefetch instructions or data. A memory management exception condition cannot be taken until the prefetched data is referenced.

NOTE

\If the granularity of access to memory is larger than the request and there is a hardware error (e.g. uncorrected read error, bus parity error, etc.) in part of the requested data (but not the part being accessed), it is valid to report the error as including the valid part. \

9. Processor initialization must leave the cache and/or write-buffer either empty or valid.

9.6 STACKS

To provide support for exception handling, and emulation of missing instructions on subset implementations, the PRISM architecture reserves the right to modify the next 256 quadwords (2048 bytes) of the stack, given normal access checks allow access. These are the

bytes in the range from -1(SP)..-2048(SP). Programs should not store data in this area.

9.7 SYNCHRONIZATION BETWEEN VECTOR AND SCALAR MEMORY ACCESSES

A PRISM processor may contain both a vector unit and a scalar unit each of which can independently access memory. Since the vector unit may access memory concurrently with the scalar unit, the architecture provides methods to synchronize the two units. Software is responsible for determining when read/write memory data conflicts between scalar and vector references might produce incorrect results, and inserting memory synchronization instructions to ensure correct operation.

9.7.1 Synchronization Instructions

DRAINM should be used for synchronizing scalar and vector memory references. DRAINM stalls all instructions from issuing until all previously issued scalar and vector load/store instructions are guaranteed to complete without encountering memory management exceptions. It also stops the issuing of subsequent scalar and vector load/store instructions until all outstanding memory accesses have been completed.

The architecture also allows for the implementation of multiple vector load/store paths to memory. Using DRAINM for scalar/vector memory synchronization ensures correct operation, but it does more than is needed. If it is known that memory access synchronization is needed only between multiple vector load/store instructions, DRAINV is sufficient. If an implementation provides multiple vector load/store paths to memory, DRAINV stalls all instructions from issuing until all previously issued vector load/store instructions are guaranteed to complete without encountering memory management exceptions. It also stalls the issuing of subsequent vector load/store instructions until all pending vector load/store instructions with possible conflicting memory accesses complete. DRAINV does not have any synchronizing effect between scalar and vector load/store instructions. \ DRAINV can be a no-op instruction in an implementation that has a single vector load/store path to memory. \

9.7.2 Required Use Of Memory Synchronization Instructions

Table 9-2 shows for all possible pairs of vector or scalar reads and writes to a common memory location, whether the DRAINM instruction or the DRAINV instruction must be issued after the first reference and before the second. Since DRAINM also includes the DRAINV function, DRAINM can always be used instead of DRAINV.

Table 9-2: When DRAINM (M) Or DRAINV (V)

Is Required Between Instructions That Reference The Same Memory Location

First Reference / Second Reference	Scalar Scalar	Scalar Vector	Vector Scalar	Vector Vector
Operation Sequence				
Read Read	No (1,2)	No (1)	No (1)	No (1)
Read Write	No (2)	M (3)	M	V (5)
Write Read	No (2)	M (4)	M	V
Write Write	No (2)	M (4)	M	V

Key:
1 - DRAINM or DRAINV is never required between two read accesses to a memory location.
2 - DRAINM is never required between two accesses by the scalar unit to a memory location.
3 - DRAINM ensures that the read completes before the vector store operation is issued.
4 - DRAINM is required to ensure that the write is made visible to the vector unit before the vector memory reference is issued.
5 - The rules for the required use of DRAINV are described elsewhere in this section.

In general, these rules apply to any sequence of instructions that access a common memory location, no matter how many other vector or scalar instructions are issued between the first instruction that accesses the common location and the second instruction that accesses the same location. For example, the code sequence shown below depicts a vector load followed by a scalar write to the same memory location. Between these two instructions are other scalar/vector instructions that do not access the common memory location. A DRAINM instruction must be executed sometime after the vector read and before the scalar write to the common location.

```
VLDL   #4, R6, V0
.
other scalar/vector instructions
that do not access data loaded by VLDL
.
DRAINM
STL    R0, 4(R6)
```

In the following examples, however, data dependencies between two vector memory access instructions to common memory ensure correct order of memory access without DRAINV:

```
example 1:  VLDL    #4, R6, V0
            VSTL    #4, R6, V0
```

```
example 2:  VLDL    #4, R6, V0
            VSUB    R0, V0, V1
            VSTL    #4, R6, V1

example 3:  VLDQ    #8, R6, V0
            VLDQ    #8, R7, V1
            VADDG   R8, V0, V2
            VVMULG  V2, V1, V3
            VSTQ    #8, R6, V3

example 4:  VLDL      #4, R6, V0
            VCMPFGT   R0, V0            ;writes VMR
            VLDL/1    #4, R7, V1
            VLDL/0    #4, R8, V2
            VMERGE/0  V2, V1, V3
            VSTL      #4, R6, V3
```

Thus when both of the following conditions are satisfied, DRAINV is not required:

1. There is a read/write pair of vector memory access instructions that touch identical memory in the same order. (For VLD/VST instructions the base and stride operands must be identical and for VGATH/VSCAT instructions the base and 32-bit offsets must be identical.)

2. There is a data dependency between this read/write pair. If the operation involving that dependency is somehow recognized as not required, the information can not be used by hardware to optimize (and perhaps no-op) the operation. Several examples follow:

```
VLDL    #4, R6, V0
VMULL   R0, V0, V1
VSTL    #4, R6, V1
```

   If the vector unit notices that the scalar multiplier is R0 (zero), it could just clear V1 and start storing V1 before the loading of V0 finishes. In that case, V0 would not necessarily contain the correct data at the end of the sequence.

```
VGATHL    R6, V2, V0
VMERGE/0  V0, V1, V1  and VMR is all ones
VSCATL    R6, V2, V1
```

   If the vector unit notices that the VMERGE operation is merely copying V1 to itself, it could skip the VMERGE and start storing V1 before the loading of V0 finishes. Depending on the values of V2 elements, V0 may contain incorrect data at the end of the sequence.

Since both of the above conditions are not satisfied in the following code sequence, a DRAINV instruction is required after the vector read and before the vector write to the common memory location:

```
VLDL/1  #4, R6, V0
VCMPLT  #0, V1
DRAINV
VSTL/1  #4, R6, V1
```

If the DRAINV is not included, V0 could contain incorrect data at the end of the sequence. Without DRAINV, there is no guarantee that the storing of V1 will start after the loading of V0 is finished.

Only DRAINM guarantees that the memory operations of the vector and scalar units are synchronized. Writes to I/O space, changes in access mode, machine checks, interprocessor interrupts, execution of a HALT, REI, or interlocked instruction do not make the results of vector instructions that write to memory visible to the scalar unit, I/O subsystem, or other processors. Execution of a DRAINM instruction must precede any of these mechanisms to ensure synchronization of all system components.

DRAINM is required:

- After a vector instruction that stores to memory and before a peripheral (I/O) data transfer of the stored location is initiated to ensure that the value stored will be transferred to the output device.
- After a vector instruction that stores to memory and before an interlocked instruction that releases ownership of the shared memory data to another processor.
- After an interlocked instruction that acquires ownership of shared memory data from another processor and before a vector instruction is initiated that that reads the shared data.
- After a vector instruction that stores to memory and before the associated scalar unit can execute a HALT. This ensures that a read or modify by another processor will access the updated memory value.
- Before the vector unit state is saved as a result of power failure. A read or modify of the same memory must read the updated value (provided that the duration of the power failure does not exceed the maximum non-volatile period of the main memory).
- Before a context switch. Software is responsible for ensuring that the vector unit has completed all its memory accesses before performing a context switch.

NOTE

Note that the rules described in this section only deal with data coherency between the scalar and vector units within the same processor. In a multiprocessor system, the interprocessor signaling rules described in 9.2.2.1 apply. To guarantee that vector writes are made visible to other processors in a system, software must execute a DRAINM (to make vector writes visible to the scalar unit in the same processor) followed by an interlocked instruction (to make all writes visible to other processors).

Revision History:

Revision 3.0, 26 April 1988

1. Change granularity of sharing to longword.
2. Clarify rules for sharing data.
3. Add rules for scalar/vector synchronization.

Revision 2.0, 24 June 1986

1. Remove distinction between IPROT and DPROT in table 9-1.
2. Remove reference to TBIASN IPR.
3. Corrected range of access allowed beyond end of stack.

Revision 1.0, December 22, 1985

1. General rewrite to reflect change from byte granularity of access for independent modification to quadword or less granularity of access for independent modification.
2. Expanded Translation Buffer invalidation rules.
3. Expanded cache rules to cover write-buffers.
4. Corrected range of access allowed beyond end of stack.

Revision 0.0, July 5, 1985

1. First Review Distribution

CHAPTER 10

EXTENDED PROCESSOR INSTRUCTION CODE

10.1 INTRODUCTION

In a family of machines both users and operating system implementors require functions to be implemented consistently. When functions conform to a common interface, the code that uses those functions can be used on several different implementations without modification.

These functions range from the binary encoding of the instruction and data, to the exception mechanisms and synchronization primitives. Some of these functions can be cost effectively implemented in hardware, but others are impractical to implement directly in hardware. These functions include low-level hardware support functions such as Translation Buffer miss fill routines, interrupt acknowledge, and vector dispatch. It also includes support for privileged and atomic operations that require long instruction sequences such as Return from Exception or Interrupt (REI).

In the VAX, these functions are generally provided by microcode. This is not seen as a problem because the VAX architecture lends itself to a microcoded implementation.

One of the PRISM goals is that microcode will not be necessary for practical implementation. However it is still desirable to provide an architected interface to these functions that will be consistent across the entire family of machines. The Extended Processor Instruction code (Epicode) provides a mechanism to implement these functions without resorting to a microcoded machine. Hardware development groups provide and maintain the Epicode for a given implementation.

NOTE

> \The hardware development groups provide and maintain the Epicode for a given implementation. The Epicode may be in ROM or loaded into RAM from some sort of a console load device. Many of the same trade-offs exist for Epicode that exist for VAX microcode around patching, loading, and booting.\

10.2 EPICODE ENVIRONMENT

Epicode runs in an environment with privileges enabled, and I-stream

mapping and interrupts disabled. The enabling of privileges allows all functions of the machine to be controlled. Disabling of I-stream mapping allows Epicode to be used to support the memory management functions (e.g. Translation Buffer miss routines cannot be run via mapped memory). Epicode also needs to make both virtual and physical D-stream references. Disabling interrupts allows the system to provide multi-instruction sequences as atomic operations (e.g. RMAQI).

The PRISM architecture allows these functions to be implemented in standard machine code resident in main memory. Epicode is written in standard machine code with some implementation specific extensions to provide access to the "real hardware." Epicode can be used to implement the following functions:

- Instructions that require complex sequencing as an atomic operation (e.g. REI)
- Instructions that require interlocked memory access (e.g. RMAQI)
- Privileged instructions (e.g. MxPR)
- Memory management control functions (e.g. TB miss routines, ACV/TNV dispatch routines)
- Interrupt and exception dispatch routines
- Power-up initialization and booting
- Console functions
- Emulation of instructions with no hardware support (e.g. an implementation may chose to do MULL via a multiply step function in the integer ALU)
- Support for unaligned memory operands

A PRISM implementation can make various design trade-offs based on the hardware technology being used to implement the machine. The Epicode will then be used to hide these differences from the system software.

For example, in a MOS VLSI implementation, a small (16 entry) fully associative TB may be the right match to the media given that chip area is a costly resource. In an ECL version, a large (1024 entry) direct-mapped TB may be used because it will use RAM chips and does not have fast associative memories available. This difference would be handled by implementation-specific versions of the epicode on the two systems, both providing transparent TB miss service routines. The operating system code would not need to know there were any differences.

10.3 EPICODE EFFECTS ON SYSTEM CODE

Epicode will have one minor effect on system code. Because Epicode may be resident in main memory and maintain privileged data structures in main memory, the operating system code that allocates physical

memory cannot use all of physical memory. The amount of memory Epicode will require will be small, so the loss to the system is negligible.

10.4 SPECIAL FUNCTIONS REQUIRED FOR EPICODE

Epicode uses the PRISM instruction set for most of its operations. There are a small number of additional functions needed to implement the Epicode. There are five opcodes reserved to implement Epicode functions (i.e. EPIRES0, EPIRES1, EPIRES2, EPIRES3 and EPIRES4). These instructions produce a Reserved Opcode fault if executed while not in the Epicode environment.

- Epicode needs a hardware mechanism to transition the machine from the Epicode environment to the non-Epicode environment. This instruction loads the PC, enables interrupts, enables mapping, and disables Epicode privileges in a single instruction.

- Epicode needs a set of instructions to access the hardware control registers (i.e. a hardware MxPR).

- Epicode needs a mechanism to save the current state of the machine and dispatch into Epicode.

A PRISM implementation may also choose to provide additional functions to simplify or improve performance of some Epicode functions. The following are some examples:

- A PRISM implementation may include a READ/WRITE virtual function that allows Epicode to perform mapped memory accesses using the mapping hardware rather than providing the virtual-to-physical translation in Epicode routines. Epicode may provide a special function to do PHYSICAL READs/WRITEs and have the PRISM LOADs/STOREs continue to operate on virtual address in the Epicode environment.

- A PRISM implementation may include hardware assists for various functions, for example, saving the virtual address of a reference on a memory management error rather than having to generate it by simulating the effective address calculation in Epicode.

- A PRISM implementation may include private registers so it can function without having to save and restore the native general registers.

Revision History:

Revision 3.0, 26 April 1988

1. Minor edits.

Revision 2.0, 24 June 1986

1. Minor edits

Revision 1.0, December 22, 1985

1. General edits to make it clear that Epicode can be done in any way that works well for a given implementation.

.

Revision 0.0, July 5, 1985

1. First Review Distribution

CHAPTER 11

SYSTEM BOOTSTRAPPING AND CONSOLE

This chapter describes system bootstrapping and required console functionality.

NOTE

/This chapter is not yet complete and will evolve as the hardware and software design progresses. A common console architecture for uniprocessor and multiprocessor systems is currently being defined and is likely to have a major impact on this chapter. /

11.1 BOOTSTRAPPING

This section describes PRISM bootstrapping. Topics covered include responsibilities of the console, the initial state seen by system software, and powerfail recovery. Bootstrapping is discussed in both a multiprocessor and uniprocessor environment.

Many of the actions described below are the responsibility of the console. This does not imply that a separate console processor is required. Rather, it is expected that console functionality will be implemented in Epicode running in the PRISM processor. Thus, anywhere the console is referred to in this chapter, it is meant that the function must be provided, not that a console processor exists.

11.1.1 Bootstrapping In A Uniprocessor Environment

This section describes a cold start in a uniprocessor environment. Powerfail recovery and multiprocessor bootstrapping are described in Sections 11.1.3 and 11.1.4.

The following steps occur in the bootstrap sequence.

1. Test memory for bootstrapping
2. Build the Restart Parameter Block (RPB)

3. Load Epicode
4. Initialize the page table
5. Load system software
6. Initialize processor IPRs
7. Transfer control to system software

Note that these steps may be performed in a different order on different implementations of the PRISM architecture. The final state seen by system software is defined, but the implementation-dependent procedure is not.

11.1.1.1 Memory Testing

In general, memory sizing and testing is the responsibility of system software. The exception to this is the memory needed to set up the initial environment for system software as described below. This includes the memory for Epicode, the RPB, page tables, and system software. It is the responsibility of the console to find the lowest addressable good memory for these purposes.

11.1.1.2 Restart Parameter Block

The Restart Parameter Block is the primary mechanism for passing data between the console and system software. It is also critical in powerfail recovery. The console is responsible for setting up a page aligned RPB in the first good memory that can be found. UNDEFINED operation will result if the RPB memory is reused by system software for any other purpose.

An area is reserved in the RPB for each processor. The per-processor areas immediately follow the main portion of the RPB in the same page and any necessary contiguous pages. Each per-processor area must be quadword aligned. A field in the RPB specifies the number of processor slots.

A state longword for each processor is included in the per-processor area. It contains several flags used to either control bootstrapping or record progress. This longword can only be modified with interlocked instructions to guarantee proper synchronization in multiprocessor systems.

The RPB, including all per-processor areas, is initialized at this time. Other than the fields listed below, the initialization value is **zero**:

- Physical address of RPB
- Version number

- Number of processor slots
- Physical address of per-processor area
- Physical address of checksum area
- Checksum
- Page size
- ASN size
- Number of physical address bits

A checksum area must be created for use during powerfail. This area exists only to help guarantee that a valid RPB can be located. This area can be anywhere that is accessible to all processors, including at the end of the RPB. It can contain any data that does not change. (Zero data is not recommended because it increases the probability of locating a spurious RPB.)

Note that the RPB does not contain a save area for vector registers. Instead, there is only a pointer to this area. It is the responsibility of system software to allocate a page aligned 8-Kbyte vector register save area for each processor.

The length of the RPB can be calculated by software based on the version number and the number of slots.

The number of per-processor RPB areas is at least equal to the number of the highest numbered existing processor plus one. (processor numbers are zero-based) Areas corresponding to nonexisting processors are zeroed. An implementation may choose to create more per-processor areas than are necessary.

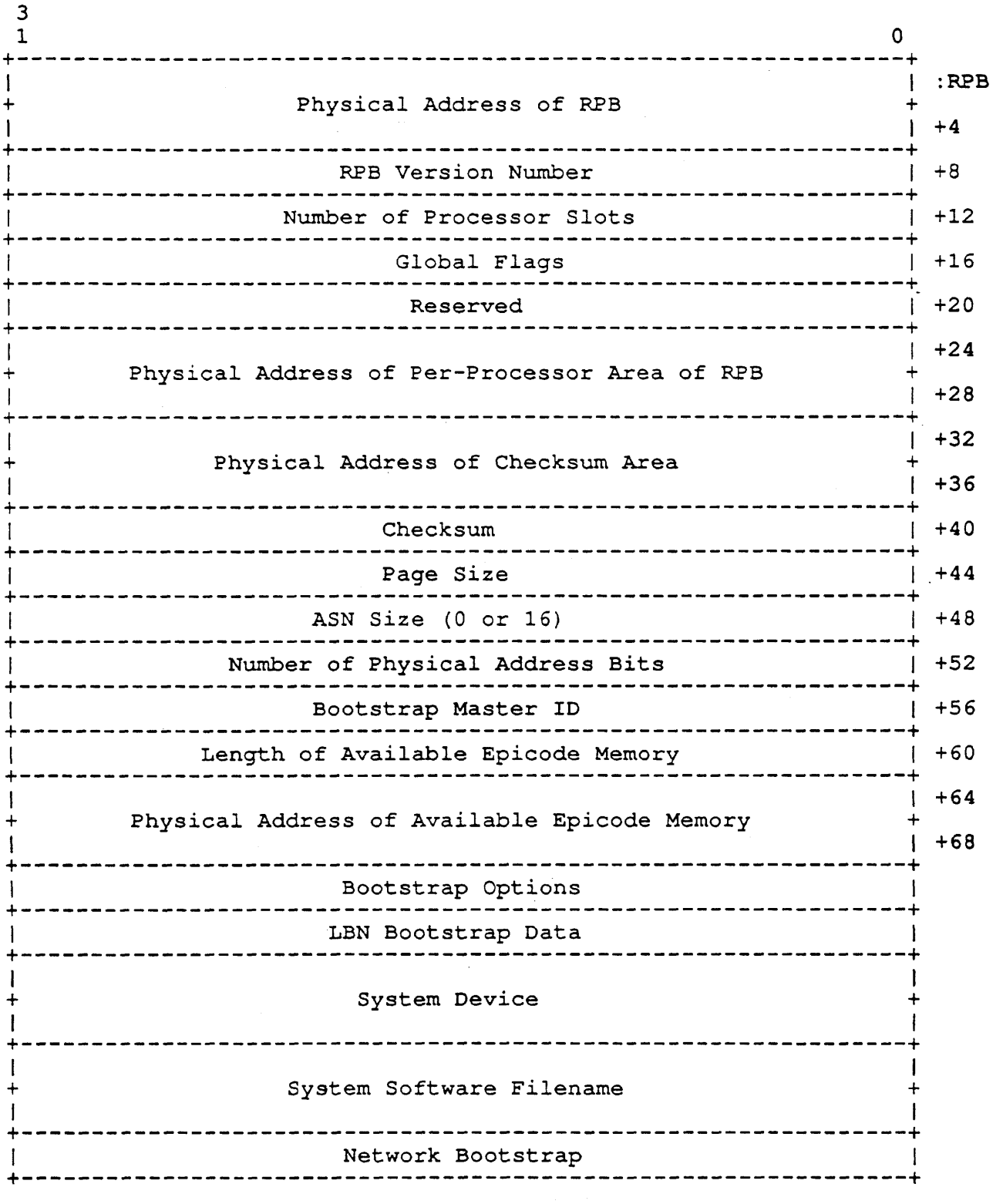


Figure 11-1: Restart Parameter Block

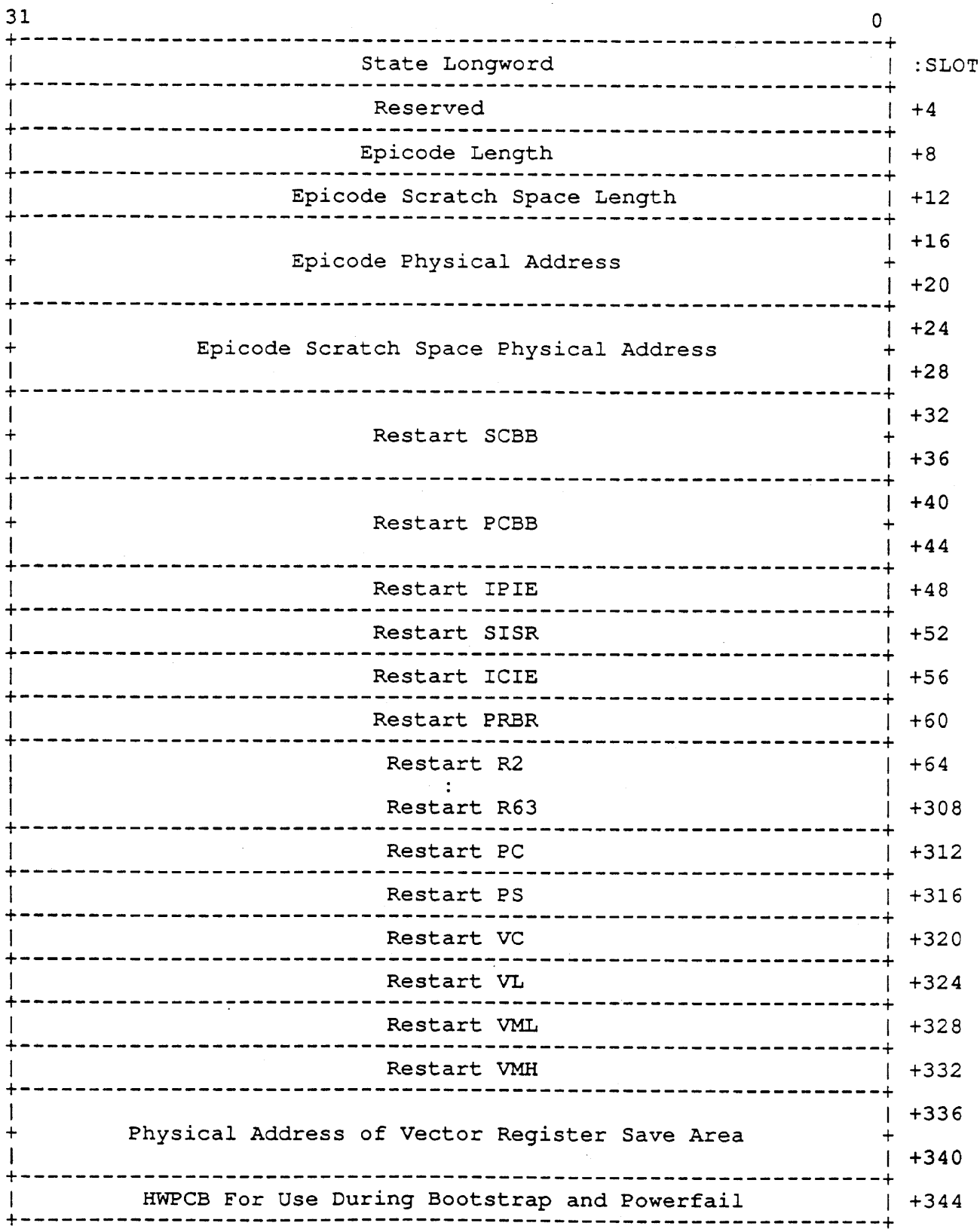


Figure 11-2: Per-Processor Portion of RPB

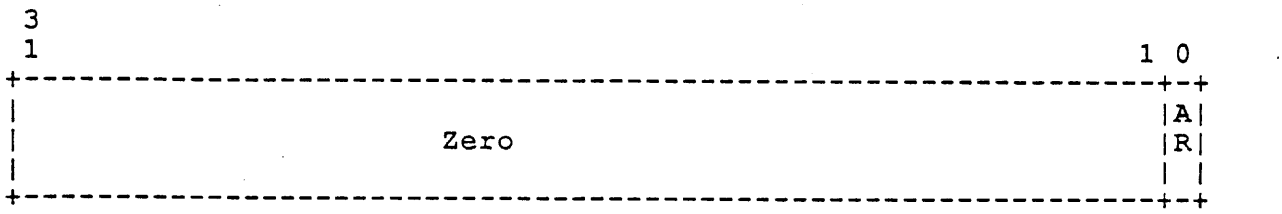


Figure 11-3: Global Flags

Fields in the global flags longword are interpreted as follows:

Bits	Description
1	Auto Reboot (AR) - This bit is set if an autoreboot is in progress. This bit is set by epicode and remains set until the next reboot.

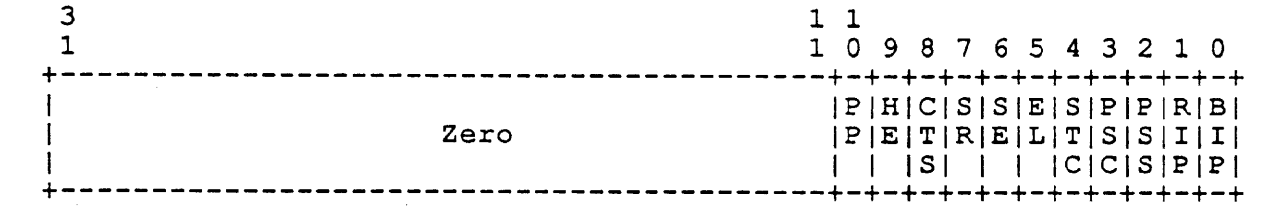


Figure 11-4: State Longword

Fields in the state longword are interpreted as follows:

Bits	Description
0	Bootstrap in Progress (BIP) - The processor is currently bootstrapping. This bit is set by Epicode and cleared by system software.
1	Restart in Progress (RIP) - The processor is currently restarting after powerfail. This bit is set by Epicode and cleared by system software.
2	Powerfail Sequence Started (PSS) - Epicode has entered powerfail processing. This bit is set and cleared by Epicode.
3	Powerfail Sequence Completed (PSC) - Epicode has completed powerfail processing. This bit is set and cleared by Epicode.
4	Self-Test Complete (STC) - Any self-test functions have been completed during bootstrapping or powerfail restart. This bit is set by Epicode.
5	Epicode Loaded (EL) - Epicode loading is complete. This bit is set by Epicode.

6 Software Enabled (SE) - This bit is set and cleared by system software and enables this processor for multiprocessor system operation. (Note that the boot master is always enabled.)

7 Slave Request (SR) - A slave processor is ready to bootstrap in a multiprocessor system. This bit is set by slave processor Epicode and cleared by system software.

8 Control Transferred to System Software (CTS) - Epicode has transferred control to system software during bootstrapping. This bit is set by Epicode.

9 Hardware Enabled (HE) - This bit indicates that the CPU is available for use by PRISM software. It is set by the console and may differ from the processor present based on self-test, other diagnostics, or console commands which explicitly make a CPU unavailable.

10 Processor Present (PP)- This bit indicates that the CPU is physically present in the configuration. It is set by the console.

11.1.1.3 Epicode Loading

Epicode may be loaded into the next available good memory and its address and length are recorded in the per-processor slot of the RPB. The Epicode is always page aligned. The Epicode source and its loading mechanism is implementation-specific. The source may be a special console device, a system device, or any other implementation-specific source.

If control must be transferred to Epicode at this point, it is done in an implementation-specific manner.

Certain assumptions are made about the state of the system when Epicode is to be loaded or is to gain control if it is in ROM. First, it must be possible to access a bootstrap device. This may be ROM, mass storage, or a communication line. This is necessary to load either Epicode, controller microcode, or system software. Note that this does not have to be the device which contains the system software. Another device, perhaps one dedicated to console functions, may contain the necessary Epicode and controller microcode. Second, the I/O devices need not contain microcode to support their full functionality. They need only be capable of the primitive operations necessary to read the full microcode from disk.

11.1.1.4 Initial Page Tables

All system software runs in a virtual memory environment. Thus, it is the responsibility of the console to set up initial page tables. These are located in the next available good memory. These page tables map four regions of virtual memory:

1. The page tables themselves

2. The Restart Parameter Block (RPB)

3. The I/O registers for the boot device

4. 256 Kbytes of good memory for use by system software

The virtual memory is laid out in the 32-bit virtual address space as shown below:

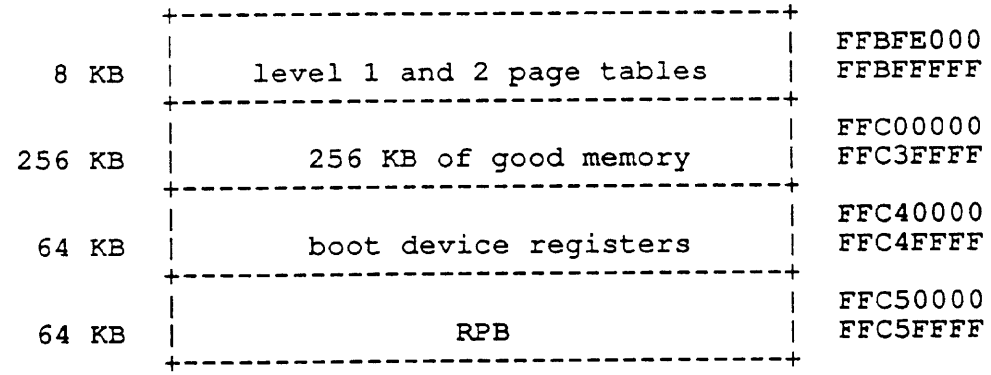


Figure 11-5: Initial Virtual Memory Layout

The initial virtual memory layout was selected so that a single page of memory serves as both a level 1 and level 2 page table. This page is mapped as the last level 1 page table entry and points to itself. Thus it appears at address FFBFE000. The unused second half of the level 1 page table page is then available for use in the level 2 page table and maps the rest of the initial address space.

All pages have Kernel read/write, user no access protection with all fault bits (FOR, FOW, FOE) clear.

11.1.1.5 Bootstrap Flags

The console sets the Bootstrap-in-Progress (BIP) flag in the RPB state longword whenever a cold (not powerfail recovery) bootstrap is done. System software is responsible for clearing the flag at the appropriate time. This should be done after system software is capable of handling powerfail recovery.

\The Bootstrap-in-Progress (BIP) and Restart-in-Progress (RIP) flags exist only in the RPB. They do not exist in an IPR as is the case in a VAX. The RPB is sufficient since it is accessible to both the console and the system software.\

11.1.1.6 Loading Of System Software

The console is responsible for loading system software into the 256 Kbytes of good memory. This software is expected to be a bootstrap which is responsible for loading other system software. However, it may be diagnostics or other special purpose software, see Section 11.3 below.

11.1.1.7 IPR Initialization

Before control is transferred to system software, certain IPRs must be initialized as shown in the following table:

Table 11-1: IPR Initialization

Mnemonic	Register Name	Initialized State
ASN	Address Space Number	zero
ASTEN	AST Enable	disabled
ASTSR	AST Summary	zero
CRCS	Console Receive Status	disabled
CTCS	Console Transmit Status	disabled
ICIE	Interval Clock Int Enable	disabled
IPIE	Interprocessor Int Enable	disabled
PCBB	Privileged Context Block	RPB HWPCB
PTBR	Page Table Base Register	bootstrap page table PFN
SISR	Software Interrupt Summary	zero

The contents of all other IPRs are UNPREDICTABLE.

11.1.1.8 Transfer Of Control To System Software

At this point there is a conceptual change from console control to normal Epicode since the PRISM system is now running in its normal mode rather than bootstrapping. There may or may not be an actual change of control. Depending on implementation details of a PRISM processor, normal Epicode may have gained control at any point before this.

When the console has completed the actions described above, control is transferred to system software in Kernel mode at IPL 7 with virtual memory management enabled. The Hardware Privileged Context Block (HWPCB) in the RPB is already initialized and is active. System software is loaded into the lowest portion of the 256-Kbyte region reserved for this purpose and control is transferred to its first byte. All pages have Kernel read/write, user no access protection with all fault bits (FOR, FOW, FOE) clear.

The Kernel stack pointer (SP) is initialized to point to the top of the 256-Kbyte region of good memory (address FFC40000). All other scalar and vector register contents are undefined.

All bootstrap information is passed from the console to system software in the RPB. This includes:

- System device name
- System software file name
- Bootstrap options
- Logical Block Number (LBN) bootstrap data if appropriate
- Network bootstrap data if appropriate

The rest of the bootstrap process is the responsibility of system software.

11.1.2 Powerfail

When powerfail is detected, control is transferred to Epicode in an implementation-specific manner. If the Restart-in-Progress (RIP) or Bootstrap-in-Progress (BIP) flag is set in the RPB per-processor state longword, no powerfail processing is possible and Epicode takes no action. Otherwise, Epicode sets the Powerfail Sequence Started (PSS) flag in the per-processor state longword in the RPB and then saves all volatile processor state in a combination of the per-processor portion of the RPB and Epicode private storage. Vector registers are saved if system software has allocated a save area and recorded its address in the RPB even if the Vector Enable bit is not set in the Processor Status (PS<VEN>). System software does not have the opportunity to take any action until powerfail recovery. After Epicode completes all powerfail processing, the Powerfail Sequence Complete (PSC) flag in the per-processor state longword in the RPB is set.

11.1.3 Powerfail Recovery

Powerfail recovery occurs if memory is preserved by battery backup during an interruption of power to the processor and the halt action setting is restart. After determining that memory was backed up and the halt action setting is restart, the console locates the RPB and examines the per-processor RPB state longword flags to determine that powerfail was completed (PSC set) and that restart or bootstrapping was not in progress (BIP and RIP clear). If these conditions are not met, the processor either halts or starts a cold bootstrap.

The RPB is found by a search of memory looking for the distinctive signature of the RPB as described below. If the search fails, the processor either halts or starts a cold bootstrap.

1. Search for a page of memory that contains its physical address in the first two longwords. If none is found, the search for an RPB has failed.
2. Get the physical address of the checksum area from the potential RPB. If it is not a valid physical address, or if it is zero, return to Step 1. The check for zero is necessary to ensure that a page of zeros does not pass the test for a valid RPB.

3. Calculate the 32-bit twos complement sum (ignoring overflows) of the 31 longwords in the checksum area. If the sum does not match the checksum in the potential RPB, return to Step 1.

4. A valid RPB has been found.

If all tests pass, the console transfers control to the Epicode restart routine in an implementation-specific manner. Epicode properly restores internal processor registers and the contents of the HWPCB. After setting the Restart-in-Progress (RIP) flag and clearing the Powerfail Sequence Started (PSS) and Completed (PSC) flags in the per-processor state longword, Epicode initiates a Powerfail Recovery interrupt to transfer control to system software. When the Powerfail Recovery interrupt is initiated, PC and PS (saved in the RPB) are pushed onto the Kernel stack. System software is responsible for restoring all other scalar and vector registers. Note that no Epicode or system software is loaded during a restart.

11.1.4 Multiprocessor Bootstrapping

Multiprocessor bootstrapping differs from uniprocessor bootstrapping primarily in areas relating to synchronization between processors. Obviously, in a shared memory system, processors cannot independently load and start system software.

11.1.4.1 Initial Synchronization

In a multiprocessor system, the console must be capable of some primitive operations before Epicode is loaded into memory. These are necessary to synchronize with other processors in the system as described below.

Before continuing the bootstrap process a master processor must be chosen to control bootstrapping. This can be done in any fashion that guarantees choosing exactly one master.

To provide one example of choosing a master, the presence of a register which can be accessed with interlocked instructions is assumed. Note that this is only an example; any workable mechanism, including a predefined master, can be used. An interlocked sequence must be done to see if the interlocked register is clear. If the register is clear, it is loaded with a flag (1) to indicate that a processor is in control of bootstrapping. If the register is already set, there must be a mechanism to loop waiting for an interprocessor interrupt. This can be Epicode in ROM or any other implementation-specific mechanism.

11.1.4.2 Actions Of Bootstrap Master

The first processor to gain control is referred to as the bootstrap master. (In the example, this was the first processor to gain the interlock.) It is the responsibility of this processor to control

bootstrapping and allow all other processors to proceed only at the appropriate time. The bootstrap master allocates an RPB and writes its ID into the RPB. It then proceeds with the normal uniprocessor bootstrap. When bootstrapping is complete, system software sets Software Enabled (SE) flags in the RPB per-processor state longwords to indicate which other processors are enabled. At this time, it requests interprocessor interrupts to these processors.

Note that processor IDs are determined in an implementation specific manner. The only requirement is that they are small, unique numbers. If the numbers are not small, excessive memory may be required for unused per-processor RPB slots.

11.1.4.3 Actions Of Bootstrap Slaves

Bootstrapping processors other than the bootstrap master are referred. to as bootstrap slaves. After failing to become master, a slave remains in console mode and polls for interprocessor interrupts. When an interprocessor interrupt is received, the bootstrap slave must locate the RPB and then check its state longword to ensure that it is enabled. If Epicode memory is required, the slave loads the Epicode length field in the RPB slot. If Epicode scratch space is required, the slave loads the Epicode scratch space length field in the RPB slot. Regardless of the need for memory, the slave then sets the Slave Request (SR) bit in its state word and initiates an interrupt to the bootstrap master. The slave now waits for an interrupt to indicate that memory has been allocated and the address returned in the RPB. Epicode is then loaded by the slave (possibly different Epicode than that loaded by the master). If no memory was required, the slave simply continues with the bootstrap process at this point. The master clears the Slave Request bit before initiating the second interrupt to the slave.

All processors should be prepared to load Epicode on any 8-Kbyte boundary. This is to allow packing of Epicode in large pages in the future. An RPB cell is used to keep track of available memory.

Note that system software in the bootstrap master is responsible for allocating the Epicode memory for the slaves. The master should wait a "reasonable" period of time for a memory request from each slave. Slaves that do not respond are disabled. Explicit operator action is then required to enable additional slaves at a later time. (This is described in the next section.)

Once Epicode is loaded and control transferred to Epicode, the proper environment must be established for system software. This is done by loading the powerfail restart IPRs and registers from the per-processor portion of the RPB and then transferring control to the address specified in the PC field of the RPB. System software in the master is responsible for initializing the RPB fields containing the IPRs and registers.

11.1.4.4 Addition Of A Processor To A Running System

Once bootstrapping is complete, system software is no longer expecting requests for Epicode memory from bootstrapping processors. Thus, the

RPB is not examined when interprocessor interrupts are received. In order to add a new processor, system software must provide an operator function to request that the bootstrap sequence be completed for any new processor.

Note that this section only suggests a mechanism for adding processors to running systems. System software may be designed to use alternative methods.

11.1.5 Powerfail In A Multiprocessing System

Powerfail processing is identical in multiprocessor and uniprocessor systems. Epicode saves state without any communication with other processors.

Powerfail recovery proceeds almost exactly as in a uniprocessor system. Epicode determines if powerfail was not successfully completed (PSC clear) or if restart or bootstrapping was in progress (RIP or BIP set). If so, further checks are done as described below. In the normal case, Epicode restores state and initiates a powerfail recovery interrupt just as in a uniprocessor system. It is the responsibility of system software to coordinate recovery in a multiprocessor system. The multiprocessor system software has the context to determine if it is necessary to wait for some other processor or if this processor should be rebooted. It is responsible for all further powerfail recovery synchronization.

If a processor cannot complete normal powerfail recovery, further checks are needed to distinguish between cases where a cold bootstrap must be initiated and those where the processor must enter slave mode waiting for an interrupt from another processor. The processor must examine all per-processor RPB slots looking for a processor which is either running (PSS, PSC, RIP, and BIP clear, and HE and SE set) or has successfully completed powerfail processing (PSC set). If one is found, the processor enters slave mode and waits for an interrupt from the running or powerfailed processor. Note that this is exactly the state a slave enters after failing to become a master on cold bootstrap. If no processors are running or have successfully completed powerfail, a cold bootstrap is initiated. This procedure is necessary to guarantee that a processor which failed to complete powerfail processing cannot interfere with powerfail recovery of the rest of the system by becoming a master and performing a cold bootstrap. Very unlikely cases do exist where all processors can hang. In particular, if the master/slave interlock is not cleared, it may be impossible to select the new master. However, this is considered more acceptable than an unsynchronized bootstrap.

This procedure is independent of whether or not all processors powerfailed.

11.2 CONSOLE

This section describes the PRISM console functionality. Implementation-specific considerations such as diagnostic functions are not discussed.

A console terminal is connected to each PRISM processor. More information on communication with console terminals can be found in Chapter 8, Internal Processor Registers.

11.2.1 Required Functionality

All PRISM systems must provide console functionality to perform all of the functions described as console responsibility in the bootstrapping portion of this chapter. These include testing part of memory, loading Epicode, setting up a system software environment, loading system software, and handling powerfail recovery. Note that all of these functions are expected to be done with special Epicode executed in the PRISM processor.

11.2.2 Entering Console Mode

The PRISM processor can be put in console mode as follows:

1. Console terminal BREAK key
2. HALT instruction, Kernel Stack Not Valid, or a Double Machine Check Error

In all cases, the console is now ready to accept commands.

The result of a HALT instruction, Kernel Stack Not Valid, or a Double Machine Check Error depends on the current setting of the implementation-dependent halt action. This may be either halt, restart/halt, boot/halt, or restart/boot/halt. The multiple action settings specify secondary or tertiary actions to be taken after a failure. For example, restart/halt indicates that if restart fails, the processor should halt.

If enabled, the BREAK key on the console terminal will always cause the PRISM processor to enter console mode.

11.2.3 Program Controlled Console I/O

Program controlled console I/O is necessary to allow system software to communicate with the operator during the bootstrap process. More information on communication with console terminals can be found in Chapter 8, Internal Processor Registers.

11.3 CONSOLE LANGUAGE

The PRISM console interprets commands typed on the console terminal and controls the operation of the PRISM processor.

Through the console terminal, an operator can boot the operating system, or a field service engineer can maintain the system. When the

processor is halted, the operator controls the system through the console command language. When the processor is in console mode, the operator is prompted for input with the string "Pn>>>" where n is the processor number.

It may be possible for the operator to put the system in an inconsistent state through the use of the console commands. For example, it may be possible to use the console to set bits in MBZ fields, or to set conflicting control bits. The operation of the processor in such a state is UNDEFINED.

11.3.1 Control Characters

In console I/O mode, several characters have special meanings.

- Carriage Return - Ends a command line. A null line terminated by a carriage return is treated as a valid, null command. Carriage return is echoed as carriage return, line feed.
- RUBOUT - When the operator types RUBOUT, the console ignores the entire line and prompts for another command.
- CTRL/U - When the operator types CTRL/U the console ignores the entire line and prompts for another command. If CTRL/U is typed on an empty line, it is echoed, and otherwise ignored. The console prompts for another command.
- CTRL/S - Stops output to the console terminal until CTRL/Q is typed. Additional input between CTRL/S and CTRL/Q is ignored. Additional CTRL/Ss before the CTRL/Q are ignored. CTRL/S and CTRL/Q are not echoed.
- CTRL/Q - Resumes output stopped by CTRL/S. Additional CTRL/Qs are ignored. CTRL/S and CTRL/Q are not echoed.
- BREAK - If the console is in console I/O mode, BREAK is ignored. If the console is in program I/O mode and BREAK is disabled, BREAK is passed to the operating system like any other character. If the console is in program I/O mode and BREAK is enabled, BREAK causes the processor to enter console I/O mode.

11.3.2 Command Syntax

All commands are abbreviated to a single character. Multiple adjacent spaces and tabs are treated as a single space by the console. Leading and trailing spaces and tabs are ignored. Illegal characters are ignored and echoed as BEL (ASCII code 7).

Command qualifiers must appear immediately after the command keyword without intervening spaces.

All numbers (addresses, data, counts) are in hexadecimal. (Note,

though, that symbolic register names include decimal digits.) Hex digits are 0 through 9, and A through F. The console does not distinguish between upper and lower case. Both are accepted.

11.3.3 Commands

Processor control commands:

- HALT
- INITIALIZE
- START
- CONTINUE
- BOOT

Data transfer commands:

- EXAMINE
- DEPOSIT

Console control commands:

- TEST

BOOT

Format:

B [<qualifier list>] [<device:>][<filename>]

Qualifiers:

- /<data> - This allows a console user to specify the bootstrap options parameter.
- /S - The console loads the bootstrap program and prompts for further console commands.
- /L:n - The console loads a one block bootstrap program from logical block number n.

Description:

The device specification format is consistent with the PRISM system software naming conventions.

The console initializes the processor, and loads a file and starts the system bootstrap program running; see Section 11.1 above. The default device and filename are implementation-dependent. The console searches through an implementation-dependent default search list.

The information supplied in this command is stored in the RPB.

CONTINUE

Format:

C

Qualifiers:

/ALL - Continue all processors in a multiprocessor system.

Description:

The processor begins instruction execution at the address currently contained in the Program Counter. Processor initialization is not performed. The console enters program I/O mode.

DEPOSIT

Format:

D [<qualifier list>] <address> <data>

Qualifiers:

See Table 11-2 in the description of the EXAMINE command.

Description:

Deposits the data into the address specified. If no address space or data size qualifiers are specified, the defaults are the last address space and data size used in a DEPOSIT or EXAMINE command. On each entry to console mode, the default address space is virtual memory, the default data size is longword, and the default address is zero.

If the specified data is larger than the destination data size, the console truncates the data to the least significant digits typed. If the specified data is smaller than the data size to be deposited, it is zero extended.

Examples:

D/P/B/N:200 0 0	Clears the first 512 bytes of physical memory.
D/V/L/N:4 1234 5	Deposits "5" into 4 longwords in virtual memory.
D/R/N:8 R2 FFFFFFFF	Loads general registers R2 through R9 with FFFFFFFF.
D/N:200 - 0	Clears 512 locations starting at the previous address.
D/I KSP 1234FACE	Deposits 1234FACE into the Kernel Stack Pointer.

EXAMINE

Format:

E [<qualifier list>] <address> [additional parameters]

Qualifiers:

See Table 11-2

Response:

<tab><address space identifier> <address> <data>

The address space identifier can be:

- I - Internal Processor Register.
- P - Physical memory. Note that when virtual memory is examined, the address space and address in the response are the translated physical address.
- R - Register.
- M - Machine-dependent address space.

Description:

Examines the contents of the specified address. If no address is specified, "+" is assumed.

Examining an IPR does not write into the scalar registers that would be written if an MFPR to the IPR were executed. The response the data.

TBCHK requires the ASN to be specified as an additional parameter.

Examples:

The response to E/I WHAMI on processor 3 is:

I WHAMI 00000003

The response to E/V 1234564 is:

P 0000FE3C 01739102

Where the virtual address 1234564 maps to the physical address FE3C and the contents of that location are 01739102.

The response to E/P FE3C is:

P 0000FE3C 01739102

Table 11-2: Qualifiers for Examine and Deposit

Qualifier	Meaning
/B	The data size is byte.
/W	The data size is word.
/L	The data size is longword.
/Q	The data size is quadword.
/V	The address space is virtual memory. No access and protection checking occurs. If the virtual address cannot be translated due an invalid PTE, the console issues a "?TNV" error message.
/P	The address space is physical memory. If an attempt is made to reference a non-existent memory location, The console issues a "?NXM" error message.
/R	The address space is registers. These are the scalar and vector registers. The following symbolic addresses can be used for either Examine or Deposit commands: SP - Current Mode Stack Pointer (scalar register R1). Rn - Scalar Register 'n'. The register number is in decimal and in the range 0-63. Vn[m] - Vector Register 'n', element 'm'. The register number is decimal and in the range 0-15; the element number is decimal and in the range 0-63.
/I	The address space is internal registers. These are the vector control registers, internal processor registers, Program Counter, and Processor Status. The following symbolic addresses can be used for either Examine or Deposit commands: PS - Processor Status. PC - Program Counter. VC - Vector Count. VL - Vector Length. VM - Vector Mask. ASTEN - AST Enable. CRCS - Console Receive Control Status. CTCS - Console Transmit Control Status. ICIE - Interval Clock Interrupt Enable. CCR - Cycle Count Register.

Table 11-2: Qualifiers for Examine and Deposit (Continued)

Qualifier	Meaning
	IPIE - Interprocessor Interrupt Enable. KSP - Kernel Stack Pointer. PRBR - Processor Base Register. SCBB - System Control Block Base. TOY - Time Of Year. USP - User Stack Pointer. The following symbolic addresses can be used for the Examine command only: ALL - All IPR's listed below, except TBCHK. ASN - Address Space Number. ASTSR - AST Summary Register. CRDB - Console Receive Data Buffer. PCBB - Privileged Context Block Base. SSN - System Serial Number. PTBR - Page Table Base Register. SID - System Identification. SISR - Software Interrupt Summary Register. TBCHK - Translation Buffer Check. WHAMI - Who-Am-I. The following symbolic addresses can be used for the Deposit command only: ASTRR - AST Request Register. CTDB - Console Transmit Data Buffer. IPIR - Interprocessor Interrupt Request. SIRR - Software Interrupt Request Register. TBIS - Translation Buffer Invalidate Single.
/M	(Optional) The address space is machine dependent.
/N:<count>	The address is the first of a range. The console examines or deposits the specified number of addresses starting at the first address. If the first address is the symbolic address "-", the succeeding addresses are at still larger addresses. The symbolic address specifies only the starting address, not the direction of succession. This qualifier may be used only for memory and register address spaces. Operation is UNDEFINED if it is used for internal registers.

Table 11-2: Qualifiers for Examine and Deposit (Continued)

Qualifier	Meaning

The address parameter may also be one of the following symbolic addresses:

'+' - The location immediately following the last location referenced in an examine or deposit. For references to physical or virtual memory spaces, the location referenced is the last address, plus the size of the last reference (1 for byte, 2 for word, 4 for longword, and 8 for quadword). For other address spaces, the address is the last addressed referenced, plus 1.

This variant may be used only for memory and register address spaces. Operation is UNDEFINED if it is used for internal registers.

'-' - The location immediately preceding the last location referenced in an examine or deposit. For references to physical or virtual memory spaces, the location referenced is the last address minus the size of this reference (1 for byte 2 for word, 4 for longword, and 8 for quadword). For other address spaces, the address is the last addressed referenced minus 1.

This variant may be used only for memory and register address spaces. Operation is UNDEFINED if it is used for internal registers.

'*' - The location last referenced in an examine or deposit.

'@' - The location addressed by the last location referenced in an examine or deposit.

HALT

Format:

H

Qualifiers:

/ALL - Halt all processors in a multiprocessor system.

Description:

The processor halts at the address currently contained in the Program Counter.

INITIALIZE

Format:

I

Qualifiers:

None

Description:

A processor initialization is performed; see Section 11.1.1.7 for initial register contents.

START

Format:

S [<address>]

Qualifiers:

None

Description:

The console starts instruction execution at the specified address. The default address is implementation dependent. Instructions are executed from virtual memory. The START command is equivalent to a DEPOSIT to PC, followed by a CONTINUE. No INITIALIZE is performed.

TEST

Format:

T [<qualifier list>]

Qualifiers:

Implementation-dependent

Description:

The PRISM processor executes a self test. All qualifiers are optional.

11.3.4 Error Messages

The following are the console error messages:

- BEL - Illegal characters are ignored and are echoed as BEL.
- ?NXM - Non-existent memory.
- ?TNV - Translation Not Valid.

Revision History:

Revision 3.0, 26 April 1988

1. Change Examine and Deposit of IPR's to not use scalar registers.
2. Minor changes to RPB.

Revision 2.0, 24 June 1986

1. New initial address space layout.
2. Clarify initial address space protection.
3. Add CYCCR.
4. Allow load from any logical block.
5. Restrict use of range and +,- parameters in Examine and Deposit.
6. Remove references to execute protection.
7. Add check for zero in RPB search.
8. SP is initialized on boot.
9. Split registers into internal and scalar or vector.
10. Minor editorial changes.

Revision 1.0, 22 December 1985

1. Initial review version.

CHAPTER 12

I/O ARCHITECTURE

12.1 SCOPE

This chapter specifies requirements on hardware and software implementors which ensure compatibility between PRISM processors, memory, and I/O subsystems.

Common to all implementations of the I/O subsystem is the fact that communications between the PRISM processors and the I/O processors/devices take place via the following mechanisms:

1. Reads/writes of locations in shared system memory.
2. Reads/writes of locations in PRISM I/O space.
3. Interrupts.

The I/O architecture specifies a number of requirements relevant to these mechanisms. I/O processors, buses, and I/O devices are collectively referred to as I/O devices in the remainder of this chapter. I/O devices are accessible to PRISM processors in the I/O portion of PRISM physical address space. Some I/O devices may be directly programmed by reads and writes from the PRISM processor to control registers in the I/O device. Other I/O devices may transfer data directly from/to PRISM system memory.

12.2 SYSTEM MEMORY

The requirements regarding system memory concern visibility and accessibility of system memory and with virtual memory translation by I/O devices:

1. All system memory must be visible and accessible to all processors all of the time.
2. All system memory must be accessible from all I/O devices. However, it does not have to be all visible all of the time. For example, bus maps are allowed. Therefore, I/O devices may need map registers to be set appropriately before they can access any given memory location.
3. Direct memory access I/O devices must provide a way to gather/scatter segments of a transfer to/from physical

memory. This can be via map registers (e.g. MBA), I/O bus map registers (e.g. UBA), or command/mapping control lists (e.g. BCA).

4. I/O devices may request interlocked access to system memory.

5. I/O devices are not allowed to interpret PRISM page tables.

6. Software must guarantee that all I/O buffers are quadword aligned.

7. I/O devices are allowed to read only the integral number of quadwords containing the buffer to be written from system memory to an I/O device. If the I/O device supports a transfer granularity of less than a quadword (e.g. magtape), a partial write of the last quadword to the I/O device may occur.

8. Writes from an I/O device to system memory may specify any byte count, but the remainder, if any, of the final quadword is UNPREDICTABLE.

9. Any buffered/non-buffered datapath provided by a bus map or bus adapter must be totally transparent to software.

12.3 PRISM I/O SPACE AND DEVICE INTERRUPTS

Every PRISM system must conform to certain requirements regarding accessibility of I/O devices and interrupt delivery and service:

1. A processor that can be interrupted by an I/O device must have access to that device's I/O registers.

2. It is preferred that all I/O devices be accessible from all processors, and I/O device interrupts be serviced by any processor.

3. Interrupts from a specific I/O device may be delivered to a single processor. In this case, the interrupting I/O device must be accessible from the interrupted processor.

4. An I/O device may direct an interrupt request to more than one processor. In this case, hardware must guarantee that the device interrupt is visible to software on exactly one processor.

5. I/O devices are allowed but not required to cause a system to enter the bootstrap sequence.

6. Processors are not required to be capable of initiating read-pause-write cycles. I/O devices or I/O buses that require such cycles are therefore not supported.

7. I/O devices are not required to be able to address all of PRISM I/O space.

12.4 GRANULARITY OF I/O SPACE ACCESSES

The physical address of an I/O register must be an integral multiple of the register size in bytes (which must be a power of 2); i.e. all registers must be aligned on natural boundaries. References using a length attribute other than the length of the register, or to unaligned addresses, produces UNPREDICTABLE results. For example, a byte reference to a word-length register will not necessarily respond by supplying or modifying the byte addressed.

Revision History:

Revision 3.0, 26 April 1988.

1. Add granularity of I/O accesses.

Revision 2.0, 24 June 1986

1. Initial review version.

APPENDIX A

INSTRUCTION SET SUMMARY

This appendix summarizes the instruction mnemonics and their opcode and function code fields in hex. There are three listings:

- Functional group listing - Groups related instructions together.
- Mnemonic listing - Lists the instructions sorted by mnemonic.
- Opcode listing - Lists the instructions sorted by opcode and function code.

A.1 ENCODING HINTS

The instruction encoding was designed so that it would simplify instruction-issue logic. The following comments and equations may be helpful in understanding the encoding that was chosen. In the following, the term OPCODE is used for instruction bits <31:26> and FUNC is used for instruction bits <13:9>.

1. All scalar load and store instructions have OPCODE<5:3> equal to 111(bin). OPCODE<2> is a 0 for load and a 1 for store. OPCODE<1:0> specifies the data size (0 for byte, 1 for word 2 for longword, and 3 for quadword).
2. All floating-point instructions encode floating underflow enable in FUNC<3> (0 for underflow disabled and 1 for underflow enabled).
3. All floating-point instructions encode floating rounding mode in FUNC<2> (0 for round toward zero and 1 for VAX rounding).
4. All vector instructions use FUNC<4> to determine whether the Ra field selects a scalar or a vector register (0 for scalar Ra and 1 for vector Ra).

A.2 FUNCTIONAL GROUP LISTING

Mnemonic		Opcode (hex)	Function Code (hex)
LDB	d(rb),ra	38	-
LDW	d(rb),ra	39	-
LDL	d(rb),ra	3A	-
LDQ	d(rb),ra	3B	-
STB	ra,d(rb)	3C	-
STW	ra,d(rb)	3D	-
STL	ra,d(rb)	3E	-
STQ	ra,d(rb)	3F	-
VLDL	ra,rb,vc	30	00
VLDQ	ra,rb,vc	30	01
VLDL/W	ra,rb,vc	30	02
VLDQ/W	ra,rb,vc	30	03
VSTL	ra,rb,vc	30	04
VSTQ	ra,rb,vc	30	05
VGATHL	ra,vb,vc	31	00
VGATHQ	ra,vb,vc	31	01
VGATHL/W	ra,vb,vc	31	02
VGATHQ/W	ra,vb,vc	31	03
VSCATL	ra,vb,vc	31	04
VSCATQ	ra,vb,vc	31	05
RDVL	rc	32	00
RDVC	rc	32	01
RDVML	rc	32	02
RDVMH	rc	32	03
WRVL	ra	33	00
WRVC	ra	33	01
WRVML	ra	33	02
WRVMH	ra	33	03
EPIRES3		36	-
EPIRES4		37	-
BEQ	ra,dest	20	-
BNE	ra,dest	21	-
BGT	ra,dest	22	-
BLE	ra,dest	23	-
BGE	ra,dest	24	-
BLT	ra,dest	25	-
BLBC	ra,dest	26	-
BLBS	ra,dest	27	-
JSR	ra,dest	28	-
JSR	ra,(rb)	29	00
FLBC	ra	2A	-
EPIRES0		2D	-
EPIRES1		2E	-

EPIRES2		2F	-
ADD	ra,rb,rc	01	00
ADD/V	ra,rb,rc	01	01
SUB	ra,rb,rc	01	08
SUB/V	ra,rb,rc	01	09
CMPEQ	ra,rb,rc	02	08
CMPNE	ra,rb,rc	02	09
CMPGT	ra,rb,rc	02	0A
CMPLE	ra,rb,rc	02	0B
CMPGE	ra,rb,rc	02	0C
CMPLT	ra,rb,rc	02	0D
CMPUGT	ra,rb,rc	02	1A
CMPULE	ra,rb,rc	02	1B
CMPUGE	ra,rb,rc	02	1C
CMPULT	ra,rb,rc	02	1D
SLL	ra,rb,rc	03	04
SRL	ra,rb,rc	03	05
SRA	ra,rb,rc	03	06
ROT	ra,rb,rc	03	07
AND	ra,rb,rc	03	00
BIC	ra,rb,rc	03	08
OR	ra,rb,rc	03	01
ORNOT	ra,rb,rc	03	09
XOR	ra,rb,rc	03	02
EQV	ra,rb,rc	03	0A
LDA	d(rb),ra	04	-
CVTFL	ra,rc	05	04
CVTFL/C	ra,rc	05	00
CVTLF	ra,rc	05	05
CVTLF/C	ra,rc	05	01
CVTFG	ra,rc	07	02
CVTLG	ra,rc	07	03
CVTGL	ra,rc	06	04
CVTGL/C	ra,rc	06	00
CVTGF	ra,rc	06	05
CVTGF/C	ra,rc	06	01
CVTGF/U	ra,rc	06	0D
CVTGF/CU	ra,rc	06	09
ADDG	ra,rb,rc	08	04
ADDG/C	ra,rb,rc	08	00
ADDG/U	ra,rb,rc	08	0C
ADDG/CU	ra,rb,rc	08	08
SUBG	ra,rb,rc	08	05
SUBG/C	ra,rb,rc	08	01
SUBG/U	ra,rb,rc	08	0D
SUBG/CU	ra,rb,rc	08	09
ADDF	ra,rb,rc	09	04
ADDF/C	ra,rb,rc	09	00
ADDF/U	ra,rb,rc	09	0C
ADDF/CU	ra,rb,rc	09	08
SUBF	ra,rb,rc	09	05

SUBF/C	ra,rb,rc	09	01
SUBF/U	ra,rb,rc	09	0D
SUBF/CU	ra,rb,rc	09	09
CMPGEQ	ra,rb,rc	0A	00
CMPGNE	ra,rb,rc	0A	01
CMPGGT	ra,rb,rc	0A	02
CMPGLE	ra,rb,rc	0A	03
CMPGGE	ra,rb,rc	0A	04
CMPGLT	ra,rb,rc	0A	05
CMPFEQ	ra,rb,rc	0B	00
CMPFNE	ra,rb,rc	0B	01
CMPFGT	ra,rb,rc	0B	02
CMPFLE	ra,rb,rc	0B	03
CMPFGE	ra,rb,rc	0B	04
CMPFLT	ra,rb,rc	0B	05
DIVG	ra,rb,rc	0C	04
DIVG/C	ra,rb,rc	0C	00
DIVG/U	ra,rb,rc	0C	0C
DIVG/CU	ra,rb,rc	0C	08
DIVF	ra,rb,rc	0D	04
DIVF/C	ra,rb,rc	0D	00
DIVF/U	ra,rb,rc	0D	0C
DIVF/CU	ra,rb,rc	0D	08
DIV	ra,rb,rc	0D	02
DIV/V	ra,rb,rc	0D	0A
MULG	ra,rb,rc	0E	04
MULG/C	ra,rb,rc	0E	00
MULG/U	ra,rb,rc	0E	0C
MULG/CU	ra,rb,rc	0E	08
MULF	ra,rb,rc	0F	04
MULF/C	ra,rb,rc	0F	00
MULF/U	ra,rb,rc	0F	0C
MULF/CU	ra,rb,rc	0F	08
MULL	ra,rb,rc	0F	02
MULL/V	ra,rb,rc	0F	0A
UMULH	ra,rb,rc	0F	03
VMERGE	ra,vb,vc	10	00
VMERGE	va,vb,vc	10	10
IOTA	ra,vc	10	01
VADD	ra,vb,vc	11	00
VADD/V	ra,vb,vc	11	08
VSUB	ra,vb,vc	11	01
VSUB/V	ra,vb,vc	11	09
VADD	va,vb,vc	11	10
VADD/V	va,vb,vc	11	18
VSUB	va,vb,vc	11	11
VSUB/V	va,vb,vc	11	19
VCMPEQ	ra,vb	12	00
VCMPNE	ra,vb	12	01
VCMPGT	ra,vb	12	02
VCMPLE	ra,vb	12	03

VCMPGE	ra,vb	12	04
VCMPLT	ra,vb	12	05
VCMPUGT	ra,vb	12	0A
VCMPULE	ra,vb	12	0B
VCMPUGE	ra,vb	12	0C
VCMPULT	ra,vb	12	0D
VCMPEQ	va,vb	12	10
VCMPNE	va,vb	12	11
VCMPGT	va,vb	12	12
VCMPLE	va,vb	12	13
VCMPGE	va,vb	12	14
VCMPLT	va,vb	12	15
VCMPUGT	va,vb	12	1A
VCMPULE	va,vb	12	1B
VCMPUGE	va,vb	12	1C
VCMPULT	va,vb	12	1D
VSLL	ra,vb,vc	13	04
VSRL	ra,vb,vc	13	05
VAND	ra,vb,vc	13	00
VBIC	ra,vb,vc	13	08
VOR	ra,vb,vc	13	01
VORNOT	ra,vb,vc	13	09
VXOR	ra,vb,vc	13	02
VEQV	ra,vb,vc	13	0A
VSLL	va,vb,vc	13	14
VSRL	va,vb,vc	13	15
VAND	va,vb,vc	13	10
VBIC	va,vb,vc	13	18
VOR	va,vb,vc	13	11
VORNOT	va,vb,vc	13	19
VXOR	va,vb,vc	13	12
VEQV	va,vb,vc	13	1A
VCVTFL	vb,vc	17	14
VCVTFL/C	vb,vc	17	10
VCVTLF	vb,vc	17	15
VCVTLF/C	vb,vc	17	11
VCVTFG	vb,vc	17	12
VCVTLG	vb,vc	17	13
VCVTGL	vb,vc	16	14
VCVTGL/C	vb,vc	16	10
VCVTGF	vb,vc	16	15
VCVTGF/C	vb,vc	16	11
VCVTGF/U	vb,vc	16	1D
VCVTGF/CU	vb,vc	16	19
VADDG	ra,vb,vc	18	04
VADDG/C	ra,vb,vc	18	00
VADDG/U	ra,vb,vc	18	0C
VADDG/CU	ra,vb,vc	18	08
VSUBG	ra,vb,vc	18	05
VSUBG/C	ra,vb,vc	18	01
VSUBG/U	ra,vb,vc	18	0D
VSUBG/CU	ra,vb,vc	18	09
VADDG	va,vb,vc	18	14
VADDG/C	va,vb,vc	18	10
VADDG/U	va,vb,vc	18	1C
VADDG/CU	va,vb,vc	18	18

VSUBG	va,vb,vc	18	15
VSUBG/C	va,vb,vc	18	11
VSUBG/U	va,vb,vc	18	1D
VSUBG/CU	va,vb,vc	18	19
VADDF	ra,vb,vc	19	04
VADDF/C	ra,vb,vc	19	00
VADDF/U	ra,vb,vc	19	0C
VADDF/CU	ra,vb,vc	19	08
VSUBF	ra,vb,vc	19	05
VSUBF/C	ra,vb,vc	19	01
VSUBF/U	ra,vb,vc	19	0D
VSUBF/CU	ra,vb,vc	19	09
VADDF	va,vb,vc	19	14
VADDF/C	va,vb,vc	19	10
VADDF/U	va,vb,vc	19	1C
VADDF/CU	va,vb,vc	19	18
VSUBF	va,vb,vc	19	15
VSUBF/C	va,vb,vc	19	11
VSUBF/U	va,vb,vc	19	1D
VSUBF/CU	va,vb,vc	19	19
VCMPGEQ	ra,vb	1A	00
VCMPGNE	ra,vb	1A	01
VCMPGGT	ra,vb	1A	02
VCMPGLE	ra,vb	1A	03
VCMPGGE	ra,vb	1A	04
VCMPGLT	ra,vb	1A	05
VCMPGEQ	va,vb	1A	10
VCMPGNE	va,vb	1A	11
VCMPGGT	va,vb	1A	12
VCMPGLE	va,vb	1A	13
VCMPGGE	va,vb	1A	14
VCMPGLT	va,vb	1A	15
VCMPFEQ	ra,vb	1B	00
VCMPFNE	ra,vb	1B	01
VCMPFGT	ra,vb	1B	02
VCMPFLE	ra,vb	1B	03
VCMPFGE	ra,vb	1B	04
VCMPFLT	ra,vb	1B	05
VCMPFEQ	va,vb	1B	10
VCMPFNE	va,vb	1B	11
VCMPFGT	va,vb	1B	12
VCMPFLE	va,vb	1B	13
VCMPFGE	va,vb	1B	14
VCMPFLT	va,vb	1B	15
VDIVG	ra,vb,vc	1C	04
VDIVG/C	ra,vb,vc	1C	00
VDIVG/U	ra,vb,vc	1C	0C
VDIVG/CU	ra,vb,vc	1C	08
VDIVG	va,vb,vc	1C	14
VDIVG/C	va,vb,vc	1C	10
VDIVG/U	va,vb,vc	1C	1C
VDIVG/CU	va,vb,vc	1C	18
VDIVF	ra,vb,vc	1D	04
VDIVF/C	ra,vb,vc	1D	00
VDIVF/U	ra,vb,vc	1D	0C

VDIVF/CU	ra,vb,vc	1D	08
VDIVF	va,vb,vc	1D	14
VDIVF/C	va,vb,vc	1D	10
VDIVF/U	va,vb,vc	1D	1C
VDIVF/CU	va,vb,vc	1D	18
VMULG	ra,vb,vc	1E	04
VMULG/C	ra,vb,vc	1E	00
VMULG/U	ra,vb,vc	1E	0C
VMULG/CU	ra,vb,vc	1E	08
VMULG	va,vb,vc	1E	14
VMULG/C	va,vb,vc	1E	10
VMULG/U	va,vb,vc	1E	1C
VMULG/CU	va,vb,vc	1E	18
VMULF	ra,vb,vc	1F	04
VMULF/C	ra,vb,vc	1F	00
VMULF/U	ra,vb,vc	1F	0C
VMULF/CU	ra,vb,vc	1F	08
VMULF	va,vb,vc	1F	14
VMULF/C	va,vb,vc	1F	10
VMULF/U	va,vb,vc	1F	1C
VMULF/CU	va,vb,vc	1F	18
VMULL	ra,vb,vc	1F	02
VMULL/V	ra,vb,vc	1F	0A
VUMULH	ra,vb,vc	1F	03
VMULL	va,vb,vc	1F	12
VMULL/V	va,vb,vc	1F	1A
VUMULH	va,vb,vc	1F	13
DRAIN		2B	00
DRAINM		2B	10
DRAINV		2B	11
HALT		00	00
BOOT		00	01
REI		00	02
BPT		00	03
IFLUSH		00	31
MOVPS		00	32
PROBER		00	0A
PROBEW		00	0B
SWASTEN		00	05
SWIPL		00	06
SWPCTX		00	07
RMAQI		00	38
CMPSWQI		00	39
RMALI		00	3A
CMPSWLI		00	3B
CMPSWQIP		00	3C
LDQP		00	3D
STQP		00	3E
RDCC		00	0C

WRCC		00	0D
TBFLUSH		00	08
MFPR	USP	00	C3
MTPR	USP	00	83
MFPR	PTBR	00	C4
MFPR	PCBB	00	C5
MFPR	SCBB	00	C6
MTPR	SCBB	00	86
MTPR	ASTRR	00	87
MFPR	ASTSR	00	C8
MTPR	ASTEN	00	89
MFPR	ASTEN	00	C9
MTPR	SIRR	00	8A
MFPR	SISR	00	CA
MFPR	ICIE	00	CB
MTPR	ICIE	00	8B
MFPR	TOY	00	CC
MTPR	TOY	00	8C
MFPR	ASN	00	CD
MFPR	TBCHK	00	CE
MTPR	TBIS	00	8F
MTPR	IPIR	00	91
MFPR	IPIE	00	D2
MTPR	IPIE	00	92
MFPR	PRBR	00	D3
MTPR	PRBR	00	93
MFPR	WHAMI	00	D4
MFPR	SID	00	D5
MFPR	SSN	00	D6
MFPR	CRCS	00	D7
MTPR	CRCS	00	97
MFPR	CRDB	00	D8
MFPR	CTCS	00	D9
MTPR	CTCS	00	99
MTPR	CTDB	00	9A
MTPR	VEN	00	9B
MFPR	MCES	00	DC
MTPR	MCES	00	9C
reserved		14	00
reserved		15	00
reserved		2C	00
reserved		34	-
reserved		35	-

A.3 MNEMONIC LISTING

Mnemonic		Opcode (hex)	Function Code (hex)
ADD	ra,rb,rc	01	00
ADD/V	ra,rb,rc	01	01
ADDF	ra,rb,rc	09	04
ADDF/C	ra,rb,rc	09	00
ADDF/CU	ra,rb,rc	09	08
ADDF/U	ra,rb,rc	09	0C
ADDG	ra,rb,rc	08	04
ADDG/C	ra,rb,rc	08	00
ADDG/CU	ra,rb,rc	08	08
ADDG/U	ra,rb,rc	08	0C
AND	ra,rb,rc	03	00
BEQ	ra,dest	20	-
BGE	ra,dest	24	-
BGT	ra,dest	22	-
BIC	ra,rb,rc	03	08
BLBC	ra,dest	26	-
BLBS	ra,dest	27	-
BLE	ra,dest	23	-
BLT	ra,dest	25	-
BNE	ra,dest	21	-
BOOT		00	01
BPT		00	03
CMPEQ	ra,rb,rc	02	08
CMPFEQ	ra,rb,rc	0B	00
CMPFGE	ra,rb,rc	0B	04
CMPFGT	ra,rb,rc	0B	02
CMPFLE	ra,rb,rc	0B	03
CMPFLT	ra,rb,rc	0B	05
CMPFNE	ra,rb,rc	0B	01
CMPGE	ra,rb,rc	02	0C
CMPGEQ	ra,rb,rc	0A	00
CMPGGE	ra,rb,rc	0A	04
CMPGGT	ra,rb,rc	0A	02
CMPGLE	ra,rb,rc	0A	03
CMPGLT	ra,rb,rc	0A	05
CMPGNE	ra,rb,rc	0A	01
CMPGT	ra,rb,rc	02	0A
CMPLE	ra,rb,rc	02	0B
CMPLT	ra,rb,rc	02	0D
CMPNE	ra,rb,rc	02	09
CMPSWLI		00	3B
CMPSWQI		00	39
CMPSWQIP		00	3C
CMPUGE	ra,rb,rc	02	1C
CMPUGT	ra,rb,rc	02	1A
CMPULE	ra,rb,rc	02	1B
CMPULT	ra,rb,rc	02	1D
CVTFG	ra,rc	07	02
CVTFL	ra,rc	05	04
CVTFL/C	ra,rc	05	00
CVTGF	ra,rc	06	05
CVTGF/C	ra,rc	06	01
CVTGF/CU	ra,rc	06	09

CVTGF/U	ra,rc	06	0D
CVTGL	ra,rc	06	04
CVTGL/C	ra,rc	06	00
CVTLF	ra,rc	05	05
CVTLF/C	ra,rc	05	01
CVTLG	ra,rc	07	03
DIV	ra,rb,rc	0D	02
DIV/V	ra,rb,rc	0D	0A
DIVF	ra,rb,rc	0D	04
DIVF/C	ra,rb,rc	0D	00
DIVF/CU	ra,rb,rc	0D	08
DIVF/U	ra,rb,rc	0D	0C
DIVG	ra,rb,rc	0C	04
DIVG/C	ra,rb,rc	0C	00
DIVG/CU	ra,rb,rc	0C	08
DIVG/U	ra,rb,rc	0C	0C
DRAIN		2B	00
DRAINM		2B	10
DRAINV		2B	11
EPIRES0		2D	-
EPIRES1		2E	-
EPIRES2		2F	-
EPIRES3		36	-
EPIRES4		37	-
EQV	ra,rb,rc	03	0A
FLBC	ra	2A	-
HALT		00	00
IFLUSH		00	31
IOTA	ra,vc	10	01
JSR	ra,(rb)	29	00
JSR	ra,dest	28	-
LDA	d(rb),ra	04	-
LDB	d(rb),ra	38	-
LDL	d(rb),ra	3A	-
LDQ	d(rb),ra	3B	-
LDQP		00	3D
LDW	d(rb),ra	39	-
MFPR	ASN	00	CD
MFPR	ASTEN	00	C9
MFPR	ASTSR	00	C8
MFPR	CRCS	00	D7
MFPR	CRDB	00	D8
MFPR	CTCS	00	D9
MFPR	ICIE	00	CB
MFPR	IPIE	00	D2
MFPR	MCES	00	DC
MFPR	PCBB	00	C5
MFPR	PRBR	00	D3
MFPR	PTBR	00	C4
MFPR	SCBB	00	C6
MFPR	SID	00	D5
MFPR	SISR	00	CA
MFPR	SSN	00	D6
MFPR	TBCHK	00	CE
MFPR	TOY	00	CC
MFPR	USP	00	C3
MFPR	WHAMI	00	D4
MOVPS		00	32
MTPR	ASTEN	00	89
MTPR	ASTRR	00	87

MTPR	CRCS	00	97
MTPR	CTCS	00	99
MTPR	CTDB	00	9A
MTPR	ICIE	00	8B
MTPR	IPIE	00	92
MTPR	IPIR	00	91
MTPR	MCES	00	9C
MTPR	PRBR	00	93
MTPR	SCBB	00	86
MTPR	SIRR	00	8A
MTPR	TBIS	00	8F
MTPR	TOY	00	8C
MTPR	USP	00	83
MTPR	VEN	00	9B
MULF	ra,rb,rc	0F	04
MULF/C	ra,rb,rc	0F	00
MULF/CU	ra,rb,rc	0F	08
MULF/U	ra,rb,rc	0F	0C
MULG	ra,rb,rc	0E	04
MULG/C	ra,rb,rc	0E	00
MULG/CU	ra,rb,rc	0E	08
MULG/U	ra,rb,rc	0E	0C
MULL	ra,rb,rc	0F	02
MULL/V	ra,rb,rc	0F	0A
OR	ra,rb,rc	03	01
ORNOT	ra,rb,rc	03	09
PROBER		00	0A
PROBEW		00	0B
RDCC		00	0C
RDVC	rc	32	01
RDVL	rc	32	00
RDVMH	rc	32	03
RDVML	rc	32	02
REI		00	02
RMALI		00	3A
RMAQI		00	38
ROT	ra,rb,rc	03	07
SLL	ra,rb,rc	03	04
SRA	ra,rb,rc	03	06
SRL	ra,rb,rc	03	05
STB	ra,d(rb)	3C	-
STL	ra,d(rb)	3E	-
STQ	ra,d(rb)	3F	-
STQP		00	3E
STW	ra,d(rb)	3D	-
SUB	ra,rb,rc	01	08
SUB/V	ra,rb,rc	01	09
SUBF	ra,rb,rc	09	05
SUBF/C	ra,rb,rc	09	01
SUBF/CU	ra,rb,rc	09	09
SUBF/U	ra,rb,rc	09	0D
SUBG	ra,rb,rc	08	05
SUBG/C	ra,rb,rc	08	01
SUBG/CU	ra,rb,rc	08	09
SUBG/U	ra,rb,rc	08	0D
SWASTEN		00	05
SWIPL		00	06
SWPCTX		00	07
TBFLUSH		00	08
UMULH	ra,rb,rc	0F	03

VADD	ra,vb,vc	11	00
VADD	va,vb,vc	11	10
VADD/V	ra,vb,vc	11	08
VADD/V	va,vb,vc	11	18
VADDF	ra,vb,vc	19	04
VADDF	va,vb,vc	19	14
VADDF/C	ra,vb,vc	19	00
VADDF/C	va,vb,vc	19	10
VADDF/CU	ra,vb,vc	19	08
VADDF/CU	va,vb,vc	19	18
VADDF/U	ra,vb,vc	19	0C
VADDF/U	va,vb,vc	19	1C
VADDG	ra,vb,vc	18	04
VADDG	va,vb,vc	18	14
VADDG/C	ra,vb,vc	18	00
VADDG/C	va,vb,vc	18	10
VADDG/CU	ra,vb,vc	18	08
VADDG/CU	va,vb,vc	18	18
VADDG/U	ra,vb,vc	18	0C
VADDG/U	va,vb,vc	18	1C
VAND	ra,vb,vc	13	00
VAND	va,vb,vc	13	10
VBIC	ra,vb,vc	13	08
VBIC	va,vb,vc	13	18
VCMPEQ	ra,vb	12	00
VCMPEQ	va,vb	12	10
VCMPFEQ	ra,vb	1B	00
VCMPFEQ	va,vb	1B	10
VCMPFGE	ra,vb	1B	04
VCMPFGE	va,vb	1B	14
VCMPFGT	ra,vb	1B	02
VCMPFGT	va,vb	1B	12
VCMPFLE	ra,vb	1B	03
VCMPFLE	va,vb	1B	13
VCMPFLT	ra,vb	1B	05
VCMPFLT	va,vb	1B	15
VCMPFNE	ra,vb	1B	01
VCMPFNE	va,vb	1B	11
VCMPGE	ra,vb	12	04
VCMPGE	va,vb	12	14
VCMPGEQ	ra,vb	1A	00
VCMPGEQ	va,vb	1A	10
VCMPGGE	ra,vb	1A	04
VCMPGGE	va,vb	1A	14
VCMPGGT	ra,vb	1A	02
VCMPGGT	va,vb	1A	12
VCMPGLE	ra,vb	1A	03
VCMPGLE	va,vb	1A	13
VCMPGLT	ra,vb	1A	05
VCMPGLT	va,vb	1A	15
VCMPGNE	ra,vb	1A	01
VCMPGNE	va,vb	1A	11
VCMPGT	ra,vb	12	02
VCMPGT	va,vb	12	12
VCMPLE	ra,vb	12	03
VCMPLE	va,vb	12	13
VCMPLT	ra,vb	12	05
VCMPLT	va,vb	12	15
VCMPNE	ra,vb	12	01
VCMPNE	va,vb	12	11

VCMPUGE	ra,vb	12	0C
VCMPUGE	va,vb	12	1C
VCMPUGT	ra,vb	12	0A
VCMPUGT	va,vb	12	1A
VCMPULE	ra,vb	12	0B
VCMPULE	va,vb	12	1B
VCMPULT	ra,vb	12	0D
VCMPULT	va,vb	12	1D
VCVTFG	vb,vc	17	12
VCVTFL	vb,vc	17	14
VCVTFL/C	vb,vc	17	10
VCVTGF	vb,vc	16	15
VCVTGF/C	vb,vc	16	11
VCVTGF/CU	vb,vc	16	19
VCVTGF/U	vb,vc	16	1D
VCVTGL	vb,vc	16	14
VCVTGL/C	vb,vc	16	10
VCVTLF	vb,vc	17	15
VCVTLF/C	vb,vc	17	11
VCVTLG	vb,vc	17	13
VDIVF	ra,vb,vc	1D	04
VDIVF	va,vb,vc	1D	14
VDIVF/C	ra,vb,vc	1D	00
VDIVF/C	va,vb,vc	1D	10
VDIVF/CU	ra,vb,vc	1D	08
VDIVF/CU	va,vb,vc	1D	18
VDIVF/U	ra,vb,vc	1D	0C
VDIVF/U	va,vb,vc	1D	1C
VDIVG	ra,vb,vc	1C	04
VDIVG	va,vb,vc	1C	14
VDIVG/C	ra,vb,vc	1C	00
VDIVG/C	va,vb,vc	1C	10
VDIVG/CU	ra,vb,vc	1C	08
VDIVG/CU	va,vb,vc	1C	18
VDIVG/U	ra,vb,vc	1C	0C
VDIVG/U	va,vb,vc	1C	1C
VEQV	ra,vb,vc	13	0A
VEQV	va,vb,vc	13	1A
VGATHL	ra,vb,vc	31	00
VGATHL/W	ra,vb,vc	31	02
VGATHQ	ra,vb,vc	31	01
VGATHQ/W	ra,vb,vc	31	03
VLDL	ra,rb,vc	30	00
VLDL/W	ra,rb,vc	30	02
VLDQ	ra,rb,vc	30	01
VLDQ/W	ra,rb,vc	30	03
VMERGE	ra,vb,vc	10	00
VMERGE	va,vb,vc	10	10
VMULF	ra,vb,vc	1F	04
VMULF	va,vb,vc	1F	14
VMULF/C	ra,vb,vc	1F	00
VMULF/C	va,vb,vc	1F	10
VMULF/CU	ra,vb,vc	1F	08
VMULF/CU	va,vb,vc	1F	18
VMULF/U	ra,vb,vc	1F	0C
VMULF/U	va,vb,vc	1F	1C
VMULG	ra,vb,vc	1E	04
VMULG	va,vb,vc	1E	14
VMULG/C	ra,vb,vc	1E	00
VMULG/C	va,vb,vc	1E	10

VMULG/CU	ra,vb,vc	1E	08
VMULG/CU	va,vb,vc	1E	18
VMULG/U	ra,vb,vc	1E	0C
VMULG/U	va,vb,vc	1E	1C
VMULL	ra,vb,vc	1F	02
VMULL	va,vb,vc	1F	12
VMULL/V	ra,vb,vc	1F	0A
VMULL/V	va,vb,vc	1F	1A
VOR	ra,vb,vc	13	01
VOR	va,vb,vc	13	11
VORNOT	ra,vb,vc	13	09
VORNOT	va,vb,vc	13	19
VSCATL	ra,vb,vc	31	04
VSCATQ	ra,vb,vc	31	05
VSLL	ra,vb,vc	13	04
VSLL	va,vb,vc	13	14
VSRL	ra,vb,vc	13	05
VSRL	va,vb,vc	13	15
VSTL	ra,rb,vc	30	04
VSTQ	ra,rb,vc	30	05
VSUB	ra,vb,vc	11	01
VSUB	va,vb,vc	11	11
VSUB/V	ra,vb,vc	11	09
VSUB/V	va,vb,vc	11	19
VSUBF	ra,vb,vc	19	05
VSUBF	va,vb,vc	19	15
VSUBF/C	ra,vb,vc	19	01
VSUBF/C	va,vb,vc	19	11
VSUBF/CU	ra,vb,vc	19	09
VSUBF/CU	va,vb,vc	19	19
VSUBF/U	ra,vb,vc	19	0D
VSUBF/U	va,vb,vc	19	1D
VSUBG	ra,vb,vc	18	05
VSUBG	va,vb,vc	18	15
VSUBG/C	ra,vb,vc	18	01
VSUBG/C	va,vb,vc	18	11
VSUBG/CU	ra,vb,vc	18	09
VSUBG/CU	va,vb,vc	18	19
VSUBG/U	ra,vb,vc	18	0D
VSUBG/U	va,vb,vc	18	1D
VUMULH	ra,vb,vc	1F	03
VUMULH	va,vb,vc	1F	13
VXOR	ra,vb,vc	13	02
VXOR	va,vb,vc	13	12
WRCC		00	0D
WRVC	ra	33	01
WRVL	ra	33	00
WRVMH	ra	33	03
WRVML	ra	33	02
XOR	ra,rb,rc	03	02
reserved		14	00
reserved		15	00
reserved		2C	00
reserved		34	-
reserved		35	-

A.4 OPCODE LISTING

Mnemonic		Opcode (hex)	Function Code (hex)
HALT		00	00
BOOT		00	01
REI		00	02
BPT		00	03
SWASTEN		00	05
SWIPL		00	06
SWPCTX		00	07
TBFLUSH		00	08
PROBER		00	0A
PROBEW		00	0B
RDCC		00	0C
WRCC		00	0D
IFLUSH		00	31
MOVPS		00	32
RMAQI		00	38
CMPSWQI		00	39
RMALI		00	3A
CMPSWLI		00	3B
CMPSWQIP		00	3C
LDQP		00	3D
STQP		00	3E
MTPR	USP	00	83
MTPR	SCBB	00	86
MTPR	ASTRR	00	87
MTPR	ASTEN	00	89
MTPR	SIRR	00	8A
MTPR	ICIE	00	8B
MTPR	TOY	00	8C
MTPR	TBIS	00	8F
MTPR	IPIR	00	91
MTPR	IPIE	00	92
MTPR	PRBR	00	93
MTPR	CRCS	00	97
MTPR	CTCS	00	99
MTPR	CTDB	00	9A
MTPR	VEN	00	9B
MTPR	MCES	00	9C
MFPR	USP	00	C3
MFPR	PTBR	00	C4
MFPR	PCBB	00	C5
MFPR	SCBB	00	C6
MFPR	ASTSR	00	C8
MFPR	ASTEN	00	C9
MFPR	SISR	00	CA
MFPR	ICIE	00	CB
MFPR	TOY	00	CC
MFPR	ASN	00	CD
MFPR	TBCHK	00	CE
MFPR	IPIE	00	D2
MFPR	PRBR	00	D3
MFPR	WHAMI	00	D4
MFPR	SID	00	D5
MFPR	SSN	00	D6

MFPR	CRCS	00	D7
MFPR	CRDB	00	D8
MFPR	CTCS	00	D9
MFPR	MCES	00	DC
ADD	ra,rb,rc	01	00
ADD/V	ra,rb,rc	01	01
SUB	ra,rb,rc	01	08
SUB/V	ra,rb,rc	01	09
CMPEQ	ra,rb,rc	02	08
CMPNE	ra,rb,rc	02	09
CMPGT	ra,rb,rc	02	0A
CMPLE	ra,rb,rc	02	0B
CMPGE	ra,rb,rc	02	0C
CMPLT	ra,rb,rc	02	0D
CMPUGT	ra,rb,rc	02	1A
CMPULE	ra,rb,rc	02	1B
CMPUGE	ra,rb,rc	02	1C
CMPULT	ra,rb,rc	02	1D
AND	ra,rb,rc	03	00
OR	ra,rb,rc	03	01
XOR	ra,rb,rc	03	02
SLL	ra,rb,rc	03	04
SRL	ra,rb,rc	03	05
SRA	ra,rb,rc	03	06
ROT	ra,rb,rc	03	07
BIC	ra,rb,rc	03	08
ORNOT	ra,rb,rc	03	09
EQV	ra,rb,rc	03	0A
LDA	d(rb),ra	04	-
CVTFL/C	ra,rc	05	00
CVTLF/C	ra,rc	05	01
CVTFL	ra,rc	05	04
CVTLF	ra,rc	05	05
CVTGL/C	ra,rc	06	00
CVTGF/C	ra,rc	06	01
CVTGL	ra,rc	06	04
CVTGF	ra,rc	06	05
CVTGF/CU	ra,rc	06	09
CVTGF/U	ra,rc	06	0D
CVTFG	ra,rc	07	02
CVTLG	ra,rc	07	03
ADDG/C	ra,rb,rc	08	00
SUBG/C	ra,rb,rc	08	01
ADDG	ra,rb,rc	08	04
SUBG	ra,rb,rc	08	05
ADDG/CU	ra,rb,rc	08	08
SUBG/CU	ra,rb,rc	08	09
ADDG/U	ra,rb,rc	08	0C
SUBG/U	ra,rb,rc	08	0D
ADDF/C	ra,rb,rc	09	00
SUBF/C	ra,rb,rc	09	01
ADDF	ra,rb,rc	09	04
SUBF	ra,rb,rc	09	05
ADDF/CU	ra,rb,rc	09	08
SUBF/CU	ra,rb,rc	09	09
ADDF/U	ra,rb,rc	09	0C
SUBF/U	ra,rb,rc	09	0D
CMPGEQ	ra,rb,rc	0A	00
CMPGNE	ra,rb,rc	0A	01
CMPGGT	ra,rb,rc	0A	02

CMPGLE	ra,rb,rc	0A	03
CMPGGE	ra,rb,rc	0A	04
CMPGLT	ra,rb,rc	0A	05
CMPFEQ	ra,rb,rc	0B	00
CMPFNE	ra,rb,rc	0B	01
CMPFGT	ra,rb,rc	0B	02
CMPFLE	ra,rb,rc	0B	03
CMPFGE	ra,rb,rc	0B	04
CMPFLT	ra,rb,rc	0B	05
DIVG/C	ra,rb,rc	0C	00
DIVG	ra,rb,rc	0C	04
DIVG/CU	ra,rb,rc	0C	08
DIVG/U	ra,rb,rc	0C	0C
DIVF/C	ra,rb,rc	0D	00
DIV	ra,rb,rc	0D	02
DIVF	ra,rb,rc	0D	04
DIVF/CU	ra,rb,rc	0D	08
DIV/V	ra,rb,rc	0D	0A
DIVF/U	ra,rb,rc	0D	0C
MULG/C	ra,rb,rc	0E	00
MULG	ra,rb,rc	0E	04
MULG/CU	ra,rb,rc	0E	08
MULG/U	ra,rb,rc	0E	0C
MULF/C	ra,rb,rc	0F	00
MULL	ra,rb,rc	0F	02
UMULH	ra,rb,rc	0F	03
MULF	ra,rb,rc	0F	04
MULF/CU	ra,rb,rc	0F	08
MULL/V	ra,rb,rc	0F	0A
MULF/U	ra,rb,rc	0F	0C
VMERGE	ra,vb,vc	10	00
IOTA	ra,vc	10	01
VMERGE	va,vb,vc	10	10
VADD	ra,vb,vc	11	00
VSUB	ra,vb,vc	11	01
VADD/V	ra,vb,vc	11	08
VSUB/V	ra,vb,vc	11	09
VADD	va,vb,vc	11	10
VSUB	va,vb,vc	11	11
VADD/V	va,vb,vc	11	18
VSUB/V	va,vb,vc	11	19
VCMPEQ	ra,vb	12	00
VCMPNE	ra,vb	12	01
VCMPGT	ra,vb	12	02
VCMPLE	ra,vb	12	03
VCMPGE	ra,vb	12	04
VCMPLT	ra,vb	12	05
VCMPUGT	ra,vb	12	0A
VCMPULE	ra,vb	12	0B
VCMPUGE	ra,vb	12	0C
VCMPULT	ra,vb	12	0D
VCMPEQ	va,vb	12	10
VCMPNE	va,vb	12	11
VCMPGT	va,vb	12	12
VCMPLE	va,vb	12	13
VCMPGE	va,vb	12	14
VCMPLT	va,vb	12	15
VCMPUGT	va,vb	12	1A
VCMPULE	va,vb	12	1B
VCMPUGE	va,vb	12	1C

VCMPULT	va,vb	12	1D
VAND	ra,vb,vc	13	00
VOR	ra,vb,vc	13	01
VXOR	ra,vb,vc	13	02
VSLL	ra,vb,vc	13	04
VSRL	ra,vb,vc	13	05
VBIC	ra,vb,vc	13	08
VORNOT	ra,vb,vc	13	09
VEQV	ra,vb,vc	13	0A
VAND	va,vb,vc	13	10
VOR	va,vb,vc	13	11
VXOR	va,vb,vc	13	12
VSLL	va,vb,vc	13	14
VSRL	va,vb,vc	13	15
VBIC	va,vb,vc	13	18
VORNOT	va,vb,vc	13	19
VEQV	va,vb,vc	13	1A
reserved		14	00
reserved		15	00
VCVTGL/C	vb,vc	16	10
VCVTGF/C	vb,vc	16	11
VCVTGL	vb,vc	16	14
VCVTGF	vb,vc	16	15
VCVTGF/CU	vb,vc	16	19
VCVTGF/U	vb,vc	16	1D
VCVTFL/C	vb,vc	17	10
VCVTLF/C	vb,vc	17	11
VCVTFG	vb,vc	17	12
VCVTLG	vb,vc	17	13
VCVTFL	vb,vc	17	14
VCVTLF	vb,vc	17	15
VADDG/C	ra,vb,vc	18	00
VSUBG/C	ra,vb,vc	18	01
VADDG	ra,vb,vc	18	04
VSUBG	ra,vb,vc	18	05
VADDG/CU	ra,vb,vc	18	08
VSUBG/CU	ra,vb,vc	18	09
VADDG/U	ra,vb,vc	18	0C
VSUBG/U	ra,vb,vc	18	0D
VADDG/C	va,vb,vc	18	10
VSUBG/C	va,vb,vc	18	11
VADDG	va,vb,vc	18	14
VSUBG	va,vb,vc	18	15
VADDG/CU	va,vb,vc	18	18
VSUBG/CU	va,vb,vc	18	19
VADDG/U	va,vb,vc	18	1C
VSUBG/U	va,vb,vc	18	1D
VADDF/C	ra,vb,vc	19	00
VSUBF/C	ra,vb,vc	19	01
VADDF	ra,vb,vc	19	04
VSUBF	ra,vb,vc	19	05
VADDF/CU	ra,vb,vc	19	08
VSUBF/CU	ra,vb,vc	19	09
VADDF/U	ra,vb,vc	19	0C
VSUBF/U	ra,vb,vc	19	0D
VADDF/C	va,vb,vc	19	10
VSUBF/C	va,vb,vc	19	11
VADDF	va,vb,vc	19	14
VSUBF	va,vb,vc	19	15
VADDF/CU	va,vb,vc	19	18

VSUBF/CU	va,vb,vc	19	19
VADDF/U	va,vb,vc	19	1C
VSUBF/U	va,vb,vc	19	1D
VCMPGEQ	ra,vb	1A	00
VCMPGNE	ra,vb	1A	01
VCMPGGT	ra,vb	1A	02
VCMPGLE	ra,vb	1A	03
VCMPGGE	ra,vb	1A	04
VCMPGLT	ra,vb	1A	05
VCMPGEQ	va,vb	1A	10
VCMPGNE	va,vb	1A	11
VCMPGGT	va,vb	1A	12
VCMPGLE	va,vb	1A	13
VCMPGGE	va,vb	1A	14
VCMPGLT	va,vb	1A	15
VCMPFEQ	ra,vb	1B	00
VCMPFNE	ra,vb	1B	01
VCMPFGT	ra,vb	1B	02
VCMPFLE	ra,vb	1B	03
VCMPFGE	ra,vb	1B	04
VCMPFLT	ra,vb	1B	05
VCMPFEQ	va,vb	1B	10
VCMPFNE	va,vb	1B	11
VCMPFGT	va,vb	1B	12
VCMPFLE	va,vb	1B	13
VCMPFGE	va,vb	1B	14
VCMPFLT	va,vb	1B	15
VDIVG/C	ra,vb,vc	1C	00
VDIVG	ra,vb,vc	1C	04
VDIVG/CU	ra,vb,vc	1C	08
VDIVG/U	ra,vb,vc	1C	0C
VDIVG/C	va,vb,vc	1C	10
VDIVG	va,vb,vc	1C	14
VDIVG/CU	va,vb,vc	1C	18
VDIVG/U	va,vb,vc	1C	1C
VDIVF/C	ra,vb,vc	1D	00
VDIVF	ra,vb,vc	1D	04
VDIVF/CU	ra,vb,vc	1D	08
VDIVF/U	ra,vb,vc	1D	0C
VDIVF/C	va,vb,vc	1D	10
VDIVF	va,vb,vc	1D	14
VDIVF/CU	va,vb,vc	1D	18
VDIVF/U	va,vb,vc	1D	1C
VMULG/C	ra,vb,vc	1E	00
VMULG	ra,vb,vc	1E	04
VMULG/CU	ra,vb,vc	1E	08
VMULG/U	ra,vb,vc	1E	0C
VMULG/C	va,vb,vc	1E	10
VMULG	va,vb,vc	1E	14
VMULG/CU	va,vb,vc	1E	18
VMULG/U	va,vb,vc	1E	1C
VMULF/C	ra,vb,vc	1F	00
VMULL	ra,vb,vc	1F	02
VUMULH	ra,vb,vc	1F	03
VMULF	ra,vb,vc	1F	04
VMULF/CU	ra,vb,vc	1F	08
VMULL/V	ra,vb,vc	1F	0A
VMULF/U	ra,vb,vc	1F	0C
VMULF/C	va,vb,vc	1F	10
VMULL	va,vb,vc	1F	12

VUMULH	va,vb,vc	1F	13
VMULF	va,vb,vc	1F	14
VMULF/CU	va,vb,vc	1F	18
VMULL/V	va,vb,vc	1F	1A
VMULF/U	va,vb,vc	1F	1C
BEQ	ra,dest	20	-
BNE	ra,dest	21	-
BGT	ra,dest	22	-
BLE	ra,dest	23	-
BGE	ra,dest	24	-
BLT	ra,dest	25	-
BLBC	ra,dest	26	-
BLBS	ra,dest	27	-
JSR	ra,dest	28	-
JSR	ra,(rb)	29	00
FLBC	ra	2A	-
DRAIN		2B	00
DRAINM		2B	10
DRAINV		2B	11
reserved		2C	00
EPIRES0		2D	-
EPIRES1		2E	-
EPIRES2		2F	-
VLDL	ra,rb,vc	30	00
VLDQ	ra,rb,vc	30	01
VLDL/W	ra,rb,vc	30	02
VLDQ/W	ra,rb,vc	30	03
VSTL	ra,rb,vc	30	04
VSTQ	ra,rb,vc	30	05
VGATHL	ra,vb,vc	31	00
VGATHQ	ra,vb,vc	31	01
VGATHL/W	ra,vb,vc	31	02
VGATHQ/W	ra,vb,vc	31	03
VSCATL	ra,vb,vc	31	04
VSCATQ	ra,vb,vc	31	05
RDVL	rc	32	00
RDVC	rc	32	01
RDVML	rc	32	02
RDVMH	rc	32	03
WRVL	ra	33	00
WRVC	ra	33	01
WRVML	ra	33	02
WRVMH	ra	33	03
reserved		34	-
reserved		35	-
EPIRES3		36	-
EPIRES4		37	-
LDB	d(rb),ra	38	-
LDW	d(rb),ra	39	-
LDL	d(rb),ra	3A	-
LDQ	d(rb),ra	3B	-
STB	ra,d(rb)	3C	-
STW	ra,d(rb)	3D	-
STL	ra,d(rb)	3E	-
STQ	ra,d(rb)	3F	-

Revision History:

Revision 3.0, 26 April 1988

1. Change opcode assignments for converts.
2. Change mnemonics to use opcode qualifiers.

Revision 2.0, 24 June 1986

1. Split MULx and DIVx into different opcodes.
2. Move LDA to keep MULG/MULF in even-odd pair.
3. Add in new DRAIN instructions.
4. Add in VCMPUxx.
5. Add in CYCCR IPR.
6. Change FOB to FLBC.
7. Remove the MULH, REM, and VREM instructions.
8. Remove the ESP, SSP, and TBIASN IPRs.

Revision 1.0, December 22, 1985

1. Total rework for 32 bit system.

Revision 0.0, July 5, 1985

1. First Review Distribution

APPENDIX B

PROGRAMMING HINTS

B.1 INTRODUCTION

This appendix documents programming techniques for using the PRISM instruction set efficiently to perform operations that are not supported directly by the PRISM architecture.

B.2 INTEGER DIVIDE

On a machine that does not implement integer divide, division can be performed by using G_floating divide as shown below.

r4 = dividend
r5 = divisor
r6 = quotient

Divide without overflow checking:

```
        add     #1, r5, r6      ;check divisor and
        beq     r6, 10$         ;branch if divisor is -1
        cvtlg   r4, r6          ;convert divisor to G_floating
        cvtlg   r5, r8          ;convert divisor to G_floating
        divg/c  r6, r8, r6      ;divide
        cvtgl/c r6, r6          ;convert to longword
        beq     r0, 20$         ;done
10$:    sub     r4, r0, r6      ;negate dividend
20$:
```

Divide with overflow checking enabled:

```
        add     #1, r5, r6      ;check divisor and
        beq     r6, 10$         ;branch if divisor is -1
        cvtlg   r4, r6          ;convert divisor to G_floating
        cvtlg   r5, r8          ;convert divisor to G_floating
        divg/c  r6, r8, r6      ;divide
        cvtgl/c r6, r6          ;convert to longword
        beq     r0, 20$         ;done
10$:    sub/v   r4, r0, r6      ;negate dividend
20$:
```

B.3 FAST INTEGER DIVIDE BY FIXED INTEGERS

This section presents a new algorithm for the exact quotient Q of the signed or unsigned integer division X/K. The algorithm uses one 32 bit approximate reciprocal, an unsigned multiply (UMULH), and a shift. This algorithm is faster than the general case described in the previous section. However, it requires a table containing an approximate reciprocal, and a shift amount.

A set of approximate reciprocals and shift amounts for K=2..255, 10, 10**2, 10**3, 10**6, and 10**9 is given. A simple recipe is given for generating other table entries.

B.3.1 THE ALGORITHM

Handling signed inputs:

The recipe: If the original inputs X and K are signed, then:

> if X and K have opposite signs, complement (2's) the negative one, and then complement the final result.
>
> if X and K are both negative, complement both before proceeding.

Note that the 2's complement of -2**31 yields the right (unsigned) value for the magnitude of that number so there is no need to detect and special case this particular X.

Notation and description of algorithm:

Let K >= 2 and 0<= X <= 2**32-1. Let

$$a = 1/K$$

Let A denote an approximation of a with 32 significant bits, such that

$$A >= a.$$

An approximation I to the desired final result, Q = floor(X/K), can be obtained by computing

```
I := floor(A*X);      (*)
```

The choice of A, shown below, will guarantee that I equals Q.

An exact computation of the quantity floor(A*X), using a UMULH and a shift, will be shown later.

First, some notation: Define

> S(a) = the number of leading zeros to the right of the binary point in the binary expansion of a=1/K. S(a) will be referred to as the "shift" S.
>
> Put N = 32 + S .

Let J be the smallest 32 bit (unsigned) integer satisfying

J*K >= 2**N, or J/2**N >= 1/K

Such a J can always be found and in fact an easy way to get J is to "round up" the appropriate bit field in the binary expansion of 1/K. See the example below.

Now define the approximation A by

A := J/2**N = (0.J)*2**(-S) (**)

where (0.J) denotes the binary fraction J/2**32. Clearly

A >= a .

Thus A satisfies A >= a = 1/K, and is the best one-sided approximation to 1/K of the given form (32 bit J, denominator 2**N)

The algorithm is simple: UMULH X by J, and right logical shift by S. It is clear from the right hand side of (**) that this computes I in equation (*) correctly. The theoretical results summarized in the next section simply state that I = Q.

The example below is intended to help clarify the notation.

Example 1:

K=10: a = 1/K = .000 1100 1100 1100 1100 1100 1100 1100 11... **(binary)**

S = 3; J = CCCCCCCD (hex)

Thus, with say X = FF (hex),

UMULH of J,X is 000000CC (hex),

and right shifting by S = 3 yields Q=00000019, which is correct.

In actual PRISM code (for K=10) the algorithm A looks like this:

I = floor(X*.CCCCCCCD*2**(-3)),

so, assuming X (positive, or else unsigned) is in R10:

```
INV10:  .LONG ^XCCCCCCCD

LDL     INV10,R11
UMULH   R10,R11,R11
SRL     #3,R11,R11
```

B.3.2 Analysis

This section simply states the main result, and gives an example to illustrate the sharpness of the inequality. The result is stated in terms of n significant bit approximants A to 1/K; n=32 for PRISM.

Theorem 1: Let a=1/K have the binary expansion

a = 0.(S leading zeros)d(1)d(2)...d(n)d(n+1)...

Let, the unsigned, n significant bit integer J, and the approximant A be defined by:

A = J/2**(n+S) = (ceiling(2**(n+S)*a))/2**(n+S)

That is, use n+S bits to define A, rounding up ("towards +infinity") if any of d(n+1)... are nonzero, so A >= a. (For algorithmic purposes, if d(1)...d(n) are all 1 and rounding up occurs, A is encoded by reducing S by 1 and putting d(1)=1, d(2)=...=d(n)=0.)

Then, for integer X with 0 <= X < 2**n,

floor(A*X) = floor(a*X), that is, I = Q.

Proof: Since Q = floor(a*X), there is an integer t such that

a*X = Q + t/K, with 0 <= t <= K-1

Also, I = floor(A*X) can be written as follows:

I = floor(a*X + (A-a)*X), or,

I= floor(Q + t/K + (A-a)*X)

Thus, I equals Q so long as

(A-a)*X < 1/K, or,

X < (1/K)/(A-a). (***)

Define the modified remainder R by

R = J*K - 2**(n+S).

Now, can be rewritten (A-a) as

(A-a) = J/2**(n+S) - 1/K = R/(K*2**(n+S)).

If R=0, then A=a, and I=Q is trivially satisfied for any X. Also, by the definition of A and J,

0 <= R <= K-1.

Substituting in (***), it can be concluded that have I equals Q whenever

X < 2**(n+S)/R (****),

and the smallest (worst case) upper bound provided by (****) is

X < 2**(n+S)/(K-1).

To complete the proof, it will be shown that

2**S/(K-1) <= 1 :

For this last inequality, note that any K >= 2, for which 1/K has S leading zeros, satisfies

$$2^{**}S + 1 <= K <= 2^{**}(S+1).$$

The left hand inequality shows that

$2^{**}S <= K - 1.$

This completes the proof of Theorem 1.

The following example shows that in general the upper bound on X above cannot be improved upon. Let K=5 and take n=6. Thus, A=2**(-2)*(.110100). With X=63, I equals Q as it should. However, X=64=2**n yields I=13, whereas Q=64/5=12.

For fixed n and a specific K, the upper bound on X for which I=Q can be 2**n or more, provided the the modified remainder R is small. However, for arbitrary K, the bound in theorem 1 above has been shown to be sharp.

To summarize: taking n=32 in theorem 1, the algorithm works for the full range of 32 bit X, 0 <= X < 2**32.

B.3.3 Table For Some Powers Of 10:

Perhaps the only special cases of obvious importance are K=10, 10**2, 10**3, 10**6, and 10**9. Appropriate tables are given below.

Table for some Powers of 10 (J in hex; K, S in decimal):

K	J	S
10**1	CCCCCCCD	3
10**2	51EB851F	5
10**3	10624DD3	6
10**6	431BDE83	12
10**9	44B82FA1	28

Examples:

With K=10**6, X=7FFFFFFF (hex), I = 863 (hex) = 2147 = Q. With K=10**6, X=FFFFFFFF (hex), I = 10C6 (hex) = 4294 = Q.

B.3.4 Table For Numbers Between 2 And 255

The table is for K=2..255. The entries for K a power of 2 are for completeness only (one shift and no UMULH will do). All entries are in hex. The remainder R is J*K-2**(32+S). Only J and S are actually needed to carry out the algorithm.

K	J	S	R
00000002	80000000	00000000	00000000
00000003	AAAAAAAB	00000001	00000001
00000004	40000000	00000000	00000000
00000005	CCCCCCCD	00000002	00000001
00000006	AAAAAAAB	00000002	00000002
00000007	92492493	00000002	00000005
00000008	20000000	00000000	00000000
00000009	38E38E39	00000001	00000004
0000000A	CCCCCCCD	00000003	00000002
0000000B	BA2E8BA3	00000003	00000001
0000000C	AAAAAAAB	00000003	00000004
0000000D	4EC4EC4F	00000002	00000006
0000000E	92492493	00000003	0000000A
0000000F	88888889	00000003	00000007
00000010	10000000	00000000	00000000
00000011	F0F0F0F1	00000004	00000001
00000012	38E38E39	00000002	00000008
00000013	6BCA1AF3	00000003	00000012
00000014	CCCCCCCD	00000004	00000004
00000015	30C30C31	00000002	00000014
00000016	BA2E8BA3	00000004	00000002
00000017	B21642C9	00000004	0000000F
00000018	AAAAAAAB	00000004	00000008
00000019	51EB851F	00000003	0000000E
0000001A	4EC4EC4F	00000003	0000000C
0000001B	4BDA12F7	00000003	0000001A
0000001C	92492493	00000004	00000014
0000001D	8D3DCB09	00000004	00000005
0000001E	88888889	00000004	0000000E
0000001F	84210843	00000004	0000001D
00000020	08000000	00000000	00000000
00000021	3E0F83E1	00000003	00000004
00000022	F0F0F0F1	00000005	00000002
00000023	EA0EA0EB	00000005	00000021
00000024	38E38E39	00000003	00000010
00000025	DD67C8A7	00000005	00000023
00000026	6BCA1AF3	00000004	00000024
00000027	D20D20D3	00000005	00000025
00000028	CCCCCCCD	00000005	00000008
00000029	C7CE0C7D	00000005	00000005
0000002A	30C30C31	00000003	00000028
0000002B	2FA0BE83	00000003	00000004
0000002C	BA2E8BA3	00000005	00000004
0000002D	B60B60B7	00000005	0000002B
0000002E	B21642C9	00000005	0000001E
0000002F	AE4C415D	00000005	00000013
00000030	AAAAAAAB	00000005	00000010
00000031	5397829D	00000004	0000001A
00000032	51EB851F	00000004	0000001C
00000033	A0A0A0A1	00000005	00000013
00000034	4EC4EC4F	00000004	00000018
00000035	4D4873ED	00000004	00000022
00000036	4BDA12F7	00000004	00000034
00000037	094F2095	00000001	00000030
00000038	92492493	00000005	00000028
00000039	8FB823EF	00000005	00000037
0000003A	8D3DCB09	00000005	0000000A
0000003B	22B63CBF	00000003	00000014
0000003C	88888889	00000005	0000001C

```
0000003D 4325C53F 00000004 00000006
0000003E 84210843 00000005 0000003A
0000003F 82082083 00000005 0000003D
00000040 04000000 00000000 00000000
00000041 FC0FC0FD 00000006 0000003D
00000042 3E0F83E1 00000004 00000008
00000043 07A44C6B 00000001 00000020
00000044 F0F0F0F1 00000006 00000004
00000045 76B981DB 00000005 0000000E
00000046 EA0EA0EB 00000006 00000042
00000047 E6C2B449 00000006 0000003F
00000048 38E38E39 00000004 00000020
00000049 E070381D 00000006 00000045
0000004A DD67C8A7 00000006 00000046
0000004B 1B4E81B5 00000003 00000038
0000004C 6BCA1AF3 00000005 00000048
0000004D 3531DEC1 00000004 00000034
0000004E D20D20D3 00000006 0000004A
0000004F CF6474A9 00000006 00000027
00000050 CCCCCCCD 00000006 00000010
00000051 CA4587E7 00000006 00000017
00000052 C7CE0C7D 00000006 0000000A
00000053 3159721F 00000004 00000034
00000054 30C30C31 00000004 00000050
00000055 C0C0C0C1 00000006 00000015
00000056 2FA0BE83 00000004 00000008
00000057 2F149903 00000004 00000014
00000058 BA2E8BA3 00000006 00000008
00000059 B81702E1 00000006 00000039
0000005A B60B60B7 00000006 00000056
0000005B B40B40B5 00000006 00000057
0000005C B21642C9 00000006 0000003C
0000005D B02C0B03 00000006 00000017
0000005E AE4C415D 00000006 00000026
0000005F AC769185 00000006 0000005B
00000060 AAAAAAAB 00000006 00000020
00000061 151D07EB 00000003 00000058
00000062 5397829D 00000005 00000034
00000063 A57EB503 00000006 00000029
00000064 51EB851F 00000005 00000038
00000065 288DF0CB 00000004 0000005C
00000066 A0A0A0A1 00000006 00000026
00000067 13E22CBD 00000003 00000058
00000068 4EC4EC4F 00000005 00000030
00000069 9C09C09D 00000006 00000065
0000006A 4D4873ED 00000005 00000044
0000006B 4C8F8D29 00000005 00000046
0000006C 4BDA12F7 00000005 00000068
0000006D 964FDA6D 00000006 00000069
0000006E 094F2095 00000002 00000060
0000006F 939A85C5 00000006 0000006B
00000070 92492493 00000006 00000050
00000071 487EDE05 00000005 0000006A
00000072 8FB823EF 00000006 0000006E
00000073 473C1AB7 00000005 0000006A
00000074 8D3DCB09 00000006 00000014
00000075 8C08C08D 00000006 00000071
00000076 22B63CBF 00000004 00000028
00000077 44D72045 00000005 00000026
00000078 88888889 00000006 00000038
```

```
00000079 043B3D5B 00000001 00000060
0000007A 4325C53F 00000005 0000000C
0000007B 214D0215 00000004 0000005C
0000007C 84210843 00000006 00000074
0000007D 10624DD3 00000003 00000038
0000007E 82082083 00000006 0000007A
0000007F 81020409 00000006 00000077
00000080 02000000 00000000 00000000
00000081 0FE03F81 00000003 00000010
00000082 FC0FC0FD 00000007 0000007A
00000083 FA232CF3 00000007 00000059
00000084 3E0F83E1 00000005 00000010
00000085 F6603D99 00000007 0000007D
00000086 07A44C6B 00000002 00000040
00000087 F2B9D649 00000007 0000007F
00000088 F0F0F0F1 00000007 00000008
00000089 077975B9 00000002 00000020
0000008A 76B981DB 00000006 0000001C
0000008B 75DED953 00000006 00000022
0000008C EA0EA0EB 00000007 00000084
0000008D 3A196B1F 00000005 0000004C
0000008E E6C2B449 00000007 0000007E
0000008F E525982B 00000007 00000005
00000090 38E38E39 00000005 00000040
00000091 E1FC780F 00000007 0000007F
00000092 E070381D 00000007 0000008A
00000093 DEE95C4D 00000007 00000037
00000094 DD67C8A7 00000007 0000008C
00000095 DBEB61EF 00000007 0000001B
00000096 1B4E81B5 00000004 00000070
00000097 36406C81 00000005 0000005C
00000098 6BCA1AF3 00000006 00000090
00000099 D62B80D7 00000007 0000007F
0000009A 3531DEC1 00000005 00000068
0000009B D3680D37 00000007 0000004D
0000009C D20D20D3 00000007 00000094
0000009D 342DA7F3 00000005 0000001C
0000009E CF6474A9 00000007 0000004E
0000009F 19C2D14F 00000004 00000088
000000A0 CCCCCCCD 00000007 00000020
000000A1 CB8727C1 00000007 00000061
000000A2 CA4587E7 00000007 0000002E
000000A3 C907DA4F 00000007 0000004D
000000A4 C7CE0C7D 00000007 00000014
000000A5 634C0635 00000006 00000052
000000A6 3159721F 00000005 00000068
000000A7 621B97C3 00000006 0000006A
000000A8 30C30C31 00000005 000000A0
000000A9 60F25DEB 00000006 00000046
000000AA C0C0C0C1 00000007 0000002A
000000AB BFA02FE9 00000007 000000A3
000000AC 2FA0BE83 00000005 00000010
000000AD 17AD2209 00000004 000000A8
000000AE 2F149903 00000005 00000028
000000AF 5D9F7391 00000006 0000003E
000000B0 BA2E8BA3 00000007 00000010
000000B1 B92143FB 00000007 0000008B
000000B2 B81702E1 00000007 00000072
000000B3 B70FBB5B 00000007 000000A1
000000B4 B60B60B7 00000007 000000AC
```

```
000000F1 10FEF011 00000004 00000008
000000F2 043B3D5B 00000002 000000C0
000000F3 86D90545 00000007 0000007F
000000F4 4325C53F 00000006 00000018
000000F5 42DF9BB1 00000006 000000CA
000000F6 214D0215 00000005 000000B8
000000F7 84A9F9C9 00000007 000000EF
000000F8 84210843 00000007 000000E8
000000F9 83993053 00000007 000000BB
000000FA 10624DD3 00000004 00000070
000000FB 828CBFBF 00000007 00000045
000000FC 82082083 00000007 000000F4
000000FD 81848DA9 00000007 00000005
000000FE 81020409 00000007 000000EE
000000FF 80808081 00000007 0000007F
```

Revision History:

Revision 3.0, 26 April 1988.

1. Initial distribution.