PRISM System Reference Manual

<u>Second Draft</u>

Note

Revision No: 1.0		**Document Copy:**
Date: 22-DEC-1985		**104**

CONTENTS

PREFACE

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

These competitors are mostly targeting the high end of the VAX market, which is our most profitable product space. However, we are also receiving increasing pressure at the low end of our product family where simplified architectures offer cheaper and faster custom 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 summarizes 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 will be constructed that has a compatible file system, network, and user interface, and a functionally compatible set of system services. A continuing effort will be made to ensure that a compatible applications interface is maintained between VMS and the new operating system.

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. Architectural trade-offs will be made toward higher performance rather than lowest cost. However, competitive and cost effective chip-based implementations must be possible without having to resort to risky advanced technologies.

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 which 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 which 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 an I/O architecture that will support current and future corporate I/O strategies (e.g., BI).

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

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

13. 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.

14. To allow for the implementation of special purpose coprocessors.

15. 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 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.
6. Added a goal to define an I/O architecture that will support current, as well as future, corporate I/O strategies.

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-END VAX

It is currently very difficult to build a high-performance implementation (20 to 40 times 11/780) of the VAX architecture even though the circuit technology exists. VAX is an extremely complex architecture with a large number of intra-instruction and inter-instruction conflicts.

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

o 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.

o The side effects of autoincrement and autodecrement specifiers make pipelining, and the coordinated update of multiple register file copies, difficult.

o 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.

o 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.

o 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.

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

o 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.

o 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:

o 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.

- o 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:

- o The cost/performance and higher absolute performance advantages of simplified instruction set architectures.
- o Advances in compiler technology. In particular, the ability to compile procedures inline, better register allocation algorithms, and instruction scheduling.
- o A processor organization model that allows parallel instruction execution and out-of-order instruction completion.
- o The ability to implement both chip-level and high-end machines.
- o The declining cost of memory; memory costs in FY88 are expected to be around $150 per megabyte.

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

- o All instructions are 32 bits long and have a regular format.
- o 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 6-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. There is 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 is a brief outline of 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.

- o 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 have a fixed, non-data-independent execution time, except loads, which are predicted on the basis of cache hits.

- o The function unit used by the instruction during execution must be free. All units are pipelined (except for divide) and so can accept a new scalar instruction each machine cycle. A vector instruction reserves the function unit.

  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 issues, 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 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 via 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 liabilities 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 (and possibly vector 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. It is envisioned that the PRISM and VAX operating systems will support clustering of PRISM and VAX processors. It is also envisioned that the PRISM/VMS operating system will provide a VAX/VMS-compatible file system, DECNET, DCL, and functionally compatible system services, thus preserving the customer's VAX computing environment.

3. 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, however, 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.

1.5.2 Why No VAX Compatibility Mode Is Provided

No VAX compatibility mode is 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 cluster PRISM and VAX processors so customer applications on VAXs can share data with applications on PRISM. Customers will already own VAXs on which to run those applications that they don't wish to port to PRISM.

4. VAX memory management would be difficult to emulate without giving up the advantage of a larger page size.

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 are provide additional clarification and will be removed from externally distributed editions.\

Revision History:

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:

```
 7              0
+----------------+
|                | :A
+----------------+
```

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:

```
 1
 5                               0
+---------------------------------+
|                                 | :A
+---------------------------------+
```

Figure 2-2: Word Format

A word is specified by its address A. 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 in place 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 a 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:

```
 3
 1                                                              0
+---------------------------------------------------------------+
|                                                               | :A
+---------------------------------------------------------------+
```

Figure 2-3: Longword Format

A longword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a longword is a twos 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 the purposes of addition, subtraction, 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 on access to 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:

```
 3
 1                                                              0
+----------------------------------------------------------------+
|                                                                | :A
+----------------------------------------------------------------+
|                                                                | :A+4
+----------------------------------------------------------------+
 6                                                              3
 3                                                              2
```

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 on access to 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.

```
 1 1
 5 4             7 6           0
+-+---------------+-------------+
|S|      exp      |  fraction   | :A
+-+---------------+-------------+
|           fraction            | :A+2
+-------------------------------+
```

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. 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 on access to 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:

```
1 1
5 4                   4 3     0
+-+-------------------+-------+
|S|   exp             | fract | :A
+-+-------------------+-------+
|           fraction          | :A+2
+-----------------------------+
|           fraction          | :A+4
+-----------------------------+
|           fraction          | :A+6
+-----------------------------+
```

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. 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 on access to 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).

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

Revision History:

Revision 1.0, 22 December 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 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, each containing 64 elements numbered 0 through 63. Each element is 64 bits wide. A vector instruction that reads or writes longword or F_floating data reads bits <31:0> of each source element and writes bits <31:0> of each destination element. Bits <63:32> of the destination element are 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 6-bit Vector Length register (VL) controls how many vector elements are processed. VL is loaded prior to executing a vector instruction. Once loaded, VL specifies the length of all subsequent vector instructions until VL is loaded with a new value. When VL is zero, 64 elements are processed; otherwise VL elements are processed.

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 vector mask is used by the vector compare, merge, and IOTA instructions.

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.

3.2 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 (ones) 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 means a G_floating add while "+" applied to longword operands is an integer add. Similarly, "LS" 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 five PRISM instruction formats. They are:

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

All instruction formats are 32 bits long with a 6-bit major opcode field in bits <31:26> of the instruction. There are up to three 6-bit register fields, Ra, Rb, and Rc, in an instruction.

Each format is described below.

3.3.1 Memory Instruction Format

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

```
 3          2 2         2 1         1 1
 1          6 5         0 9         4 3                         0
+-----------+-----------+-----------+---------------------------+
|  opcode   |    Ra     |    Rb     |        Memory_disp        |
+-----------+-----------+-----------+---------------------------+
```

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.

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 PC relative subroutine jumps. It has the following format:

```
 3          2 2         2 1
 1          6 5         0 9                                     0
+-----------+-----------+---------------------------------------+
|  opcode   |    Ra     |              Branch_disp              |
+-----------+-----------+---------------------------------------+
```

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. 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 register-to-register operations. 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.

```
 3         2 2         2 1         1 1
 1         6 5         0 9         4 3       9 8 7 6 5          0
+-----------+-----------+-----------+---------+-+---+-----------+
|  opcode   |    Ra     |    Rb     |  func   |0|SBZ|    Rc     |
+-----------+-----------+-----------+---------+-+---+-----------+

+-----------+-----------+-----------+---------+-+---+-----------+
|  opcode   |    lit    |    Rb     |  func   |1|lit|    Rc     |
+-----------+-----------+-----------+---------+-+---+-----------+
```

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>)
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.

3.3.3.1 Convert Instructions

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

3.3.4 Epicode Instruction Format

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

```
3          2 2                       1 1
1          6 5                       4 3             6 5         0
+-----------+-----------------------+---------------+-----------+
|  opcode   |///////// SBZ //////////| Epicode func |/// SBZ ///|
+-----------+-----------------------+---------------+-----------+
```

Figure 3-4: 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.

\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, some 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 on Epicode for more details.\

3.3.5 Coprocessor Instruction Format

The Coprocessor format is used for reading and writing Coprocessor registers. It has the following format:

```
 3          2 2          2 1                  1
 1          6 5          0 9                  0 9 8                0
+-----------+-----------+-------------------+-+-----------------+
|  opcode   |    Ra     | Co-Proc Control   |T| Co-Proc Address |
+-----------+-----------+-------------------+-+-----------------+
```

Figure 3-5: Coprocessor Instruction Format

There is a 6-bit opcode field, a 6-bit Ra field, a 10-bit Coprocessor control field, a 1-bit trap enable field (T), and a 9-bit Coprocessor address field.

The Ra field on a Coprocessor Read or Write specifies a PRISM destination or source scalar register, respectively.

The Coprocessor control field is transmitted to the coprocessor to control the operation performed.

The Coprocessor address field selects a specific coprocessor in a system with multiple coprocessors.

The trap enable field (T) is used to enable exceptions on transactions with a coprocessor. See Chapter 4, Instruction Descriptions, Page 4-99.

The Coprocessor instruction format may be omitted in a subset implementation that does not provide a Coprocessor interface.

Revision History:

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
8. Coprocessor

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

- o The group name.
- o The format of each instruction in the group. This gives the name, access type, and data type of each instruction operand.
- o The operation of the instruction.
- o Exceptions specific to the instruction.
- o The mnemonic and name of each instruction in the group.

- o A description of the instruction operation.
- o 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 means that the instruction is not performed in either hardware or Epicode. System software may provide emulation routines for subsetted instructions. The following groups of instructions may be omitted as a group in a subset implementation. If one instruction in a group is provided then all other instructions in that group must be provided.

1. Integer Multiplication (MULV, MULL, MULH, UMULH)

2. Integer Division and Remainder (DIV, DIVV, REM)

3. Add F_floating (ADDF, ADDFZ, ADDFU, ADDFUZ)
   Subtract F_floating (SUBF, SUBFZ, SUBFU, SUBFUZ)
   Compare F_floating (CMPFEQ, CMPFNE, CMPFLT, CMPFLE, CMPFGT, CMPFGE)

4. Convert Longword Integer to F_floating (CVTLF, CVTLFZ)
   Convert F_floating to Longword Integer (CVTFL, CVTFLZ)

5. Convert F_floating to G_floating (CVTFG)
   Convert G_floating to F_floating (CVTGF, CVTGFZ, CVTGFU, CVTGFUZ)

6. Multiply F_floating (MULF, MULFZ, MULFU, MULFUZ)

7. Divide F_floating (DIVF, DIVFZ, DIVFU, DIVFUZ)

8. Add G_floating (ADDG, ADDGZ, ADDGU, ADDGUZ)
   Subtract G_floating (SUBG, SUBGZ, SUBGU, SUBGUZ)
   Compare G_floating (CMPGEQ, CMPGNE, CMPGLT, CMPGLE, CMPGGT, CMPGGE)

9. Convert Longword Integer to G_floating (CVTLG)
   Convert G_floating to Longword Integer (CVTGL, CVTGLZ)

10. Multiply G_floating (MULG, MULGZ, MULGU, MULGUZ)

11. Divide G_floating (DIVG, DIVGZ, DIVGU, DIVGUZ)

12. The vector instructions (including the instructions that read and write vector count (VC), vector length (VL), and vector mask (VM))

13. Coprocessor instructions (COPRD, COPWR)

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 is provided. One 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.

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

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

   - a - The operand is used in an address calculation to form an effective address. The data type code which 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).
   - i - The operand is an 8-bit immediate literal in the instruction.
   - r - The operand is read only.
   - w - The operand is write only.

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

   - b - Byte
   - f - F_floating
   - g - G_floating
   - l - Longword
   - q - Quadword
   - w - Word
   - 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.

4.2 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
LDA	Load Address
LDB	Load Zero Extended Byte
LDW	Load Zero Extended Word
LDL	Load Longword
LDQ	Load Quadword
RMAQI	Read, Mask, Add Quadword, Interlocked
STB	Store Byte
STW	Store Word
STL	Store Longword
STQ	Store Quadword
VLDL	Vector Load Longword
VLDQ	Vector Load Quadword
VGATHL	Vector Gather Longword
VGATHQ	Vector Gather Quadword
VSTL	Vector Store Longword
VSTQ	Vector Store Quadword
VSCATQ	Vector Scatter Quadword
VSCATL	Vector Scatter Longword

Load Address

Format:

```
        LDA     disp.ab(Rb.ab),Ra.wl            !Memory format
```

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:

```
        LD      disp.ab(Rb.ab),Ra.wx            !Memory format
```

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 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 emit the multi-instruction sequence inline to fetch the operand in pieces rather than incur the emulation overhead.

Read, Mask, Add Quadword Interlocked

Format:

```
        RMAQI                                        !Epicode format
```

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 in a multiprocessor system 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:

```
        ST      Ra.rx,disp.ab(Rb.ab)            !Memory format
```

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 emit the multi-instruction sequence inline to store the operand in pieces rather than incur the emulation overhead.

Load Memory Data into Vector Register

Format:

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

Operation:

```
        va <- Rbv
        FOR i <- 0 TO VL-1
            BEGIN
            IF {va unaligned} THEN
                {Vector Alignment Exception}

            Vc[i] <- (va)<31:0>           !VLDL
            Vc[i] <- (va)<63:0>           !VLDQ
            va <- va + Rav                !Increment by stride
            END
```

Exceptions:

Access Violation
Fault On Read
Translation Not Valid
Vector Alignment

Opcodes:

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

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 can be either positive or negative.

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.

An implementation may allow multiple vector streams or scalar and vector streams to proceed concurrently on the same processor. It is the responsibility of software to determine when read/write memory data conflicts might produce incorrect results and insert DRAIN instructions to ensure correct operation.

These instructions may be omitted in a subset implementation.

Gather Memory Data into Vector Register

Format:

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

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            va <- Rav + Vb[i]<31:0>
            IF {va unaligned} THEN
                {Vector Alignment exception}

            Vc[i] <- (va)<31:0>          !VGATHL
            Vc[i] <- (va)<63:0>          !VGATHQ
            END
```

Exceptions:

Access Violation
Fault On Read
Translation Not Valid
Vector Alignment

Opcodes:

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

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 longword element offsets in vector register Vb. The address of element i (0 LE i LE VL-1) is computed as {Rav + Vb[i]}. The longword element offset can be either positive or negative.

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.

An implementation may allow multiple vector streams or scalar and vector streams to proceed concurrently on the same processor. It is the responsibility of software to determine when read/write memory data conflicts might produce incorrect results and insert DRAIN instructions to ensure correct operation.

These instructions may be omitted in a subset implementation.

Note:

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

Store Vector Register Data into Memory

Format:

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

Operation:

```
        va <- Rbv

        FOR i <- 0 TO VL-1
            BEGIN
            IF {va unaligned} THEN
                {Vector Alignment exception}

            (va) <- Vc[i]<31:0>                        !VSTL
            (va) <- Vc[i]                              !VSTQ
            va <- va + Rav                             !Increment by stride
            END
```

Exceptions:

Access Violation
Fault On Write
Translation Not Valid
Vector Alignment

Opcodes:

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

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 can be either positive or negative.

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

An implementation may allow multiple vector streams or scalar and vector streams to proceed concurrently on the same processor. It is the responsibility of software to determine when read/write memory data conflicts might produce incorrect results and insert DRAIN instructions to ensure correct operation.

The order in which the elements are stored is UNPREDICTABLE.

These instructions may be omitted in a subset implementation.

Scatter Vector Register Data into Memory

Format:

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

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            va <- Rav + Vb[i]<31:0>
            IF {va unaligned} THEN
                {Vector Alignment exception}

            (va) <- Vc[i]<31:0>                        !VSCATL
            (va) <- Vc[i]                              !VSCATQ
            END
```

Exceptions:

Access Violation
Fault On Write
Translation Not Valid
Vector Alignment

Opcodes:

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

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 longword element offsets in vector register Vb. The address of element i (0 LE i LE VL-1) is computed as {Rav + Vb[i]}. The longword element offset can be either positive or negative.

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

An implementation may allow multiple vector streams or scalar and vector streams to proceed concurrently on the same processor. It is the responsibility of software to determine when read/write memory data conflicts might produce incorrect results and insert DRAIN instructions to ensure correct operation.

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.

4.3 INTEGER ARITHMETIC INSTRUCTIONS

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

The integer instructions are summarized below:

Mnemonic	Operation
ADD	Add Longword with no Overflow Detect
ADDV	Add Longword with Overflow Detect
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 with no Overflow Detect
DIVV	Divide Longword with Overflow Detect
REM	Longword Remainder
MULV	Multiply Longword with Overflow Detect
MULL	Multiply Longword and Return Low 32 Product Bits
MULH	Multiply Longword and Return High 32 Product Bits
UMULH	Unsigned Multiply Longword and Return High 32 Product Bits
SUB	Subtract Longword with no Overflow Detect
SUBV	Subtract Longword with Overflow Detect

Mnemonic	Operation
VADD	Vector Add Longword with no Overflow Detect
VADDV	Vector Add Longword with Overflow Detect
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
VDIV	Vector Divide Longword with no Overflow Detect
VDIVV	Vector Divide Longword with Overflow Detect
VREM	Vector Longword Remainder
VMULL	Vector Multiply Longword and Return Low 32 Product Bits
VMULV	Vector Multiply Longword with Overflow Detect
VSUB	Vector Subtract Longword with no Overflow Detect
VSUBV	Vector Subtract Longword with Overflow Detect

Integer Add

Format:

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

Operation:

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

Exceptions:

Integer Overflow

Opcodes:

ADD	Add Integer with no Overflow Detect
ADDV	Add Integer with Longword Overflow Detect

Description:

Register Ra or a literal is added to register Rb and the 32-bit sum is written to register Rc. If integer overflow is detected, an Integer Overflow exception occurs.

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

Integer Signed Compare

Format:

```
        CMP     Ra.rl,Rb.rl,Rc.wl                   !Operate format
        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, the value one is written to register Rc; otherwise, zero is written to Rc.

Integer Unsigned Compare

Format:

```
        CMP     Ra.rl,Rb.rl,Rc.wl                   !Operate format
        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, the value one is written to register Rc; otherwise, zero is written to Rc.

Integer Divide

Format:

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

Operation:

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

Exceptions:

Integer Divide by Zero
Integer Overflow

Opcodes:

DIV	Divide Longword with no Overflow Detect
DIVV	Divide Longword with Overflow Detect

Description:

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

DIV suppresses the detection of integer overflow. The quotient result with a zero divisor is UNPREDICTABLE.

These instructions may be omitted in a subset implementation.

Integer Remainder

Format:

```
        REM   Ra.rl,Rb.rl,Rc.wl                             !Operate format
        REM   #a.ib,Rb.rl,Rc.wl
```

Operation:

```
        Rc <- REM(Rbv, Rav)
```

Exceptions:

Integer Divide by Zero

Opcodes:

REM	Longword Integer Remainder

Description:

Register Rb is divided by register Ra or a literal and the remainder is written to register Rc. The remainder is calculated such that it has the same sign as the dividend operand.

The REM result is UNPREDICTABLE when the divisor is zero.

This instruction may be omitted in a subset implementation.

Integer Multiply

Format:

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

Operation:

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

Exceptions:

Integer Overflow

Opcodes:

MULV	Multiply Longword with Overflow Detect

The following instructions do not detect overflow:

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

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 MULV, MULL, and MULH, and unsigned for UMULH.

MULV writes the least significant 32 product bits with overflow detection. If integer overflow is detected, an Integer Overflow exception occurs.

MULL writes the least significant 32 product bits with no overflow detection.

MULH and UMULH write the most significant 32 product bits.

These instructions may be omitted in a subset implementation.

Integer Subtract

Format:

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

Operation:

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

Exceptions:

Integer Overflow

Opcodes:

SUB	Subtract Longword with no Overflow Detect
SUBV	Subtract Longword with Overflow Detect

Description:

Register Ra or a literal is subtracted from register Rb and the 32-bit difference is written to register Rc. If integer overflow is detected, an Integer Overflow exception occurs.

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

Vector Integer Add

Format:

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

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- Va[i]<31:0> + Vb[i]<31:0>      !Vector + Vector
            Vc[i] <- Rav + Vb[i]<31:0>              !Scalar + Vector
            END
```

Exceptions:

Integer Overflow

Opcodes:

VADD	Vector Add Longword with no Overflow Detect
VADDV	Vector Add Longword with Overflow Detect

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 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.

These instructions may be omitted in a subset implementation.

Vector Integer Compare

Format:

```
        VCMP    Va.rl,Vb.rl                               !Operate format
        VCMP    Ra.rl,Vb.rl
        VCMP    #a.ib,Vb.rl
```

Operation:

```
        VM <- 0
        FOR i <- 0 TO VL-1
            BEGIN
                                                          !Vector cmp Vector
            IF  Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0>  THEN
                VM<i> <- 1
                                                          !Scalar cmp Vector
            IF  Rav SIGNED_RELATION Vb[i]<31:0>  THEN
                VM<i> <- 1
            END
```

Exceptions:

None

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

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. The Vector Mask register (VM) is cleared at the start of the operation. For each element comparison, if the specified relationship is true, the Vector Mask bit (VM<i>) corresponding to the vector element is set to 1. Only bits <31:0> of each vector element participate in the operation.

These instructions may be omitted in a subset implementation.

Vector Integer Divide

Format:

```
        VDIV    Va.rl,Vb.rl,Vc.wl                       !Operate format
        VDIV    Ra.rl,Vb.rl,Vc.wl
        VDIV    #a.ib,Vb.rl,Vc.wl
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- Vb[i]<31:0> / Va[i]<31:0>      !Vector / Vector
            Vc[i] <- Vb[i]<31:0> / Rav              !Vector / Scalar
            END
```

Exceptions:

Integer Divide by Zero
Integer Overflow

Opcodes:

VDIV	Vector Divide Longword with no Overflow Detect
VDIVV	Vector Divide Longword with Overflow Detect

Description:

Vector register Vb is divided, element-wise, by a vector operand (in register Va) or a scalar operand (in register Ra or a literal) and the 32-bit quotient is written to vector register Vc. Only bits <31:0> of each vector element participate in the 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 or integer divide by zero is detected, an Integer Overflow or Integer Divide By Zero exception (possibly both) occurs when the vector operation completes. The quotient result with a zero divisor is UNPREDICTABLE.

These instructions may be omitted in a subset implementation.

Vector Integer Remainder

Format:

```
        VREM    Va.rl,Vb.rl,Vc.wl                      !Operate format
        VREM    Ra.rl,Vb.rl,Vc.wl
        VREM    #a.ib,Vb.rl,Vc.wl
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- REM(Vb[i]<31:0>, Va[i]<31:0>)  !Vector REM Vector
            Vc[i] <- REM(Vb[i]<31:0>, Rav)          !Vector REM Scalar
            END
```

Exceptions:

Integer Divide by Zero

Opcodes:

VREM	Vector Longword Remainder

Description:

Vector register Vb is divided, element-wise, by a vector operand (in register Va) or a scalar operand (in register Ra or a literal) and the 32-bit remainder is written to vector register Vc. The remainder is calculated such that it has the same sign as the dividend operand. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE. The length of the vector is specified by the VL register.

If integer divide by zero is detected, an Integer Divide By Zero exception occurs when the vector operation completes. The remainder result with a zero divisor is UNPREDICTABLE.

This instruction may be omitted in a subset implementation.

Vector Integer Multiply

Format:

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

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN                                   !Vector * Vector
            Vc[i] <- {Va[i]<31:0> * Vb[i]<31:0>}<31:0>

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

Exceptions:

Integer Overflow

Opcodes:

VMULL	Vector Multiply Longword with no Overflow Detect
VMULV	Vector Multiply Longword with Overflow Detect

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 significant 32 bits of the signed 64-bit product are written to vector register Vc. Only bits <31:0> of each vector element participate in the 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.

These instructions may be omitted in a subset implementation.

Vector Integer Subtract

Format:

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

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- Vb[i]<31:0> - Va[i]<31:0>  !Vector - Vector
            Vc[i] <- Vb[i]<31:0> - Rav          !Vector - Scalar
            END
```

Exceptions:

Integer Overflow

Opcodes:

VSUB	Vector Subtract Longword with no Overflow Detect
VSUBV	Vector Subtract Longword with Overflow Detect

Description:

A vector operand (in register Va) or a scalar operand (in register Ra or a literal) is subtracted, element-wise, from a vector operand (in register Vb). The 32-bit difference is written to vector register Vc. Only bits <31:0> of each vector element participate in the 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.

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 shifts because this requires three source register operands (two for data, one for count). Quadword shift returning a longword can be done with a three instruction sequence (SLL, SRL, OR).\

Logical Functions

Format:

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

Operation:

```
        dst <- Rav AND Rbv              !AND
        dst <- Rav OR  Rbv              !OR
        dst <- Rav XOR Rbv              !XOR
        dst <- {NOT Rav} AND Rbv        !BIC
        dst <- {NOT Rav} OR  Rbv        !ORNOT
        dst <- {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:

```
        opcode   Ra.rb,Rb.rl,Rc.wl                  !Operate format
        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:

```
        SRA      Ra.rb,Rb.rl,Rc.wl                    !Operate format
        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:

```
        ROT      Ra.rb,Rb.rl,Rc.wl                  !Operate format
        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:

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

Operation:

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

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

Exceptions:

None

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

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 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.

Vector Merge

Format:

```
        VMERGE   Va.rq,Vb.rq,Vc.wq                    !Operate format
        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 0  THEN                      !Vector op Vector
                Vc[i] <- Va[i]
            ELSE
                Vc[i] <- Vb[i]

            IF  VM<i> EQ 0  THEN                      !Scalar op Vector
                Vc[i] <- QRav
            ELSE
                Vc[i] <- Vb[i]
            END
```

Exceptions:

None

Opcodes:

VMERGE Vector Merge

Description:

A vector operand (in register Va) or a scalar operand (in register QRa 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>) is zero, Va[i] or Qrav is written to the destination vector element Vc[i]. If VM<i> is one, Vb[i] is written to the destination vector element.

Software Note:

VMERGE can be used to load a vector register with a constant or to copy a vector register.

This instruction may be omitted in a subset implementation.

Vector Shift Logical

Format:

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

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            ! shift vector by vector
            Vc[i] <- LEFT_SHIFT(Vb[i]<31:0>,  Va[i]<4:0>) !SLL
            Vc[i] <- RIGHT_SHIFT(Vb[i]<31:0>, Va[i]<4:0>) !SRL

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

Exceptions:

None

Opcodes:

VSLL	Vector Shift Left Logical
VSRL	Vector Shift Right Logical

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 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 Underflow Disabled VAX Rounding
ADDFZ	Add F_floating Underflow Disabled Round toward Zero
ADDFU	Add F_floating Underflow Enabled VAX Rounding
ADDFUZ	Add F_floating Underflow Enabled Round toward Zero
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 Integer to F_floating VAX Rounding
CVTLFZ	Convert Longword Integer to F_floating Round toward Zero
CVTFL	Convert F_floating to Longword Integer VAX Rounding
CVTFLZ	Convert F_floating to Longword Integer Round toward Zero
CVTFG	Convert F_floating to G_floating
DIVF	Divide F_floating Underflow Disabled VAX Rounding
DIVFZ	Divide F_floating Underflow Disabled Round toward Zero
DIVFU	Divide F_floating Underflow Enabled VAX Rounding
DIVFUZ	Divide F_floating Underflow Enabled Round toward Zero
MULF	Multiply F_floating Underflow Disabled VAX Rounding
MULFZ	Multiply F_floating Underflow Disabled Round toward Zero
MULFU	Multiply F_floating Underflow Enabled VAX Rounding
MULFUZ	Multiply F_floating Underflow Enabled Round toward Zero
SUBF	Subtract F_floating Underflow Disabled VAX Rounding
SUBFZ	Subtract F_floating Underflow Disabled Round toward Zero
SUBFU	Subtract F_floating Underflow Enabled VAX Rounding
SUBFUZ	Subtract F_floating Underflow Enabled Round toward Zero

Mnemonic	Operation
ADDG	Add G_floating Underflow Disabled VAX Rounding
ADDGZ	Add G_floating Underflow Disabled Round toward Zero
ADDGU	Add G_floating Underflow Enabled VAX Rounding
ADDGUZ	Add G_floating Underflow Enabled Round toward Zero
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 Nounderflow VAX Rounding
CVTGFZ	Convert G_ to F_floating Nounderflow Round toward Zero
CVTGFU	Convert G_ to F_floating Underflow Enabled VAX Rounding
CVTGFUZ	Convert G_ to F_floating Underflow Enabled Round toward Zero
CVTLG	Convert Longword Integer to G_floating
CVTGL	Convert G_floating to Longword Integer VAX Rounding
CVTGLZ	Convert G_floating to Longword Integer Round toward Zero
DIVG	Divide G_floating Underflow Disabled VAX Rounding
DIVGZ	Divide G_floating Underflow Disabled Round toward Zero
DIVGU	Divide G_floating Underflow Enabled VAX Rounding
DIVGUZ	Divide G_floating Underflow Enabled Round toward Zero
MULG	Multiply G_floating Underflow Disabled VAX Rounding
MULGZ	Multiply G_floating Underflow Disabled Round toward Zero
MULGU	Multiply G_floating Underflow Enabled VAX Rounding
MULGUZ	Multiply G_floating Underflow Enabled Round toward Zero
SUBG	Subtract G_floating Underflow Disabled VAX Rounding
SUBGZ	Subtract G_floating Underflow Disabled Round toward Zero
SUBGU	Subtract G_floating Underflow Enabled VAX Rounding
SUBGUZ	Subtract G_floating Underflow Enabled Round toward Zero

Mnemonic	Operation
VADDF	Vector Add F_floating Underflow Disabled VAX Rounding
VADDFZ	Vector Add F_floating Underflow Disabled Round toward Zero
VADDFU	Vector Add F_floating Underflow Enabled VAX Rounding
VADDFUZ	Vector Add F_floating Underflow Enabled Round toward Zero
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 Integer to F_floating VAX Rounding
VCVTLFZ	Vector Convert Longword Integer to F_floating Round toward Zero
VCVTFL	Vector Convert F_floating to Longword Integer VAX Rounding
VCVTFLZ	Vector Convert F_floating to Longword Integer Round toward Zero
VCVTFG	Vector Convert F_floating to G_floating
VDIVF	Vector Divide F_floating Underflow Disabled VAX Rounding
VDIVFZ	Vector Divide F_floating Underflow Disabled Round toward Zero
VDIVFU	Vector Divide F_floating Underflow Enabled VAX Rounding
VDIVFUZ	Vector Divide F_floating Underflow Enabled Round toward Zero
VMULF	Vector Multiply F_floating Underflow Disabled VAX Rounding
VMULFZ	Vector Multiply F_floating Underflow Disabled Round toward Zero
VMULFU	Vector Multiply F_floating Underflow Enabled VAX Rounding
VMULFUZ	Vector Multiply F_floating Underflow Enabled Round toward Zero
VSUBF	Vector Subtract F_floating Underflow Disabled VAX Rounding
VSUBFZ	Vector Subtract F_floating Underflow Disabled Round toward Zero
VSUBFU	Vector Subtract F_floating Underflow Enabled VAX Rounding
VSUBFUZ	Vector Subtract F_floating Underflow Enabled Round toward Zero

Mnemonic	Operation
VADDG	Vector Add G_floating Underflow Disabled VAX Rounding
VADDGZ	Vector Add G_floating Underflow Disabled Round toward Zero
VADDGU	Vector Add G_floating Underflow Enabled VAX Rounding
VADDGUZ	Vector Add G_floating Underflow Enabled Round toward Zero
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 No underflow VAX Rounding
VCVTGFZ	Vector Convert G_ to F_floating No underflow Round toward Zero
VCVTGFU	Vector Convert G_ to F_floating Underflow Enabled VAX Rounding
VCVTGFUZ	Vector Convert G_ to F_floating Underflow Enabled Round toward Zero
VCVTLG	Vector Convert Longword Integer to G_floating
VCVTGL	Vector Convert G_floating to Longword Integer VAX Rounding
VCVTGLZ	Vector Convert G_floating to Longword Integer Round toward Zero
VDIVG	Vector Divide G_floating Underflow Disabled VAX Rounding
VDIVGZ	Vector Divide G_floating Underflow Disabled Round toward Zero
VDIVGU	Vector Divide G_floating Underflow Enabled VAX Rounding
VDIVGUZ	Vector Divide G_floating Underflow Enabled Round toward Zero
VMULG	Vector Multiply G_floating Underflow Disabled VAX Rounding
VMULGZ	Vector Multiply G_floating Underflow Disabled Round toward Zero
VMULGU	Vector Multiply G_floating Underflow Enabled VAX Rounding
VMULGUZ	Vector Multiply G_floating Underflow Enabled Round toward Zero
VSUBG	Vector Subtract G_floating Underflow Disabled VAX Rounding
VSUBGZ	Vector Subtract G_floating Underflow Disabled Round toward Zero
VSUBGU	Vector Subtract G_floating Underflow Enabled VAX Rounding
VSUBGUZ	Vector Subtract G_floating Underflow Enabled Round toward Zero

4.5.1 Literals

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

4.5.2 Accuracy

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

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 a re-normalization may be required after rounding takes place. The following statements summarize the relations among chopped, rounded, and true (infinite-precision) results:

- o If a stored result is exact

  rounded value = chopped value = true value.

- o 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.

- o 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 ones complement if any "ones" were lost, and by twos complement if none were lost. If this is done, the error bound will be 1/2 LSB.

o 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.

o 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 on PRISM (see Chapter 6, Exceptions and Interrupts, Section 6.4.1). The floating-point operation completes by writing a reserved operand with the exception type encoded in it. The figure below illustrates this:

```
 1 1
 5 4            7 6   4 3    0
+-+--------------+-----+-------+
|1|      0       |xxxxx| ETYPE | :A
+-+--------------+-----+-------+
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| :A+2
+------------------------------+

 1 1
 5 4                  4 3    0
+-+--------------------+-------+
|1|         0          | ETYPE | :A
+-+--------------------+-------+
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| :A+2
+------------------------------+
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| :A+4
+------------------------------+
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| :A+6
+------------------------------+
```

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, multiple bits may be set in the ETYPE field.

The state of all other bits in the result (denoted with 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 is always enabled.

Floating Add

Format:

```
        ADD     Ra.rx,Rb.rx,Rc.wx                      !Operate format
```

Operation:

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

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

ADDF	Add F_Floating VAX Rounding
ADDFZ	Add F_Floating Round toward Zero
ADDG	Add G_Floating VAX Rounding
ADDGZ	Add G_Floating Round toward Zero

The following instructions enable the Floating Underflow exception:

ADDFU	Add F_floating VAX Rounding
ADDFUZ	Add F_floating Round toward Zero
ADDGU	Add G_floating VAX Rounding
ADDGUZ	Add G_floating Round toward Zero

Description:

Register Ra is added to register Rb and the sum is 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.

Floating Compare

Format:

```
        CMP     Ra.rx,Rb.rx,Rc.wl                       !Operate format
```

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 are compared. If the specified relationship is true, the value one is written to register Rc; otherwise, zero is written to Rc.

These instructions may be omitted in a subset implementation.

Convert F_Floating to G_Floating

Format:

```
        CVT     Ra.rf,Rc.wg                    !Operate format
```

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 Rc. 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:

```
        CVT     Ra.rg,Rc.wf                          !Operate format
```

Operation:

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

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

CVTGF	Convert G_floating to F_floating VAX Rounding
CVTGFZ	Convert G_floating to F_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

CVTGFU	Convert G_floating to F_floating VAX Rounding
CVTGFUZ	Convert G_floating to F_floating Round toward Zero

Description:

The G_floating source operand in register Ra 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 Integer

Format:

```
        CVT     Ra.rx,Rc.wl                             !Operate format
```

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 VAX Rounding
CVTFLZ	Convert F_floating to Longword Round toward Zero
CVTGL	Convert G_floating to Longword VAX Rounding
CVTGLZ	Convert G_floating to Longword Round toward Zero

Description:

The F_ or G_floating source operand in register Ra is converted to a longword integer and written to register Rc.

These instructions may be omitted in a subset implementation.

Convert Integer to Floating

Format:

```
        CVT     Ra.rl,Rc.wx                          !Operate format
        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 VAX Rounding
CVTLFZ	Convert Longword to F_floating Round toward Zero
CVTLG	Convert Longword to G_floating

Description:

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

These instructions may be omitted in a subset implementation.

Floating Divide

Format:

```
        DIV     Ra.rx,Rb.rx,Rc.wx                      !Operate format
```

Operation:

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

Exceptions:

Floating Divide by Zero
Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

DIVF	Divide F_floating VAX Rounding
DIVFZ	Divide F_floating Round toward Zero
DIVG	Divide G_floating VAX Rounding
DIVGZ	Divide G_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

DIVFU	Divide F_floating VAX Rounding
DIVFUZ	Divide F_floating Round toward Zero
DIVGU	Divide G_floating VAX Rounding
DIVGUZ	Divide G_floating Round toward Zero

Description:

The dividend in register Rb is divided by the divisor in register Ra, and the quotient is 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.

Floating Multiply

Format:

```
        MUL     Ra.rx,Rb.rx,Rc.wx                        !Operate format
```

Operation:

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

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

MULF	Multiply F_floating VAX Rounding
MULFZ	Multiply F_floating Round toward Zero
MULG	Multiply G_floating VAX Rounding
MULGZ	Multiply G_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

MULFU	Multiply F_floating VAX Rounding
MULFUZ	Multiply F_floating Round toward Zero
MULGU	Multiply G_floating VAX Rounding
MULGUZ	Multiply G_floating Round toward Zero

Description:

The multiplicand in register Rb is multiplied by the multiplier in register Ra, and the product is 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.

Floating Subtract

Format:

```
        SUB     Ra.rx,Rb.rx,Rc.wx                    !Operate format
```

Operation:

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

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

SUBF	Subtract F_floating VAX Rounding
SUBFZ	Subtract F_floating Round toward Zero
SUBG	Subtract G_floating VAX Rounding
SUBGZ	Subtract G_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

SUBFU	Subtract F_floating VAX Rounding
SUBFUZ	Subtract F_floating Round toward Zero
SUBGU	Subtract G_floating VAX Rounding
SUBGUZ	Subtract G_floating Round toward Zero

Description:

The subtrahend operand in register Ra is subtracted from the minuend operand in register Rb, and the difference is 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.

Vector Floating Add

Format:

```
        VADD    Va.rx,Vb.rx,Vc.wx                       !Operate format
        VADD    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
        FOR i <- 0 TO VL-1
            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

Opcodes:

The following instructions disable the Floating Underflow exception:

VADDF	Vector Add F_Floating VAX Rounding
VADDFZ	Vector Add F_Floating Round toward Zero
VADDG	Vector Add G_Floating VAX Rounding
VADDGZ	Vector Add G_Floating Round toward Zero

The following instructions enable the Floating Underflow exception:

VADDFU	Vector Add F_floating VAX Rounding
VADDFUZ	Vector Add F_floating Round toward Zero
VADDGU	Vector Add G_floating VAX Rounding
VADDGUZ	Vector Add G_floating Round toward Zero

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 VADDFx, 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:

```
        VCMP    Va.rx,Vb.rx                         !Operate format
        VCMP    Ra.rx,Vb.rx
```

Operation:

```
        VM <- 0
        FOR i <- 0 TO VL-1
            BEGIN
                                    !VCMPF Vector cmp Vector
            IF  Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0>  THEN
                VM<i> <- 1
                                    !VCMPF Scalar cmp Vector
            IF  Rav SIGNED_RELATION Vb[i]<31:0>  THEN
                VM<i> <- 1
                                    !VCMPG Vector cmp Vector
            IF  Va[i] SIGNED_RELATION Vb[i]  THEN
                VM<i> <- 1
                                    !VCMPG Scalar cmp Vector
            IF  QRav SIGNED_RELATION Vb[i]  THEN
                VM<i> <- 1
            END
```

Exceptions:

Floating Reserved Operand

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

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. The Vector Mask register (VM) is cleared at the start of the operation. For each element comparison, if the specified relationship is true, the Vector Mask bit (VM<i>) corresponding to the vector element is set to 1. In

VCMPFx, only bits <31:0> of each vector element participate in the operation.

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:

```
        VCVT     Va.rf,Vc.wg                         !Operate format
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- {conversion of Va[i]<31:0>}
            END
```

Exceptions:

Floating Reserved Operand

Opcodes:

VCVTFG Vector Convert F_floating to G_floating

Description:

The F_floating vector elements in vector register Va 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:

```
        VCVT     Va.rg,Vc.wf                            !Operate format
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- {conversion of Va[i]}
            END
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

VCVTGF	Vector Convert G_floating to F_floating VAX Rounding
VCVTGFZ	Vector Convert G_floating to F_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

VCVTGFU	Vector Convert G_floating to F_floating VAX Rounding
VCVTGFUZ	Vector Convert G_floating to F_floating Round toward Zero

Description:

The G_floating vector elements in vector register Va 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 Integer

Format:

```
        VCVT     Va.rx,Vc.wl                          !Operate format
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- {conversion of Va[i]}                 !VCVTGL
            Vc[i] <- {conversion of Va[i]<31:0>}           !VCVTFL
            END
```

Exceptions:

Floating Reserved Operand
Integer Overflow

Opcodes:

VCVTFL	Vector Convert F_floating to Longword VAX Rounding
VCVTFLZ	Vector Convert F_floating to Longword Round toward Zero
VCVTGL	Vector Convert G_floating to Longword VAX Rounding
VCVTGLZ	Vector Convert G_floating to Longword Round toward Zero

Description:

The F_ or G_floating vector elements in vector register Va 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. The length of the vector is specified by the VL register.

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

These instructions may be omitted in a subset implementation.

Vector Convert Integer to Floating

Format:

```
        VCVT     Va.rl,Vc.wx                              !Operate format
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
            Vc[i] <- {conversion of Va[i]<31:0>}
            END
```

Exceptions:

None

Opcodes:

VCVTLG	Vector Convert Longword to G_floating
VCVTLF	Vector Convert Longword to F_floating VAX Rounding
VCVTLFZ	Vector Convert Longword to F_floating Round toward Zero

Description:

The longword integer vector elements in register Va 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:

```
        VDIV    Va.rx,Vb.rx,Vc.wx                     !Operate format
        VDIV    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
                                                     !VDIVF
            Vc[i] <- Vb[i]<31:0> / Va[i]<31:0>       !Vector / Vector
            Vc[i] <- Vb[i]<31:0> / Rav               !Vector / Scalar

                                                     !VDIVG
            Vc[i] <- Vb[i] / Va[i]                   !Vector / Vector
            Vc[i] <- Vb[i] / QRav                    !Vector / Scalar
            END
```

Exceptions:

Floating Divide by Zero
Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

VDIVF	Vector Divide F_floating VAX Rounding
VDIVFZ	Vector Divide F_floating Round toward Zero
VDIVG	Vector Divide G_floating VAX Rounding
VDIVGZ	Vector Divide G_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

VDIVFU	Vector Divide F_floating VAX Rounding
VDIVFUZ	Vector Divide F_floating Round toward Zero
VDIVGU	Vector Divide G_floating VAX Rounding
VDIVGUZ	Vector Divide G_floating Round toward Zero

Description:

The dividend in vector register Vb is divided, element-wise, by a divisor vector operand (in register Va) or a scalar operand (in register Ra or QRa), 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:

```
        VMUL    Va.rx,Vb.rx,Vc.wx                         !Operate format
        VMUL    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
        FOR i <- 0 TO VL-1
            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

Opcodes:

The following instructions disable the Floating Underflow exception:

VMULF	Vector Multiply F_floating VAX Rounding
VMULFZ	Vector Multiply F_floating Round toward Zero
VMULG	Vector Multiply G_floating VAX Rounding
VMULGZ	Vector Multiply G_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

VMULFU	Vector Multiply F_floating VAX Rounding
VMULFUZ	Vector Multiply F_floating Round toward Zero
VMULGU	Vector Multiply G_floating VAX Rounding
VMULGUZ	Vector Multiply G_floating Round toward Zero

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:

```
        VSUB    Va.rx,Vb.rx,Vc.wx                        !Operate format
        VSUB    Ra.rx,Vb.rx,Vc.wx
```

Operation:

```
        FOR i <- 0 TO VL-1
            BEGIN
                                                     !VSUBF
            Vc[i] <- Vb[i]<31:0> - Va[i]<31:0>       !Vector - Vector
            Vc[i] <- Vb[i]<31:0> - Rav               !Vector - Scalar

                                                     !VSUBG
            Vc[i] <- Vb[i] - Va[i]                   !Vector - Vector
            Vc[i] <- Vb[i] - QRav                    !Vector - Scalar
            END
```

Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

The following instructions disable the Floating Underflow exception:

VSUBF	Vector Subtract F_floating VAX Rounding
VSUBFZ	Vector Subtract F_floating Round toward Zero
VSUBG	Vector Subtract G_floating VAX Rounding
VSUBGZ	Vector Subtract G_floating Round toward Zero

The following instructions enable the Floating Underflow exception:

VSUBFU	Vector Subtract F_floating VAX Rounding
VSUBFUZ	Vector Subtract F_floating Round toward Zero
VSUBGU	Vector Subtract G_floating VAX Rounding
VSUBGUZ	Vector Subtract G_floating Round toward Zero

Description:

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

In VSUBFx, 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
FOB	Fault On Low Bit Set
JSR	Jump to Subroutine

Conditional Branch

Format:

```
        Bxx   Ra.rl,disp.al                         !Branch format
```

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 Set

Format:

```
        FOB   Ra.rl,disp.al                              !Branch format
```

Operation:

```
        IF  Rav<0> EQ 1  THEN
            {FOB exception}
```

Exceptions:

Fault On Bit

Opcodes:

FOB	Fault On Low Bit Set

Description:

Bit <0> of Register Ra is tested. If it is set to 1, a Fault On Bit exception is generated (see Chapter 6, Exceptions and Interrupts, Section 6.4.3.3; 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:

```
        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
BUGCHK	System Bug Check
DRAIN	Drain the Pipeline
IFLUSH	Flush I-Stream Cache
IOTA	Generate Compressed Iota Vector
MOVPS	Move Processor Status
PROBER	Probe Read Access
PROBEW	Probe Write Access
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:

```
        BPT                                         !Epicode format
```

Operation:

```
        {push current PC and PS on Kernel stack}

        {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 and PS on the Kernel stack. It then dispatches to the address in the Breakpoint SCB vector. See Chapter 6, Exceptions and Interrupts, Section 6.4.3.1.

Bug Check

Format:

```
        BUGCHK                                          !Epicode format
```

Operation:

```
        {push current PC and PS on Kernel stack}

        {Change Mode To Kernel}

        {dispatch through BUGCHK SCB vector}
```

Exceptions:

Kernel Stack Not Valid

Opcodes:

BUGCHK Bug Check

Description:

This instruction is used to report software-detected errors in system software. It switches to Kernel mode and pushes the current PC and PS on the Kernel stack. It then dispatches to the address in the BUGCHK SCB vector. See Chapter 6, Exceptions and Interrupts, Section 6.4.3.2.

Drain Instruction Pipeline

Format:

```
        DRAIN                                        !Epicode format
```

Operation:

```
        {Stall instruction issuing until all prior instructions have
        completed.}
```

Exceptions:

None

Opcodes:

DRAIN	Drain Instruction Pipeline

Description:

The DRAIN instruction allows software to guarantee that in a pipelined implementation all previous instructions have completed 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 have completed without exceptions. If an exception occurs, the continuation PC in the exception stack frame points to the DRAIN instruction.

e s

iot
re
zerc
inat
etur
th i

ins

Flush Instruction Cache

Format:

```
        IFLUSH                                          !Epicode format
```

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 processors 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:

```
        IOTA    Ra.rl,Vc.wl                        !Operate format
        IOTA    #a.ib,Vc.wl
```

Operation:

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

Exceptions:

None

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. For example,

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

The a vector is then compressed using the contents of the Vector
Mask gister (VM). Elements of the iota vector corresponding to the
non- bits of VM are written to contiguous elements of the
dest ion vector register, Vc. The number of elements written to Vc
is r ned in the Vector Count register (VC) for use as a vector
leng n subsequent operations.

This truction may be omitted in a subset implementation.

Move Processor Status

Format:

```
        MOVPS                                             !Epicode format
```

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:

```
        PROBE                                        !Epicode format
```

Operation:

```
        ! R4 contains the base address
        ! R5 contains the signed offset
        ! R6 contains the access 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<1:0>, 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 the value 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<1:0> and the Current Mode (PS<CM>). See Chapter 6, Exceptions and Interrupts, Section 6.2 for access mode encodings.

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

Read/Write Vector Count Register

Format:

```
        RDVC    Rc.wl                                   !Operate format
        WRVC    Ra.rl
        WRVC    #a.ib
```

Operation:

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

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

Exceptions:

None

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 instructione may be omitted in a subset implementation.

Read/Write Vector Length Register

Format:

```
        RDVL    Rc.wl                                   !Operate format
        WRVL    Ra.rl
        WRVL    #a.ib
```

Operation:

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

        VL <- Rav<5:0>                                  !WRVL
```

Exceptions:

None

Opcodes:

RDVL	Read Vector Length Register
WRVL	Write Vector Length Register

Description:

RDVL reads the 6-bit Vector Length register and writes it zero extended to register Rc.

WRVL writes Rav<5:0> to the Vector Length register. Writing a zero to VL is equivalent to a vector length of 64.

These instructione may be omitted in a subset implementation.

Read/Write Vector Mask Register

Format:

```
        RDVM    Rc.wl                               !Operate format
        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:

None

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 it 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:

```
        REI                                                 !Epicode format
```

Operation:

```
        IF  SP<2:0> NE 0  THEN
            {Stack Alignment exception}

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

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

            tmp1<VEN> <- tmp1<VEN> AND PS<VEN>
            IF  {NOT tmp1<VEN>} AND tmp1<VEF>  THEN
                {Illegal Operand exception}
            END

        IF  tmp1<VMM> EQ 1  THEN
            {perform TBD action}
        IF  tmp1<VEF> EQ 1  THEN
            {perform Vector Exception Continuation}

        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
Stack Alignment
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 <VEF> is set in the new PS then REI will perform a vector exception continuation operation. See Chapter 6, Exceptions and Interrupts, Section 6.4.8.1 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 6.7.6).

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 simplified.\

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

Swap AST Enable

Format:

```
        SWASTEN                                          !Epicode format
```

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 6.7.6.4).

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
HALT	Halt Processor
MFPR	Move From Processor Register
MTPR	Move To Processor Register
RMAQIP	Read, Mask, Add Quadword, Interlocked, Physical
SWPCTX	Swap Privileged Context
SWIPL	Swap IPL
TBFLUSH	Flush Translation Buffer

Halt

Format:

```
        HALT                                          !Epicode format
```

Operation:

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

        IF  {halt_action} EQ HALT THEN
            {enter console mode}
        ELSE
            {enter restart sequence}
```

Exceptions:

Privileged Instruction

Opcodes:

HALT	Halt Processor

Description:

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

Move From Processor Register

Format:

```
        MFPR    IPR_Name                            !Epicode format
```

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 the result of reading the register, e.g., 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:

```
        MTPR    IPR_Name                              !Epicode format
```

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.

Read, Mask, Add Quadword, Interlocked, Physical

Format:

```
        RMAQIP                                   !Epicode format
```

Operation:

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

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

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

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

Exceptions:

Machine Check
Privileged Instruction

Opcodes:

RMAQIP Read, Mask, Add Quadword, Interlocked, Physical

Description:

The quadword aligned memory operand, whose physical address is in QR4, 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. The low three bits of the operand address in QR4 are ignored.

This instruction performs an interlocked memory access in that no other processor in a multiprocessor system can perform an interlocked operation on the same operand until the current interlocked operation has completed. This is an Epicode instruction.

The operation is UNDEFINED if RMAQIP 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.

Swap Privileged Context

Format:

```
        SWPCTX                                      !Epicode format
```

Operation:

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

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

        ! Store old HWPCB contents

        (HWPCB_KSP) <- SP
        IF  {internal registers for stack pointers}  THEN
            BEGIN
            (HWPCB_ESP) <- IPR_ESP
            (HWPCB_SSP) <- IPR_SSP
            (HWPCB_USP) <- IPR_USP
            END

        (HWPCB_ASTSR) <- IPR_ASTSR
        (HWPCB_ASTEN) <- IPR_ASTEN

        ! Load new HWPCB contents

        IPR_PCBB <- QR4

        IF  {ASNs not implemented}  THEN
            {invalidate translation buffer entries with PTE<ASM> EQ 0}
        ELSE
            IPR_ASN <- (HWPCB_ASN)

        IF  {virtual instruction cache implemented}  THEN
            {flush instruction cache}

        SP <- (HWPCB_KSP)

        IF {internal registers for stack pointers}  THEN
            BEGIN
            IPR_ESP <- (HWPCB_ESP)
            IPR_SSP <- (HWPCB_SSP)
            IPR_USP <- (HWPCB_USP)
            END

        IPR_PTBR  <- (HWPCB_PTBR)
        IPR_ASTSR <- (HWPCB_ASTSR)
        IPR_ASTEN <- (HWPCB_ASTEN)
```

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 four stack pointers, the Page Table Base Register (PTBR), the Address Space Number (ASN), and the AST enable and summary registers. 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 may result 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.

Note:

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 access mode in SP and switch this with the HWPCB contents whenever the current access mode changes.

Swap IPL

Format:

```
        SWIPL                                         !Epicode format
```

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 previous IPL level is returned in R4.

A check is made to determine if an AST is pending (see Chapter 6, Exceptions and Interrupts, Section 6.7.6).

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:

```
        TBFLUSH                                         !Epicode format
```

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 flushes all virtual caches. To invalidate a single TB entry use the MTPR TBIS instruction.

4.9 COPROCESSOR INSTRUCTIONS

The Coprocessor instructions provide the means to transfer data, control, and status information between a PRISM processor and one or more application-specific computing elements called coprocessors. They also provide the ability for a program on a PRISM processor to synchronize itself with the operation of a coprocessor. The actual operation performed by a coprocessor is implementation-specific.

The following instructions are provided:

Mnemonic	Operation
COPRD	Coprocessor Read
COPWR	Coprocessor Write

Coprocessor Read/Write

Format:

```
        COPRD   Ra.wl,#ctrl.ix,#caddr.ix,#te.ix    !Coprocessor format

        COPWR   Ra.rl,#ctrl.ix,#caddr.ix,#te.ix
        COPWR   #a.ib,#ctrl.ix,#caddr.ix,#te.ix
```

Operation:

```
        Coprocessor[caddr] <- Rav || ctrl                       !COPWR

        Coprocessor[caddr] <- ctrl                              !COPRD
        Ra <- Coprocessor_data[caddr]                           !COPRD

        IF  {te EQ 1} AND {Coprocessor Exception}  THEN
            {take Arithmetic exception}
```

Exceptions:

Arithmetic

Opcodes:

COPRD	Coprocessor Read
COPWR	Coprocessor Write

Description:

COPRD reads data from a coprocessor and writes it to the PRISM scalar register Ra. COPWR writes the data in PRISM scalar register Ra to a coprocessor.

The Coprocessor instruction format provides a 10-bit Coprocessor control field (ctrl operand), a 9-bit Coprocessor address field (caddr operand) and a 1-bit trap enable field (te operand). See Chapter 3, Instruction Formats, Section 3.3.5.

- o The ctrl operand is passed to the coprocessor to control the operation performed.

- o The caddr operand is used to select a specific coprocessor in a system with more than one.

- o The te operand is used to enable exceptions on transactions with a coprocessor. A Coprocessor Read or Write can generate an exception if an exception condition is present in the coprocessor and te is set to 1. When the exception occurs on a COPRD, the value written to the PRISM destination register (Ra) is UNPREDICTABLE. The coprocessor may contain a status register that can be read with a COPRD to give additional information about the exception. If te is 0, the Arithmetic exception is suppressed. This could be used to ignore exceptions (e.g., when context switching).

These instructions may be omitted in a subset implementation that does not provide a Coprocessor interface.

Revision History:

Revision 1.0, 22 December 1985

1. Changed register width from 64 bits to 32 bits.
2. Changed Epicode parameter registers to R4-R7.
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, four hierarchical access modes provide memory access control. They are, from most to least privileged: Kernel, Executive, Supervisor, and User. Protection is specified at the individual page level for data and instruction access. A page may be inaccessible or may have different data or instruction accessibility for each of the four access modes. Data accessibility can be read-only, read/write, or no access. Any location accessible as data to one mode is also accessible as data to all more privileged modes. Furthermore, for each access mode, any location that can be written can also be read. For instructions, execute access in one mode implies execute access in all more privileged modes.

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:

- o Provide a large address space for instructions and data.

- o Allow programs to run on hardware with physical memory smaller than the virtual memory used.

- o Provide convenient and efficient sharing of instructions and data.
- o Allow sparse use of a large address space without excessive page table overhead.
- o Contribute to software reliability.
- o Provide independent execute, read and write access protection.
- o 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 (see Appendix B). 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.

```
 3               2 2                 1 1
 1               3 2                 3 2                        0
+-----------------+-------------------+--------------------------+
|   Seg1_Number   |    Seg2_Number    |     Byte Within Page     |
+-----------------+-------------------+--------------------------+
```

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.

```
 3 3 2 2 2 2 2          2 1 1    1 1
 1 0 9 8 7 6 5          0 9 8    6 5                                0
+-+-+-+-+-+-+-----------+-+-----+-----------------------------------+
|                                                                   |
|                       Page Frame Number                           | :A
|                                                                   |
+-+-+-+-+-+-+-----------+-+-----+-----------------------------------+
| |F|F|F|D|A|           |I|     |                                   |
|V|O|O|O|C|S| Protection|N| RSVD|      Reserved for Software        | :A+4
| |R|W|E|V|M|           |D|     |                                   |
+-+-+-+-+-+-+-----------+-+-----+-----------------------------------+
```

Figure 5-2: Page Table Entry

Fields in the highest addressed longword are interpreted as follows:

Bits	Description
31	Valid (V) - Indicates the validity of the DCV, ASM, FOE, FOW, FOR bits and the PFN field. When V is set, the DCV, 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.
30	Fault On Read (FOR) - When set, a Fault On Read exception occurs on an attempt to read any location in the page.
29	Fault On Write (FOW) - When set, a Fault On Write exception occurs on an attempt to write any location in the page.
28	Fault On Execute (FOE) - When set, a Fault On Execute exception occurs on an attempt to execute an instruction in the page.
27	Don't Cache Virtual (DCV) - When set, contents of locations in this page are not cached in a virtual cache. \This is intended for use in systems with virtual caches when shared writable pages exist at multiple virtual addresses and map to the same physical address.\
26	Address Space Match (ASM) - When set, this PTE matches all Address Space Numbers.
25:20	Protection (PROT) - Indicates at what access modes a process can reference the page. This field is always valid in the final PTE and is used by the processor hardware even when V is clear.
19	Indirect Page Table Pointer (IND) - If V is clear, and IND is set, then bits <44:0> contain the physical address of the indirect quadword aligned PTE, and all other bits are ignored. When V is set, IND is ignored.

18:16	Reserved for future use by DIGITAL.
15:0	Reserved for software except when V is clear and IND is set.

Fields in the lowest addressed longword are interpreted as follows:

Bits	Description
31:0	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 and IND is clear, this field may be used by 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).

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 or RMAQIP) 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. PTEs may be read with non-interlocked quadword operations if they are not being modified. Software may write new values into invalid PTEs using non-interlocked quadword store instructions (i.e., STQ). Hardware must ensure that aligned quadword reads and writes are indivisible operations.

5.6 MEMORY PROTECTION

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

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

- o The virtual address, which is used to index page tables.
- o The intended access type (read data, write data, or instruction fetch).
- o The current access 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.

The intended access is READ if the operation to be performed is a data read. The intended access is WRITE if the operation to be performed is a data write. The intended access is EXECUTE if the operation to be performed is an instruction fetch.

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 Access Modes

In the order of most privileged to least privileged, the four processor modes are:

- o Kernel
- o Executive
- o Supervisor
- o User

The access mode of a running process is stored in the Current Mode field 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 executing, reading, or modifying 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 protection for each processor mode, within the following limits:

- o Each mode's access can be read/write, read-only, or no-access for data references.
- o Except for Kernel mode, each mode's access can be execute or no-execute for instruction execution.
- o Data and execution accessibility are specified independently. Thus, execute access can be allowed to a page that cannot be read. Also, execution access can be prevented to a page that can be written as data.
- o If any level has execute access then all more privileged levels also have execute access.
- o If any level has read access then all more privileged levels also have read access.
- o If any level has write access then all more privileged levels also have write access.

The protection code is specified by a 6-bit field in the PTE. Bits <1:0> specify execute accessibility and bits <5:2> specify data accessibility.

Table 5-1: PTE Protection Codes

			Accessibility			
Name	Mnemonic	PROT <5:0>	Kernel	Exec	Super	User
no data access	NDA_	0000xx	none	none	none	none
reserved	RSVD	0001xx		UNPREDICTABLE		
Kernel write	KW_	0010xx	write	none	none	none
Kernel read	KR_	0011xx	read	none	none	none
User write	UW_	0100xx	write	write	write	write
Exec write	EW_	0101xx	write	write	none	none
Exec read, Kernel write	ERKW_	0110xx	write	read	none	none
Exec read	ER_	0111xx	read	read	none	none
Super write	SW_	1000xx	write	write	write	none
Super read, Exec write	SREW_	1001xx	write	write	read	none
Super read, Kernel write	SRKW_	1010xx	write	read	read	none
Super read	SR_	1011xx	read	read	read	none
User read, Super write	URSW_	1100xx	write	write	write	read
User read, Exec write	UREW_	1101xx	write	write	read	read
User read, Kernel write	URKW_	1110xx	write	read	read	read
User read	UR_	1111xx	read	read	read	read
Kernel execute	KX	xxxx00	execute	none	none	none
Exec execute	EX	xxxx01	execute	execute	none	none
Super execute	SX	xxxx10	execute	execute	execute	none
User execute	UX	xxxx11	execute	execute	execute	execute

The full mnemonic is obtained by concatenating the data and instruction execution access mnemonics. For example, UR_KX denotes User read, Kernel execute (code=111100 (bin)).

\This encoding was chosen to simplify hardware access checking for implementations not using a table decoder. An access is allowed if:

```
{d_stream_access AND {CODE<5:2> NE 0} AND
    {{CODE<5:2> EQ 4} OR {CM LTU WM} OR {read_access AND {CM LEU RM}}}}
OR
{i_stream_access AND {CM LEU XM}}
```

Where:

CM	is current processor mode
RM	is protection code <5:4>
WM	is ones complement of protection code <3:2>
XM	is protection code <1:0>

5.6.3 Access Control Violation Fault

An Access Control Violation fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection field, or if the second longword of a PTE is zero.

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 second longword of that PTE equals zero. An Access Control Violation on a first-level PTE (zero 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)

Mbytes) map the entire 4-Gbyte address space.\

The PRISM architecture supports indirect PTEs for facilitating shared pages. If an indirect PTE resides in I/O space or in nonexistent memory, the operation of the processor is UNDEFINED. Only one level of indirection is allowed at each page table level.

\The primary benefit of indirection is that it allows the software bits to be maintained in a single place for shared pages. It is also useful for sharing page tables that map read-only code or data, e.g., shared runtime libraries.\

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
    BEGIN
    IF seg1_pte<IND> EQ 0 THEN
        IF seg1_pte<63:32> EQ 0 THEN
            {initiate Access Control Violation fault}
        ELSE
            {initiate Translation Not Valid fault}
    ELSE
        BEGIN
        seg1_pte <- (seg1_pte<44:0>)    !Read Physical
        IF seg1_pte<V> EQ 0 THEN
            IF seg1_pte<63:32> EQ 0 THEN
                {initiate Access Control Violation fault}
            ELSE
                {initiate Translation Not Valid fault}
        END
    END

seg2_pte <- ({seg1_pte<PFN> * 8192} + {8 * VA<22:13>})  !Read Physical

IF seg2_pte<V> EQ 0 THEN
    BEGIN
    IF seg2_pte<IND> EQ 0 THEN
        IF {seg2_pte<PROT> check fails} OR
           {seg2_pte<63:32> EQ 0} THEN
            {initiate Access Control Violation fault}
        ELSE
            {initiate Translation Not Valid fault}
    ELSE
        BEGIN
        seg2_pte <- (seg2_pte<44:0>)    !Read Physical
        IF seg2_pte<V> EQ 0 THEN
            IF {seg2_pte<PROT> check fails} OR
               {seg2_pte<63:32> EQ 0} THEN
                {initiate Access Control Violation fault}
            ELSE
                {initiate Translation Not Valid fault}
        END
    END

IF {seg2_pte<PROT> check fails} THEN
    {initiate Access Control Violation fault}
ELSE
    BEGIN
    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>
    END
```

5.8 TRANSLATION BUFFER

In order to save actual memory references when repeatedly referencing the same pages, a hardware implementation may include a mechanism to remember successful virtual address translations and page states. Such a mechanism is termed a Translation Buffer.

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 4-93 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 8-26.

\Some implementations may invalidate the entire Translation Buffer on an MTPR to TBIASN or 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 4-97.

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 unending 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 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 number of bits in the address space number is implementation dependent. 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.

When software reassigns an address space number to a different process, it must invalidate address translations for the previous process by executing an MTPR to the TBIASN register; see Chapter 8, Internal Processor Registers, Page 8-25.

\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.

Invalidation of TB entries for a specific ASN can take a long time if the hardware does not support a mechanism to associatively invalidate TB entries by ASN. A possible solution to this problem would be for software to assign a new ASN value to a process when it is rescheduled on a new processor. When the processor eventually runs out of unused ASN values, the entire TB can be flushed by software.

Are ASNs really a big win in multiprocessor systems? Should we get rid of them? \

5.10 MEMORY MANAGEMENT FAULTS

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

- o Access Control Violation
- o Fault On Read
- o Fault On Write
- o Fault On Execute
- o Translation Not Valid

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

An Access Control 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 access mode. An Access Control Violation fault is also taken if the second longword of a PTE is zero.

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. A PTE is invalid if V is clear and IND is clear. TNV also occurs if an indirect PTE at any level has V clear.

Note that these five faults have distinct vectors in the System Control Block. The Access Control 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 Read, Mask, Add Quadword Interlocked instruction, in which case the order that the exceptions are taken is UNPREDICTABLE; see Chapter 4, Instruction Descriptions, Page 4-9.

Revision History:

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 Lrun 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 Port Controllers, 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 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, however, such as machine check, raise 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.

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 more easily continuable without the need for complicated fix-up code. This is accomplished by placing a Drain Pipeline (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 very similar. However, there are five 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.

5. Interrupts always set the (new) current mode to Kernel while exceptions set the (new) current mode to either Kernel or leave it the same as it was immediately prior to the exception.

6.2 PROCESSOR STATE

Processor state consists of a longword of privileged information 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 target mode 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 4-85.

\Initiation of an exception or interrupt causes the PC, followed by the PS, to be pushed on the target mode 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 32-bit 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 4-80. 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 4-73 and 4-71.

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.

```
 3
 1                                              8 7   5 4 3 2 1 0
+----------------------------------------------+-----+-+-+-+---+
|                                              |     |V|V|V|   |
|                    MBZ                       | IPL |E|E|M| CM|
|                                              |     |N|F|M|   |
+----------------------------------------------+-----+-+-+-+---+
```

Figure 6-1: Processor Status

Bits   Description

1:0   Current Mode (CM). The access mode of the currently executing process as follows:

0 - Kernel
1 - Executive
2 - Supervisor
3 - User

2   Virtual Machine Mode (VMM) - When set, the processor is in virtual machine mode. This bit is only meaningful when running with a virtual machine monitor. When clear, the processor is running in real machine mode.

\The exact rules for using this bit have not been fully defined.\

3   Vector Exception Frame (VEF) - This bit can only be set in a PS which has been saved during the initiation of an exception. When set, one or more vector exception information frames have been pushed on the stack prior to the saved PS and PC.

4   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.

7:5   Interrupt Priority Level (IPL) - The current processor priority, in the range 0 to 7.

31:8   Reserved to DIGITAL, MBZ.

At bootstrap, the initial value of PS is set to E0 (hex). VEF, VEN, VMM, and CM are clear and IPL is 7.

```
 3
 1                                                      2 1 0
+-------------------------------------------------------+---+
|                                                       | I |
|              Instruction Virtual Address              | G |
|                                                       | N |
+-------------------------------------------------------+---+
```

Figure 6-2: Program Counter

All instructions are aligned on longword boundaries and, therefore, hardware can assume zero for the two low order bits of PC.

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 cause an interrupt:

o Asynchronous System Trap (AST) - IPL 1.

o Software interrupt - IPL 1 to 3.

o Console interrupts - IPL 4.

o I/O Port Controller interrupts - IPL 4 and 5.

o 10 ms Interval Clock interrupt - IPL 5.

o Interprocessor interrupt - IPL 6.

o Power Recovery interrupt - IPL 7.

o 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 below. The vector location for architecturally defined interrupts is fixed by the architecture.

\It would be nice if there were no assignable vectors. Do we really need them?\

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 an interrupt at an intermediate level which would cause the stack nesting to be incorrect.

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 4-96 and 4-85.

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 Generated 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

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 Port Controllers - Levels 4 And 5

The architecture provides two priority levels for use by I/O Port Controllers.

I/O Port Controller interrupts are requested when a completion or attention packet is inserted into an empty I/O Port Controller response queue by an I/O processor.

6.3.5 Interval Clock Interrupt - Level 5

The 10ms Interval Clock requests an interrupt every 10ms if clock interrupts are enabled.

6.3.5.1 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.

6.3.6 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, 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. Machine Check is documented below under Exceptions, 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.

Interrupt level 6 is reserved for interprocessor interrupt requests.

6.3.6.1 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.

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.

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.

6.3.6.2 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.

If the specified processor is the same as the current processor, the current IPL is less than 6, and interprocessor interrupts are enabled, the interrupt will be taken on the initiating processor before the execution of the next instruction.

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 exceptions
3. Faults occurring as a consequence of an instruction
4. Memory management faults
5. Serious system failures
6. Stack Alignment aborts
7. Vector exceptions

Each exception has a separate vector location (offset) within the System Control Block (SCB); see Section 6.6 below.

When initiating an exception, various parameters are pushed on the target 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), 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 6.4.7 below.

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 more easily continuable 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.

Furthermore, the order of discovery of F_ and G_floating arithmetic traps is UNPREDICTABLE. For example, if both a zero divisor and a reserved dividend are specified, it is UNPREDICTABLE which will actually be reported.

It is permissible for an implementation to use a forwarded or bypassed result in a subsequent instruction, even if the result is exceptional, provided that error information is propagated to the destination register and the appropriate bits are set in the respective register write mask (see below).

Arithmetic traps are initiated in Kernel mode and push the following information on the Kernel stack:

```
 3                             1 1
 1                             6 5                             0
+-------------------------------+-------------------------------+
|                                                               |
|                       Exception Summary                       | :SP
|                                                               |
+-------------------------------+-------------------------------+
|                               |        Vector Register        |
|             Zero              |        Write  Mask for        |
|                               |      Registers  V0 - V15      |
+-------------------------------+-------------------------------+
|                        Scalar Register                        |
|                        Write  Mask for                        |
|                      Registers  R0 - R31                      |
+---------------------------------------------------------------+
|                        Scalar Register                        |
|                        Write  Mask for                        |
|                      Registers R32 - R64                      |
+---------------------------------------------------------------+
|                                                               |
|                     Processor Status (PS)                     |
|                                                               |
+---------------------------------------------------------------+
|                            Virtual                            |
|                        Address of Next                        |
|                          Instruction                          |
+---------------------------------------------------------------+
```

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.

```
 3
 1                                                7 6 5 4 3 2 1 0
+------------------------------------------------+-+-+-+-+-+-+-+-+
|                                                |C|I|I|F|F|F|F|
|                     Zero                       |O|O|D|O|R|D|U|
|                                                |E|V|Z|V|S|Z|N|
+------------------------------------------------+-+-+-+-+-+-+-+-+
```

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.
6	Coprocessor Exception (COE) - A Coprocessor read or write instruction with trap enable set was executed when a Coprocessor exception was present.

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 4-46.

6.4.2 Data Alignment Exceptions

All data must be naturally aligned or an alignment exception 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 current mode and push the following information on the Current Mode stack:

```
 3
 1                                                           0
+--------------------------------------------------------------+
|                         Virtual                              |
|                       Address  of                            | :SP
|                        Reference                             |
+--------------------------------------------------------------+
|                                                              |
|                   Faulting Instruction                       |
|                                                              |
+--------------------------------------------------------------+
|                                                              |
|                  Processor Status (PS)                       |
|                                                              |
+--------------------------------------------------------------+
|                         Virtual                              |
|                  Address of Faulting                         |
|                       Instruction                            |
+--------------------------------------------------------------+
```

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, in the current mode.

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 generating this exception.

Emulation software, whether Epicode, an operating system, or 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.

6.4.2.2 Vector Alignment Abort

A Vector Alignment abort is generated when an attempt is made to load/store a longword or quadword element to/from a vector register using an address that does not have the natural alignment of the particular data reference.

Vector Alignment aborts are initiated in Kernel mode and push the following information on the Kernel stack:

```
 3
 1                                                             0
+---------------------------------------------------------------+
|                           Virtual                             |
|                         Address  of                           | :SP
|                          Reference                            |
+---------------------------------------------------------------+
|                                                               |
|                             Zero                              |
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                    Processor Status (PS)                      |
|                                                               |
+---------------------------------------------------------------+
|                           Virtual                             |
|                      Address of Next                          |
|                         Instruction                           |
+---------------------------------------------------------------+
```

Figure 6-6: Vector Alignment Abort Exception Frame

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 4-75. 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:

```
 3
 1                                                             0
+---------------------------------------------------------------+
|                                                               |
|                    Processor Status (PS)                      | :SP
|                                                               |
+---------------------------------------------------------------+
|                           Virtual                             |
|                      Address  of BPT                          |
|                         Instruction                           |
+---------------------------------------------------------------+
```

Figure 6-7: 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 Bug Check Fault

A Bug Check fault is an exception that occurs when a Bug Check (BUGCHK) instruction is executed; see Chapter 4, Instruction Descriptions, Page 4-76. This opcode is provided for use by operating system error reporting software.

Bug Check faults are initiated in Kernel mode and push the following information on the Kernel stack:

```
 3
 1                                                             0
+--------------------------------------------------------------+
|                                                              |
|                   Processor Status (PS)                      | :SP
|                                                              |
+--------------------------------------------------------------+
|                          Virtual                             |
|                    Address of BUGCHK                         |
|                        Instruction                           |
+--------------------------------------------------------------+
```

Figure 6-8: Bug Check Fault Exception Frame

6.4.3.3 Fault On Bit

A Fault On Bit fault is an exception that occurs when a Fault On Bit (FOB) instruction is executed and the low order bit of the specified scalar register is set; see Chapter 4, Instruction Descriptions, Page 4-72.

Fault On Bit faults are initiated in the current mode and push the following information on the Current Mode stack:

```
 3
 1                                                            0
+--------------------------------------------------------------+
|                                                              |
|                             Zero                             | :SP
|                                                              |
+--------------------------------------------------------------+
|                                                              |
|                     Faulting  Instruction                    |
|                                                              |
+--------------------------------------------------------------+
|                                                              |
|                     Processor Status (PS)                    |
|                                                              |
+--------------------------------------------------------------+
|                            Virtual                           |
|                        Address  of FOB                       |
|                          Instruction                         |
+--------------------------------------------------------------+
```

Figure 6-9: Fault On Bit 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, in the current mode.

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 current mode and push the following information on the Current Mode stack:

```
 3
 1                                                           0
+------------------------------------------------------------+
|                                                            |
|                           Zero                             | :SP
|                                                            |
+------------------------------------------------------------+
|                                                            |
|                    Faulting  Instruction                   |
|                                                            |
+------------------------------------------------------------+
|                                                            |
|                   Processor Status (PS)                    |
|                                                            |
+------------------------------------------------------------+
|                          Virtual                           |
|                    Address of Faulting                     |
|                        Instruction                         |
+------------------------------------------------------------+
```

Figure 6-10: Illegal Operand Fault Exception Frame

Illegal operands include:

- o An interlock address that is not quadword aligned (RMAQI)
- o An invalid combination of bits in the PS restored by 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, in the current mode.

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, Supervisor, or Executive. Privileged operations can only be executed in Kernel mode.

Privileged Instruction faults are initiated in the current mode and push the following information on the Current Mode stack:

```
 3
 1                                                             0
+---------------------------------------------------------------+
|                                                               |
|                   Processor Status (PS)                       | :SP
|                                                               |
+---------------------------------------------------------------+
|                          Virtual                              |
|                   Address of Privileged                       |
|                        Instruction                            |
+---------------------------------------------------------------+
```

Figure 6-11: Privileged Instruction Fault Exception Frame

Note that the faulting instruction is not pushed on the stack. If the instruction was contained in a page that was executable, but not readable in the current mode, then pushing the faulting instruction would provide information normally not available to the current mode.

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 current mode and push the following information on the Current Mode stack:

```
 3
 1                                                           0
+-------------------------------------------------------------+
|                                                             |
|                            Zero                             | :SP
|                                                             |
+-------------------------------------------------------------+
|                                                             |
|                    Faulting  Instruction                    |
|                                                             |
+-------------------------------------------------------------+
|                                                             |
|                    Processor Status (PS)                    |
|                                                             |
+-------------------------------------------------------------+
|                           Virtual                           |
|                    Address of Reserved                      |
|                         Instruction                         |
+-------------------------------------------------------------+
```

Figure 6-12: 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, in the current mode.

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).

Vector Enable faults are initiated in Kernel mode and push the following information on the Kernel stack:

```
 3
 1                                                            0
+-------------------------------------------------------------+
|                                                             |
|                   Processor Status (PS)                     | :SP
|                                                             |
+-------------------------------------------------------------+
|                         Virtual                             |
|                    Address of Vector                        |
|                       Instruction                           |
+-------------------------------------------------------------+
```

Figure 6-13: 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.

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 load or store operation.

Memory management faults are generated in Kernel mode and push the following information on the Kernel stack:

```
 3
 1                                                          1 0
+-----------------------------------------------------------+-+
|                          Related                            |
|                    Virtual Address in                       | :SP
|                           Page                              |
+-----------------------------------------------------------+-+
|                                                           |R|
|                           Zero                            |/|
|                                                           |W|
+-----------------------------------------------------------+-+
|                                                             |
|                   Processor Status (PS)                     |
|                                                             |
+-------------------------------------------------------------+
|                         Virtual                             |
|                     Address of Next                         |
|                       Instruction                           |
+-------------------------------------------------------------+
```

Figure 6-14: 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).

If the memory management fault was caused by a scalar load or store instruction, the virtual address of the next instruction is that of the scalar load or store instruction itself. However, if the memory managment fault was caused by a vector load or store instruction, then the virtual address of the next instruction is that of the next instruction that would have been executed had the faulting condition not been present.

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 access mode changes and protected entry to inner modes, 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 capabilities and watchpoints, 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:

```
 3
 1                                                        0
+----------------------------------------------------------+
|                        Number                            |
|                          of                              | :SP
|                     Bytes Pushed                         |
+----------------------------------------------------------+
|                                                          |
|                         Zero                             |
|                                                          |
+----------------------------------------------------------+
                           .
                           .
                           .
                    An even number of
                     implementation
                        specific
                       longwords
                           .
                           .
                           .
+----------------------------------------------------------+
|                                                          |
|                  Processor Status (PS)                   |
|                                                          |
+----------------------------------------------------------+
|                        Virtual                           |
|                    Address of Next                       |
|                      Instruction                         |
+----------------------------------------------------------+
```

Figure 6-15: Machine Check Abort Exception Frame

Implementation-specific information is pushed on the stack as longwords. An even number of informational longwords are pushed in order to keep the stack quadword aligned. A zero longword followed by the number of parameter bytes are then pushed. The number of parameter bytes does not include the processor state (PS and PC), but does include the count and zero 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 executing the appropriate action.

If a second Machine Check is detected while Epicode is initiating a machine check exception, a Double Error halt is generated and the processor enters the restart sequence; see Chapter 11, System Bootstrapping and Console.

6.4.7 Stack Alignment Abort

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 increments of quadwords.

Epicode pushes and pops information to/from the target/source stack on the initiation of exceptions and interrupts and during an REI instruction. Epicode always pushes and 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.\

A Stack Alignment abort occurs during the initiation of an exception when Epicode attempts to push information on the User, Supervisor, or Executive stack and the stack is not quadword aligned, or during the execution of an REI instruction when Epicode attempts to remove the processor state from the User, Supervisor, or Executive 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 above.

Stack Alignment aborts are initiated in Kernel mode and push the following information on the Kernel stack:

```
 3
 1                                                        0
+----------------------------------------------------------+
|                                                          |
|                 Processor Status (PS)                    | :SP
|                                                          |
+----------------------------------------------------------+
|                        Virtual                           |
|                    Address  of Next                      |
|                      Instruction                         |
+----------------------------------------------------------+
```

Figure 6-16: Stack Alignment Abort Exception Frame

6.4.8 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 faults and alignment aborts, however, must be reported before the vector operation completes and, for memory management faults, sufficient state must be saved so the appropriate vector load/store operation can be continued after the fault condition has been corrected.

One or more vector load/store operations may be in progress simultaneously, and therefore it is possible for an arithmetic exception condition to be present concurrently with one or more vector memory management fault and/or alignment abort conditions.

Memory management faults and alignment aborts occurring on vector load/store instructions push the following additional information on the Kernel stack prior to pushing the processor state:

```
 3 3   2 2 2         2 2 1 1 1         1 1
 1 0   8 7 6         1 0 9 8 7         2 1        6 5        0
+-+-----+-+-----------+-+-+-+-----------+-----------+-----------+
|V|  E  |L|           |S|O|D|           |           |    SRC    |
|F|  T  |V|   Zero    |T|P|T|    ELT    |    CNT    |    or     |
|S|  Y  |F|           |R|R|Y|           |           |    DST    |
+-+-----+-+-----------+-+-+-+-----------+-----------+-----------+
|                          Related                              |
|                     Virtual  Address in                       |
|                           Page                                |
+---------------------------------------------------------------+
|                          Vector                               |
|                           Base                                |
|                          Address                              |
+---------------------------------------------------------------+
|                          Stride                               |
|                            or                                 |
|                Index Vector Register Number                   |
+---------------------------------------------------------------+
```

Figure 6-17: Vector Exception Information Frame

Bits	Description
5:0	Vector Register (SRC/DST) - The source (store) or destination (load) vector register number.
11:6	Remaining Count (CNT) - Count of the number of elements remaining to be loaded or stored to/from the vector register.
17:12	Next Element (ELT) - The index of the next element in the vector register to be loaded or stored.
18	Datatype (DTY) - When clear, the data type is longword; when set, the data type is quadword.
19	Operation Type (OPR) - When clear, the operation is a load; when set, the operation is a store.
20	Indexing Type (STR) - When clear, the operation is stride based; when set, the operation is a scatter/gather operation.
27	Last Vector Frame (LVF) - This bit indicates whether another

vector exception information frame immediately precedes this one on the stack. When set, this is the last vector frame; when clear, there is another vector frame preceding this one on the stack.

30:28	Exception Type (ETY) - The type of exception described by this vector frame. Exception types are: 0 - Access Violation fault 1 - Fault On Read fault 2 - Fault On Write fault 3 - Translation Not Valid fault 4 - Vector Alignment abort 5 - Instruction Pending
31	Vector Frame Status (VFS) - This bit indicates whether the information in this vector frame has been processed. This bit is cleared when the vector frame is pushed on the stack and set when Epicode has built a corresponding memory management fault frame.

The above information is pushed for each concurrent vector load/store operation that has encountered a memory management fault or alignment abort condition. It is used later by the REI instruction to determine whether an exception should be initiated or the vector operation should be continued.

\This information is somewhat analogous to the First Part Done information saved in the general registers on VAX for string and decimal instructions.\

The vector base address may be the actual base address of the vector (e.g., vector gather and scatter instructions and other vector loads and stores that receive an exception on the first element) or the actual address of the data that caused the exception condition (e.g., an exception condition occurring in the middle of a stride-based vector load or store instruction).

Arithmetic exceptions that occur on vector instructions are reported as described in Section 6.4.1 provided no vector exception information frames have been pushed on the Kernel stack.

If any vector exception information frames have been pushed on the Kernel stack, then the current PC followed by a PS with Vector Exception Frame (VEF) set are pushed on the Kernel stack and either an arithmetic trap, memory management fault, or Vector Alignment abort is initiated.

If an arithmetic exception condition has occurred concurrently, the parameters described in Section 6.4.1 are pushed on the Kernel stack and an arithmetic exception is initiated. Later, when the exception has been processed, an attempt to continue execution with an REI instruction will encounter a PS with VEF set; see Section 6.4.8.1 below.

If no arithmetic exception has occurred, either a memory management fault or Vector Alignment abort exception frame is pushed on the Kernel stack; see Sections 6.4.2.2 and 6.4.5 above. If a memory management frame is pushed, then the Vector Frame Status (VFS) bit is also set. The appropriate exception is then initiated. After the operating system has processed the exception, an attempt to continue execution with an REI will encounter a PS with VEF set.

6.4.8.1 Vector Exception Continuation

Execution of an REI instruction with Vector Exception Frame (VEF) set requires special processing by Epicode. When this situation arises, Epicode must scan the vector exception information frames immediately preceding the processor state on the Kernel stack to determine whether another vector exception should be initiated or whether one or more vector load/store operations should be continued.

Epicode successively examines each vector exception information frame until a frame with Vector Frame Status (VFS) clear (unprocessed) or Last Vector Frame (LVF) set is encountered.

Each vector exception information frame must be checked for validity since it is possible for unprivileged code to forge such a frame and execute an REI. If an invalid frame is detected, an Illegal Operand fault is initiated.

The logic required to check for this condition is:

```
tmp <- CNT
IF tmp EQ 0 THEN
    tmp <- 64
IF {tmp + ELT} GT 64 THEN
    {initiate Illegal Operand fault}
```

If a vector exception information frame with VFS clear is encountered, and the exception type is not Instruction Pending (ETY NE 5), either a memory management fault or Vector Alignment abort exception frame is pushed on the Kernel stack. If a memory management frame is pushed, then the Vector Frame Status (VFS) bit is also set. The appropriate exception is then initiated. Note that since the VFS bit is not set for Vector Alignment aborts, any attempt to continue with an REI will result in the generation of another Vector Alignment abort with identical parameters.

If a vector exception information frame is encountered with Last Vector Frame (LVF) set, all vector exceptions occurring as the result of vector load/store operations have been processed and the respective operations should be continued. For each vector exception information frame, Epicode restarts the vector load/store operation in an implementation-dependent manner.

\An implementation may choose to restart vector load and store operations from the beginning or continue from the point of the memory management problem. System software must guarantee a minimum available working set of 67 pages.\

The PC and PS are then restored, the vector frames are removed from the Kernel stack, and instruction execution continues.

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 memory management faults, which take precedence over all other faults, which take precedence over interrupts.

Thus the priority of initiation is:

1. Arithmetic traps

2. Vector Alignment and memory management exceptions

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 longword 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.

```
 3
 1                                                      2 1 0
+------------------------------------------------------+---+
|                       Virtual                        | S |
|                      Address  of                     | B |
|                    Service Routine                   | Z |
+------------------------------------------------------+---+
```

Figure 6-18: System Control Block Vector

If Epicode reads a vector 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	Mode	Number Longwds	Notes
00	Unused				Reserved to DIGITAL.
04	Machine Check	Abort	Kernel	*	Implementation specific number of longwords pushed on stack.
08	Fault On Bit	Fault	Current	4	Faulting instruction pushed on stack.
0C	Vector Align	Abort	Kernel	4	Virtual address of reference is pushed on stack.
10	Scalar Align	Fault	Current	4	Faulting instruction and virtual address of reference pushed on stack.
14	Access Violat	Fault	Kernel	4	Virtual address and type of reference pushed on stack.
18	Trans Not Valid	Fault	Kernel	4	Virtual address and type of reference pushed on stack.
1C	Fault On Exec	Fault	Kernel	4	Virtual address and type of reference pushed on stack.
20	Fault On Read	Fault	Kernel	4	Virtual address and type of reference pushed on stack.
24	Fault On Write	Fault	Kernel	4	Virtual address and type of reference pushed on stack.
28	Arithmetic Trap	Trap	Kernel	6	Exception summary and vector and scalar register write masks pushed on stack.

Table 6-1: System Control Block Vector Assignments (Continued)

Vector (hex)	Name	Type	Mode	Number Longwds	Notes
2C	Interval Clock	Int	Kernel	2	IPL is raised to 5.
30	Interproc Int	Int	Kernel	2	IPL is raised to 6.
34	Software Lvl 1	Int	Kernel	2	IPL is raised to 1.
38	Software Lvl 2	Int	Kernel	2	IPL is raised to 2.
3C	Software Lvl 3	Int	Kernel	2	IPL is raised to 3.
40	AST Interrupt	Int	Kernel	2	IPL is raised to 1.
44	Priv Instruct	Fault	Current	2	
48	Illegal Operand	Fault	Current	4	Faulting instruction pushed on stack.
4C	Stack Alignment	Abort	Kernel	2	
50	Breakpoint	Fault	Kernel	2	
54	Bug Check	Fault	Kernel	2	
58	Reserved Opcode	Fault	Current	4	Faulting instruction pushed on stack.
5C	Power Recovery	Int	Kernel	2	IPL is raised to 7.
60	Console Receive	Int	Kernel	2	IPL is raised to 4.
64	Console Transmt	Int	Kernel	2	IPL is raised to 4.
68	Vector Enable	Fault	Kernel	2	
6C-3FC	Unused				Reserved to DIGITAL.
400-7FC	I/O Proc Int	Int	Kernel	2	I/O port and I/O processor specific interrupt vectors. IPL raised to 4 or 5.

6.7 STACKS

At any point in time the processor is in one of four modes (Kernel, Executive, Supervisor, or User). There is a stack pointer associated with each of these four modes. 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 four architecturally defined stack pointers is currently in use, as follows:

Mode	Stack
0	Kernel (KSP)
1	Executive (ESP)
2	Supervisor (SSP)
3	User (USP)

6.7.1 Stack Writability

In response to various exceptions and interrupts, Epicode pushes information on either the Kernel or Current Mode stack. Epicode may write this information without first probing to ensure that all such writes to the target stack will succeed. If a memory management exception occurs while pushing information, the appropriate memory management fault is generated rather than the original exception.

6.7.2 Stack Residency

The User, Supervisor, and Executive stacks do 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. If Epicode attempts to push on a stack that is not quadword aligned, a Stack Alignment abort is generated. It is the responsibility of software to ensure that stacks are quadword aligned.

Epicode pushes parameters on various stacks 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.

Once the service mode and stack have been determined, Epicode then attempts to push the PC followed by the PS, and in the case of exceptions, other parameters if required, on the target 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 the target stack is the Kernel stack and 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.

If the target stack is User, Supervisor, or Executive and the stack is unaligned, a Stack Alignment abort is initiated.

If the target stack is User, Supervisor, or Executive and a Translation Not Valid, Access Violation, or Fault On Write condition exists, the exception is turned into the corresponding memory management exception, with the PC and PS of the original fault and the virtual address of the problem in the target mode stack.

6.7.5 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 in-progress instructions to complete}
    IPR_SP[PS<CM>] <- SP
    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>
            new_mode <- 0
            new_sp <- KSP
            FOR i <- 1 TO {number of exceptions}
                BEGIN
                PUSH(stride[i], base[i])
                PUSH(virtual[i], reg_data[i])
                END
            tmp <- PS
            tmp<VEF> <- 1
            PUSH(PC, tmp)
            IF {arithmetic exception} THEN
                BEGIN
                PUSH(write_mask_R63_R32, write_mask_R31_R0)
                PUSH(write_Mask_V15_V0, summary)
                vector <- {arithmetic exception SCB offset}
                END
            ELSE
                BEGIN
                IF reg_data[1]<ETY> EQ {vector alignment abort} THEN
                    BEGIN
                    PUSH(0, virtual[1])
                    vector <- {vector alignment exception SCB offset}
                    END
                ELSE
                    BEGIN
                    (new_sp + 8)<VFS> <- 1
                    tmp <- ZEXT(reg_data[1]<OPR>)
                    PUSH(tmp, virtual[1])
                    vector <- {memory management exception SCB offset}
                    END
                END
            END
        ELSE
            BEGIN
            IF {machine check} THEN
                BEGIN
                new_ipl <- 7
                new_mode <- 0
                new_sp <- KSP
                END
            ELSE
                BEGIN
                new_ipl <- PS<IPL>
                new_mode <- {target mode of exception}
                new_sp <- IPR_SP[new_mode]
                END
            PUSH(PC, PS)
            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 <- {exception SCB offset}
            END
        END
    ELSE
        BEGIN
        new_ipl <- {interrupt source IPL}
        new_mode <- 0
        new_sp <- KSP
        PUSH(PC, PS)
        vector <- {interrupt SCB offset}
        END
    PS<CM> <- new_mode
    PS<IPL> <- new_ipl
    SP <- new_sp
    PC <- (SCBB + vector)
    END
GOTO check_for_exception_or_interrupt
```

```
PROCEDURE PUSH(first, last)

IF new_sp<2:0> NE 0 THEN
    BEGIN
    IF new_mode EQ 0 THEN
        {initiate kernel stack not valid halt}
    ELSE
        BEGIN
        new_mode <- 0
        new_sp <- KSP
        PUSH(PC, PS)
        KSP <- new_sp
        PS<CM> <- 0
        PC <- (SCBB + {stack alignment abort SCB offset})
        GOTO check_for_exception_or_interrupt
        END
    END
ELSE
    BEGIN
    IF ACCESS(new_sp - 8, new_mode) THEN
        BEGIN
        (new_sp - 4) <- first
        (new_sp - 8) <- last
        new_sp <- new_sp - 8
        RETURN
        END
    ELSE
        BEGIN
        IF new_mode EQ 0 THEN
            {initiate kernel stack not valid halt}
        ELSE
            BEGIN
            tmp <- new_sp
            new_mode <- 0
            new_sp <- KSP
            PUSH(PC, PS)
            PUSH(1, tmp)
            KSP <- new_sp
            PS<CM> <- 0
            PC <- (SCBB + {memory management SCB offset})
            GOTO check_for_exception_or_interrupt
            END
        END
    END
END
```

6.7.6 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 processors, 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 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 totally be implemented 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.

6.7.6.1 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 four 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}
```

6.7.6.2 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}
```

6.7.6.3 Return From Exception Or Interrupt

The Return from Exception or Interrupt instruction (see Chapter 4, Instruction Descriptions, Page 4-85) 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(1110(bin), PS<CM>)
IF {{tmp AND ASTEN AND ASTSR}<3:0> NE 0} AND {PS<IPL> EQ 0} THEN
    {initiate AST interrupt at IPL 1}
```

6.7.6.4 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}
```

6.7.6.5 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}
```

6.7.7 Processor State Transition Table

Table 6-2: Processor State Transitions

Initial State	Final State: User IPL=0	Super IPL=0	Exec IPL=0	Kernel IPL=0	Kernel IPL>0	Program Halt
USER IPL=0		NP	NP	Exc	Int Exc SWASTEN	NP
SUPER IPL=0	REI*		NP	Exc	Int Exc SWASTEN	NP
EXEC IPL=0	REI*	REI*		Exc	Int Exc SWASTEN	NP
KERNEL IPL=0	REI*	REI*	REI*		REI SWIPL Int Exc MTPR* SWASTEN	HALT
KERNEL IPL>0	REI*	REI*	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 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. Changed Bug Check to a fault so the only traps would be arithmetic.

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:

- o 64 scalar registers
- o 16 vector registers
- o Vector Length register (VL)
- o Vector Count register (VC)
- o Vector Mask register (VM)
- o Processor Status (PS)
- o Program Counter (PC)
- o 4 stack pointers
- o Asynchronous System Trap Enable register (ASTEN)
- o Asynchronous System Trap Summary Register (ASTSR)
- o Process Page Table Base Register (PTBR)
- o Address Space Number (ASN)

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; 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 the loading of 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 4-93) 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).

```
 3                            1 1
 1                            6 5            8 7    4 3    0
+-----------------------------+--------------+------+------+
|                                                          |
|               Kernel Stack Pointer (KSP)                 | :HWPCB
|                                                          |
+-----------------------------+--------------+------+------+
|                                                          |
|              Executive Stack Pointer (ESP)               | +4
|                                                          |
+-----------------------------+--------------+------+------+
|                                                          |
|              Supervisor Stack Pointer (SSP)              | +8
|                                                          |
+-----------------------------+--------------+------+------+
|                                                          |
|                 User Stack Pointer (USP)                 | +12
|                                                          |
+-----------------------------+--------------+------+------+
|                             |              |      |      |
|  Address Space Number (ASN) |      SBZ     |ASTEN |ASTSR | +16
|                             |              |      |      |
+-----------------------------+--------------+------+------+
|                                                          |
|             Page Table Base Register (PTBR)              | +20
|                                                          |
+----------------------------------------------------------+
```

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 four processor access modes. When the bit corresponding to an access mode is set, ASTs for that mode are enabled. The AST enable bit for an access mode may be changed by executing a Swap AST Enable instruction (SWASTEN); see Chapter 4, Instruction Descriptions, Page 4-87.

The AST Summary Register (ASTSR) contains a pending bit for each of the four processor access modes. When the bit corresponding to an access mode is set, an AST is pending for that mode. The AST pending bit for an access mode may be set by requesting an AST for the respective mode.

Kernel mode software may request an AST for a particular access 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 any access mode that is less than or equal to PS<CM> (i.e., an equal or more privileged access mode), an AST interrupt is initiated at IPL 1. ASTs that are pending and enabled for less privileged access modes are not allowed to interrupt execution in more privileged access modes.

7.3.1 A Software Model For AST Processing

It is intended that ASTs represent a single level of interrupt for each of the four processor access 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 access modes for which an AST is currently in progress and not request further ASTs for these access 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 access modes for which an AST is currently active. When an AST is delivered to a particular access 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 access 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 access 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 access 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 access mode, an AST is requested for the corresponding access 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 access mode, an AST is requested by setting the bit corresponding to the specified access 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 any access mode that is less than or equal to PS<CM> (i.e., an equal or more privileged access 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 an access mode that is less than or equal to the current mode of the saved PS (an access mode that is equal to or more privileged than the previous access 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 access 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 access 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 access mode. If another AST has been queued, an AST is requested by executing an MTPR ASTRR specifying the appropriate access 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 is not context switchable (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
    WRVL    R0                   ; set vector length to 64 elements
    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
    WRVL    R0                      ; set vector length to 64 elements
    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 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 4-90 and 4-91.

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 that produces 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	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		
Stack Pointer Registers					
Executive Stack Pointer	ESP	R/W	address		
Supervisor Stack Pointer	SSP	R/W	address		
User Stack Pointer	USP	R/W	address		
Interval Clock Int. Enable	ICIE	R/W	enable		
Interprocessor Int. Enable	IPIE	R/W	enable		
Interprocessor Int. Request	IPIR	W	number		
Privileged Context Block Base	PCBB	R	address	address	
Processor Base Register	PRBR	R/W	value		
Processor Serial Number	PRSN	R	serial		
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		
Trans. Buffer Check	TBCHK	R	number	address	status
Trans. Buffer Invalidate ASN	TBIASN	W	number		
Trans. Buffer Invalidate Single	TBIS	W	number	address	
Time Of Year	TOY	R/W	time	time	
Who-Am-I	WHAMI	R	number		

Address Space Number (ASN)

Access:

Read

Operation:

R4 <- ZEXT(ASN<15:0>)

Value at System Initialization:

Zero

Format:

```
 3                             1 1                             0
 1                             6 5                             
+-------------------------------+-------------------------------+
|                               |                               |
|              RAZ              |     Address Space Number      | :R4
|                               |                               |
+-------------------------------+-------------------------------+
```

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 4-93 and Chapter 7, Process Structure.

AST Enable (ASTEN)

Access:

Read

Operation:

R4 <- ZEXT(ASTEN<3:0>)

Value at System Initialization:

Zero

Format:

```
 3
 1                                                      4 3 2 1 0
+-------------------------------------------------------+-+-+-+-+
|                                                       |U|S|E|K|
|                         RAZ                           |E|E|E|E| :R4
|                                                       |N|N|N|N|
+-------------------------------------------------------+-+-+-+-+
```

Figure 8-2: AST Enable Register (ASTEN)

Description:

The AST Enable register records the AST enable state for each of the modes: Kernel (KEN), Executive (EEN), Supervisor (SEN), and User (UEN). The current AST enable state may be read by executing an MFPR instruction specifying ASTEN.

As processes are scheduled for execution, the state of the AST enables 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 access mode. See Chapter 4, Instruction Descriptions, Pages 4-93 and 4-87, and Chapter 7, Process Structure.

AST Request Register (ASTRR)

Access:

Write

Operation:

ASTRR <- R4<1:0>

Value at System Initialization:

Not Applicable

Format:

```
 3
 1                                                     2 1 0
+-----------------------------------------------------+---+
|                                                     | M |
|                        IGN                          | O | :R4
|                                                     | D |
+-----------------------------------------------------+---+
```

Figure 8-3: AST Request Register (ASTRR)

Description:

An AST may be requested for a particular access mode by executing an MTPR instruction specifying ASTRR. Access 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 access 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 6.7.6.

AST Summary Register (ASTSR)

Access:

Read

Operation:

R4 <- ZEXT(ASTSR<3:0>)

Value at System Initialization:

Zero

Format:

```
 3
 1                                                 4 3 2 1 0
+-------------------------------------------------+-+-+-+-+
|                                                 |U|S|E|K|
|                      RAZ                        |P|P|P|P|  :R4
|                                                 |D|D|D|D|
+-------------------------------------------------+-+-+-+-+
```

Figure 8-4: AST Summary Register (ASTSR)

Description:

The AST Summary Register records the AST pending state for each of the modes: Kernel (KPD), Executive (EPD), Supervisor (SPD), 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 4-93 and Chapter 7, Process Structure.

MTPR ASTRR requests an AST at a particular access 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 access mode bit in ASTSR is cleared; see Chapter 6, Exceptions and Interrupts, Section 6.7.6.

Console Receive Control Status (CRCS)

Access:

Read/Write

Operation:

```
        R4 <- CRCS                          ! Read

        CRCS<0> <- R4<0>                    ! Write
```

Value at System Initialization:

Zero

Format:

```
 3 3
 1 0                                                         1 0
+-+-----------------------------------------------------------+-+
|R|                                                         | | |
|D|                       IGN/RAZ                           |I| :R4
|Y|                                                         |E|
+-+-----------------------------------------------------------+-+
```

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, a value of 1 enables console receive interrupts and a value of 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 when RDY is set, a console receive interrupt is generated when conditions permit.

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:

```
 3 3
 1 0                                               8 7             0
+-+-----------------------------------------------+----------------+
|E|                                               |                |
|R|                       RAZ                     |   Character    | :R4
|R|                                               |                |
+-+-----------------------------------------------+----------------+
```

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:

Zero

Format:

```
 3 3
 1 0                                                         1 0
+-+-----------------------------------------------------------+-+
|D|                                                           | |
|O|                      IGN/RAZ                              |I| :R4
|N|                                                           |E|
+-+-----------------------------------------------------------+-+
```

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, a value of 1 enables console transmit interrupts and a value of 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 when DON is set, a console transmit interrupt is generated when conditions permit.

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:

```
 3
 1                                          8 7            0
+--------------------------------------------+--------------+
|                                            |              |
|                    IGN                     |  Character   | :R4
|                                            |              |
+--------------------------------------------+--------------+
```

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>.

Stack Pointer Registers (ESP, SSP, USP)

Access:

Read/Write

Operation:

```
R4 <- xSP                          ! Read

xSP <- R4                          ! Write
```

Value at System Initialization:

Undefined

Format:

```
 3
 1                                                               0
+-----------------------------------------------------------------+
|                                                                 |
|                         Stack Address                           | :R4
|                                                                 |
+-----------------------------------------------------------------+
```

Figure 8-9: Stack Pointer Registers (ESP, SSP, USP)

Description:

These registers allow the stack pointers for the access modes Executive (ESP), Supervisor (SSP), and User (USP) to be read and written via MFPR and MTPR instructions that specify the corresponding stack pointer.

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 four stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapter 4, Instruction Descriptions, Page 4-93 and Chapter 7, Process Structure.

Stack pointers must be quadword aligned or a stack alignment exception may occur. An unaligned Executive, Supervisor, or User stack results in a Stack Alignment abort exception. An unaligned Kernel stack results in a Kernel Stack Not Valid halt. See Chapter 6, Exceptions and Interrupts, Section 6.4.7.

Interval Clock Interrupt Enable (ICIE)

Access:

Read/Write

Operation:

```
        R4 <- ZEXT(ICIE<0>)              ! Read

        ICIE <- R4<0>                    ! Write
```

Value at System Initialization:

Zero

Format:

```
 3
 1                                                            1 0
+-------------------------------------------------------------+-+
|                                                             | |
|                          IGN/RAZ                            |I|  :R4
|                                                             |E|
+-------------------------------------------------------------+-+
```

Figure 8-10: Interval Clock Interrupt Enable Register (ICIE)

Description:

The Interval Clock provides the capability to regularly interrupt the processor at 10 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, a value of 1 enables clock interrupts and a value of 0 disables interrupts. After enabling Interval Clock interrupts, the first interrupt may occur in less than 10 milliseconds.

Interval Clock interrupts are initiated at IPL 5; see Chapter 6, Exceptions and Interrupts, Section 6.3.5.

Interprocessor Interrupt Enable (IPIE)

Access:

Read/Write

Operation:

```
R4 <- ZEXT(IPIE<0>)            ! Read

IPIE <- R4<0>                  ! Write
```

Value at System Initialization:

Zero

Format:

```
 3
 1                                                           1 0
+-------------------------------------------------------------+-+
|                                                             | |
|                         IGN/RAZ                             |I|  :R4
|                                                             |E|
+-------------------------------------------------------------+-+
```

Figure 8-11: Interprocessor Interrupt Enable Register (IPIE)

Description:

The PRISM architecture provides the capability for one processor to interrupt another processor via an IPR; see Page 8-14. 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, a value of 1 enables interprocessor interrupts and a value of 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 6.3.6.1.

Interprocessor Interrupt Request (IPIR)

Access:

Write

Operation:

IPIR <- R4

Value at System Initialization:

Not applicable

Format:

```
 3
 1                                                              0
+----------------------------------------------------------------+
|                                                                |
|                        Processor Number                        | :R4
|                                                                |
+----------------------------------------------------------------+
```

Figure 8-12: 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 8-13.

If the target processor is the same as the current processor, and proper enabling conditions are present, an interprocessor interrupt is initiated prior to issuing the next instruction; see Chapter 6, Exceptions and Interrupts, Sections 6.3.6.2 and 6.7.6.

Privileged Context Block Base (PCBB)

Access:

Read

Operation:

QR4 <- ZEXT(PCBB)

Value at System Initialization:

See Chapter 11, System Bootstrapping and Console.

Format:

```
 3                                    1 1
 1                                    3 2                     0
+---------------------------------------+-----------------------+
|                                                               |
|                    Physical Address<31:0>                     | :R4
|                                                               |
+---------------------------------------+-----------------------+
|                                       |                       |
|                  RAZ                  | Physical Address<44:32>| :R5
|                                       |                       |
+---------------------------------------+-----------------------+
```

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 4-93 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:

```
 3
 1                                                             0
+---------------------------------------------------------------+
|                                                               |
|                 Operating System Dependent Value              | :R4
|                                                               |
+---------------------------------------------------------------+
```

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.

Processor Serial Number (PRSN)

Access:

Read

Operation:

```
IF {implemented} THEN
    R4 <- PRSN
ELSE
    R4 <- 0
```

Value at System Initialization:

Processor serial number or zero

Format:

```
 3
 1                                                               0
+-----------------------------------------------------------------+
|                                                                 |
|                          Serial Number                          | :R4
|                                                                 |
+-----------------------------------------------------------------+
```

Figure 8-15: Processor Serial Number Register (PRSN)

Description:

The Processor Serial Number register provides access to the processor serial number by executing an MFPR instruction specifying PRSN.

Implementation of serial numbers is optional. If implemented, the serial number is returned. Otherwise, a value of zero is returned (zero is an invalid serial number).

Page Table Base Register (PTBR)

Access:

Read

Operation:

R4 <- PTBR

Value at System Initialization:

See Chapter 11, System Bootstrapping and Console

Format:

```
 3
 1                                                                0
+-----------------------------------------------------------------+
|                                                                 |
|                        Page Frame Number                        | :R4
|                                                                 |
+-----------------------------------------------------------------+
```

Figure 8-16: 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 4-93 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:

        See Chapter 11, System Bootstrapping and Console

Format:

```
 3                                    1 1
 1                                    3 2                    0
+-------------------------------------+-----------------------+
|                                                             |
|                    Physical Address<31:0>                   | :R4
|                                                             |
+-------------------------------------+-----------------------+
|                                     |                       |
|               IGN/RAZ               | Physical Address<44:32>| :R5
|                                     |                       |
+-------------------------------------+-----------------------+
```

Figure 8-17: 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, or an UNDEFINED operation may result.

System Identification (SID)

Access:

Read

Operation:

QR4 <- SID

Value at System Initialization:

System Identification

Format:

```
 3              2 2              1 1              8 7             0
 1              4 3              6 5
+----------------+----------------+----------------+----------------+
|                |                |                |                |
|   Processor    |    Hardware    |    Epicode     |     System     | :R4
|     Type       |    Revision    |    Revision    |      Type      |
+----------------+----------------+----------------+----------------+
|                                                                   |
|                  Implementation Dependent Data                    | :R5
|                                                                   |
+-------------------------------------------------------------------+
```

Figure 8-18: 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:

```
 3
 1                                                   2 1 0
+---------------------------------------------------+---+
|                                                   | L |
|                       IGN                         | V |  :R4
|                                                   | L |
+---------------------------------------------------+---+
```

Figure 8-19: 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 8-22. 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 6.7.6.

Software Interrupt Summary Register (SISR)

Access:

Read

Operation:

R4 <- ZEXT(SISR<3:0>)

Value at System Initialization:

Zero

Format:

```
 3
 1                                                    4 3 2 1 0
+----------------------------------------------------+-+-+-+-+
|                                                    |I|I|I|R|
|                      RAZ                           |R|R|R|A|  :R4
|                                                    |3|2|1|Z|
+----------------------------------------------------+-+-+-+-+
```

Figure 8-20: 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 8-21.

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 6.7.6.

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:

```
 3                             1 1
 1                             6 5                             0
+-------------------------------+-------------------------------+
|                               |                               |
|              IGN              |      Address Space Number     |  :R4
|                               |                               |
+-------------------------------+-------------------------------+
|                                                               |
|                        Virtual Address                        |  :R5
|                                                               |
+---------------------------------------------------------------+

 3 3
 1 0                                                         1 0
+-+-----------------------------------------------------------+-+
|I|                                                           |P|
|M|                            RAZ                            |R|  :R6
|P|                                                           |S|
+-+-----------------------------------------------------------+-+
```

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 By ASN (TBIASN)

Access:

Write

Operation:

{invalidate all TB entries with ASN EQ R4<15:0>}

Value at System Initialization:

Not applicable

Format:

```
 3                             1 1
 1                             6 5                              0
+-------------------------------+-------------------------------+
|                               |                               |
|              IGN              |     Address Space Number      | :R4
|                               |                               |
+-------------------------------+-------------------------------+
```

Figure 8-22: Translation Buffer Invalidate by ASN Register (TBIASN)

Description:

The Translation Buffer Invalidate by ASN register provides the capability to invalidate all entries in the Translation Buffer for a particular ASN by executing an MTPR instruction specifying TBIASN; see Chapter 5, Memory Management.

If ASNs are not implemented, a write to this register invalidates all Translation Buffer entries which do not have the Address Space Match (ASM) bit set; see Chapter 5, Memory Management, Section 5.5.

Translation Buffer Invalidate Single (TBIS)

Access:

        Write

Operation:

        {Invalidate single TB entry using R4<15:0>, R5}

Value at System Initialization:

        Not applicable

Format:

```
 3                            1 1                            0
 1                            6 5                            0
+-------------------------------+-------------------------------+
|                               |                               |
|              IGN              |     Address Space Number      | :R4
|                               |                               |
+-------------------------------+-------------------------------+
|                                                               |
|                        Virtual Address                        | :R5
|                                                               |
+---------------------------------------------------------------+
```

Figure 8-23: 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:

```
QR4 <- TOY                    ! Read

TOY <- QR4                    ! Write
```

Value at System Initialization:

Correct time or invalid time indication

Format:

```
 3 3   2 2     2 2     2 1     1 1     1 1
 1 0   8 7     4 3     0 9     6 5     2 1     8 7     4 3     0
+-+-----+-------+-------+-------+-------+-------+-------+-------+
|       |       |       |       |       |       |       |       |
|  D1   |  D0   |  H1   |  H0   |  MI1  |  MI0  |  S1   |  S0   | :R4
|       |       |       |       |       |       |       |       |
+-+-----+-------+-------+-------+-------+-------+-------+-------+
| |             |       |       |       |       |       |       |
|V|   IGN/RAZ   |  Y3   |  Y2   |  Y1   |  Y0   |  MO1  |  MO0  | :R5
| |             |       |       |       |       |       |       |
+-+-------------+-------+-------+-------+-------+-------+-------+
```

Figure 8-24: 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).

TOY records the time in Binary Coded Decimal (BCD) format and is updated once a second. TOY has an accuracy of .0025% (approximately 65 seconds per month) and is battery backed up. Once TOY is written, the time will remain valid until backup power is lost.

When TOY is read, a valid indication is returned in bit 31 of the high-order longword. If bit 31 is set, the contents of TOY are valid. Otherwise, backup power has been lost and the contents of TOY are invalid.

When TOY is written, the time base used is operating system dependent (e.g., Greenwich Mean Time, Universal Time, daylight savings time, standard time, etc.).

TOY encoding is:

    4 BCD digits of year (Y3,Y2,Y1,Y0)
    2 BCD digits of month (MO1,MO2)
    2 BCD digits of day (D1,D0)
    2 BCD digits of hour (H1,H0)
    2 BCD digits of minutes (MI1,MI0)
    2 BCD digits of seconds (S1,S0)

Who-Am-I (WHAMI)

Access:

Read

Operation:

R4 <- WHAMI

Value at System Initialization:

Processor number

Format:

```
 3
 1                                                              0
+----------------------------------------------------------------+
|                                                                |
|                       Processor Number                         | :R4
|                                                                |
+----------------------------------------------------------------+
```

Figure 8-25: 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 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 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:

- o Data sharing and synchronization
- o Separation of procedures and data
- o Translation Buffer
- o Caches
- o Stacks

To meet the requirements of the PRISM architecture, software and hardware implementors have to take these issues into consideration.

9.2 DATA SHARING AND SYNCHRONIZATION

The memory system must be implemented such that the granularity of access for independent modification is a quadword or less. Note that this does not imply a maximum reference size of one quadword, but only that independent accesses to adjacent quadwords produce the same results regardless of the order of execution. Systems may choose to do masked writes (less than quadword) in the cache by reading the needed quadword from memory, merging it in the cache, and then writing the quadword back to memory, thereby only supporting quadword writes to the main memory system.

NOTE

\A system may also build a VAX-style memory system with masked writes to the main memory. The quadword granularity of sharing is being included to allow simpler and cheaper systems to be built. But since some PRISM systems will use a common memory system with a given VAX implementation we are not going to disallow reusing the existing memory subsystems.\

For example, suppose locations 0 and 8 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 8. Then, regardless of the order of execution, including effectively simultaneous execution, the final contents 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.

Access to explicitly shared data that may be written must be synchronized by the programmer. Before accessing shared writable data, the programmer must acquire control of the data structure. The interlock instructions (RMAQI, and RMAQIP) are provided to allow the programmer to control "interlocked" access to a control variable. These interlocked instructions are implemented in such a way that once an interlock is granted, other processors and I/O devices are locked out of performing interlocked operations on the same control variable until the interlock is released. This is termed an interlocked sequence. Only interlocked accesses are locked out by the interlock. An interlocked access must ensure that all previous writes from the issuing processor are visible to all users of the memory system before the interlocked sequence starts, e.g., a write-buffer must be flushed before the read of any interlocked variable).

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. \

9.3 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<I_PROT>, PTE<F̄OE>	Invalidate	Invalidate	
Change PTE<D_PROT>, PTE<F̄OR>, PTE<FOW>	Invalidate		Invalidate
Change I-Stream (e.g., processor writes)	-	Invalidate	-
I/O writes new I-Stream	-	Invalidate	-

Assumptions on the above table:

- o The D Cache watches I/O and processor writes.
- o The I Cache does not watch I/O or processor writes.

Note the Translation Buffer Invalidate instructions (TBFLUSH, MTPR TBIASN, 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).

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. This is relaxed only in that the granularity of sharing is a quadword and by allowing buffering of writes between interlocked operations or writes to the I/O space.

1. Caches/write-buffers that buffer write data must be able to detect a later write from an I/O device and invalidate their write.

2. A processor must guarantee that all of its previous writes are visible to all other processors and/or I/O devices before the write of an interlocked read-modify-write becomes visible to other processors or I/O devices.

3. A processor must guarantee that all of its previous writes are visible to all other processors and I/O devices before a read or write to I/O space.

4. 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.

5. 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.

6. 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 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.

   o 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.

   o 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.

   o 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 access to I/O space and all interlocked sequences.

2. Completing a peripheral input transfer followed by the program reading of the memory must read the input value.

   o 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.

   o 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.

   o Write-buffer - In a system with a write-buffer the write-buffer must either watch the memory bus and have a mechanism invalidating pending writes when an I/O device writes a location that it has buffered or it must purge its contents on all accesses to I/O space and 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.

   o 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.

   o 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.

   o 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.

   o 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.

   o 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.

   o 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.

   o 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.

   o 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.

   o 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 variables shared between processors must be interlocked by software executing one of the interlocked instructions. 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.

   o Write-through - In a system with a write-through cache the memory is written as soon as any write is done so there are no additional requirements.

   o 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.

   o 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. \

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)..-2047(SP). Programs should not store data in this area.

Revision History:

Revision 1.0, 22 December 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 are implemented 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 several 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 leads to a microcoded implementation.

In PRISM, a goal is that microcode will not be necessary for practical implementation. But 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. The disabling of interrupts allows the system to provide multi-instruction sequences as atomic operations (i.e., RMAQI/RMAQIP).

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:

- o Instructions that require complex sequencing as an atomic operation (i.e., REI)
- o Instructions that require interlocked memory access (i.e., RMAQI)
- o Privileged instructions (i.e., MxPR, RMAQIP)
- o Memory management control functions (i.e., TB miss routines, ACV/TNV dispatch routines)
- o Interrupt and exception dispatch routines
- o Power up initialization and booting
- o Console functions
- o Emulation of instructions with no hardware support (i.e., an implementation may chose to do MULL via a multiply step function in the integer ALU)
- o 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 major 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.

- o 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.
- o Epicode needs a set of instructions to access the hardware control registers (i.e., a hardware MxPR).
- o 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:

- o 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.
- o 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 1.0, 22 December 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./

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

In this section a cold start in a uniprocessor environment is discussed. Powerfail recovery and multiprocessor bootstrapping are described in Sections 11.1.3 and 11.1.4.

The following steps occur in the bootstrap sequence. Each is discussed in more detail in subsequent sections:

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 different orders 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:

- o Physical address of RPB
- o Version number
- o Number of processor slots
- o Physical address of per-processor area
- o Physical address of checksum area
- o Checksum
- o Page size
- o ASN size
- o 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.

```
 3
 1                                                              0
+---------------------------------------------------------------+
|                                                               | :RPB
+                   Physical Address of RPB                     +
|                                                               | +4
+---------------------------------------------------------------+
|                      RPB Version Number                       | +8
+---------------------------------------------------------------+
|                  Number of Processor Slots                    | +12
+---------------------------------------------------------------+
|                                                               | +16
+       Physical Address of Per-Processor Area of RPB           +
|                                                               | +20
+---------------------------------------------------------------+
|                                                               | +24
+             Physical Address of Checksum Area                 +
|                                                               | +28
+---------------------------------------------------------------+
|                           Checksum                            | +32
+---------------------------------------------------------------+
|                           Page Size                           | +36
+---------------------------------------------------------------+
|                           ASN Size                            | +40
+---------------------------------------------------------------+
|               Number of Physical Address Bits                 | +44
+---------------------------------------------------------------+
|                     Bootstrap Master ID                       | +48
+---------------------------------------------------------------+
|              Length of Available Epicode Memory               | +52
+---------------------------------------------------------------+
|                                                               | +56
+       Physical Address of Available Epicode Memory            +
|                                                               | +60
+---------------------------------------------------------------+
|                      Bootstrap Options                        |
+---------------------------------------------------------------+
|                     LBN Bootstrap Data                        |
+---------------------------------------------------------------+
|                                                               |
+                        System Device                          +
|                                                               |
+---------------------------------------------------------------+
|                                                               |
+                  System Software Filename                     +
|                                                               |
+---------------------------------------------------------------+
|                      Network Bootstrap                        |
+---------------------------------------------------------------+
```

Figure 11-1: Restart Parameter Block

```
 3
 1                                                                0
+-----------------------------------------------------------------+
|                        State Longword                           | :SLOT
+-----------------------------------------------------------------+
|                        Epicode Length                           | +4
+-----------------------------------------------------------------+
|                                                                 | +8
+                   Epicode Physical Address                      +
|                                                                 | +12
+-----------------------------------------------------------------+
|                                                                 | +16
+                         Restart SCBB                            +
|                                                                 | +20
+-----------------------------------------------------------------+
|                                                                 | +24
+                         Restart PCBB                            +
|                                                                 | +28
+-----------------------------------------------------------------+
|                         Restart IPIE                            | +32
+-----------------------------------------------------------------+
|                         Restart SISR                            | +36
+-----------------------------------------------------------------+
|                         Restart ICIE                            | +40
+-----------------------------------------------------------------+
|                         Restart PRBR                            | +44
+-----------------------------------------------------------------+
|                          Restart R2                             | +48
|                              :                                  |
|                          Restart R63                            | +292
+-----------------------------------------------------------------+
|                          Restart PC                             | +296
+-----------------------------------------------------------------+
|                          Restart PS                             | +300
+-----------------------------------------------------------------+
|                          Restart VC                             | +304
+-----------------------------------------------------------------+
|                          Restart VL                             | +308
+-----------------------------------------------------------------+
|                          Restart VML                            | +312
+-----------------------------------------------------------------+
|                          Restart VMH                            | +316
+-----------------------------------------------------------------+
|                                                                 | +320
+        Physical Address of Vector Register Save Area            +
|                                                                 | +324
+-----------------------------------------------------------------+
|         HWPCB For Use During Bootstrap and Powerfail            | +328
|                                                                 |
+-----------------------------------------------------------------+
```

Figure 11-2: Per-Processor Portion of RPB

```
 3
 1                                         9 8 7 6 5 4 3 2 1 0
+-----------------------------------------+-+-+-+-+-+-+-+-+-+-+
|                                         |C|S|P|E|S|P|P|R|B|
|                  Zero                   |T|R|E|L|T|S|S|I|I|
|                                         |S| | | |C|C|S|P|P|
+-----------------------------------------+-+-+-+-+-+-+-+-+-+-+
```

Figure 11-3: State Longword

Fields in the state longword are interpreted as shown below:

Bits	Description
0	Bootstrap in Progress (BIP) - The system is currently bootstrapping. This bit is set by Epicode and cleared by system software.
1	Restart in Progress (RIP) - The system 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	Processor Enabled (PE) - A processor in a multiprocessor system is enabled. This bit is set and cleared by system software.
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.

11.1.1.3 Epicode Loading

If Epicode does not reside in a ROM, it is 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. Possible loading mechanisms include a diagnostic processor or ROM. The physical address and length of the Epicode is recorded in the RPB.

If control must be transferred to Epicode in memory or ROM 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 processors and controllers 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 port controller

4. 256 Kbytes of good memory for use by system software

The virtual memory is at the high end of the 32-bit virtual address space and is laid out as shown below:

```
              +----------------------------------+
              |                                  |  FFF80000
     256 KB   |      256 KB of good memory       |  FFFBFFFF
              +----------------------------------+
              |                                  |  FFFC0000
      64 KB   |  I/O port controller registers   |  FFFCFFFF
              +----------------------------------+
              |                                  |  FFFD0000
      64 KB   |               RPB                |  FFFDFFFF
              +----------------------------------+
              |                                  |  FFFE0000
      64 KB   |       level 2 page table         |  FFFEFFFF
              +----------------------------------+
              |                                  |  FFFF0000
      64 KB   |       level 1 page table         |  FFFFFFFF
              +----------------------------------+
```

Figure 11-4: Initial Virtual Memory Layout

All pages have Kernel read/write/execute protection.

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
CRSR	Console Receive Status	disabled
CTSR	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 locations have Kernel read/write/execute access.

All scalar and vector register contents, including the stack pointer, are undefined.

All bootstrap information is passed from the console to system software in the RPB. This includes:

- o System device name
- o System software file name
- o Bootstrap options
- o Logical Block Number (LBN) bootstrap data if appropriate

o 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 only if system software has allocated a save area and recorded its address in the RPB and if the Vector Enable bit is 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 is restart. After determining that memory was backed up and the halt action 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, return to Step 1.

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 Processor Enabled (PE) 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.

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. 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.

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) 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 windows 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, warm restart, or cold boot.

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.

- o 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.
- o RUBOUT - When the operator types RUBOUT, the console ignores the entire line and prompts for another command.
- o 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.
- o 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.
- o CTRL/Q - Resumes output stopped by CTRL/S. Additional CTRL/Qs are ignored. CTRL/S and CTRL/Q are not echoed.
- o 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:

- o INITIALIZE
- o START
- o CONTINUE
- o BOOT

Data transfer commands:

- o EXAMINE
- o DEPOSIT

Console control commands:

- o TEST

BOOT

Format:

```
        B [<qualifier list>] [<device:>][<filename>]
```

Qualifiers:

- o /<data> - This allows a console user to specify the bootstrap options parameter to be stored in the RPB.
- o /S - The console loads the bootstrap program and prompts for further console commands.
- o /L - The console loads the bootstrap program from the logical block number 0.

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 system bootstrap program boots the operating system from the specified device. The default device and filename are implementation-dependent. The console searches through an implementation-dependent default search list.

CONTINUE

Format:

```
        C
```

Qualifiers:

None

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.

Deposits to IPRs execute the equivalent MTPR instructions using the contents of scalar registers R4 and R5 (when needed) for their data. See Chapter 8 for register usage.

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/R ESP	Deposits the contents of R4 in the Executive Stack Pointer.

EXAMINE

Format:

```
        E [<qualifier list>] <address>
```

Qualifiers:

See Table 11-2

Response:

```
        <tab><address space identifier> <address> <data>
```

The address space identifier can be:

- 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 executes the equivalent MFPR instruction and writes the appropriate scalar registers called for in the MFPR description. See Chapter 8 for register usage. The response displays the registers that are written and the data.

Examples:

The response to E/R WHAMI on processor 3 is:

```
        R R4 00000003
```

The response to E/V 1234564 is:

```
        P 0000FE3C 01739102
```

Where the virtual address 1234564 maps to the physical address FE3C.

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-existant memory location, The console issues a "?NXM" error message.
/R	The address space is registers. These are the scalar registers, vector 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. 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 'n'. The register number is decimal and in the range 0-15; the element number is decimal and in the range 0-63. VC - Vector Count. VL - Vector Length. VM - Vector Mask. CRCS - Console Receive Control Status. CTCS - Console Transmit Control Status. ESP - Executive Stack Pointer. ICIE - Interval Clock Interrupt Enable. IPIE - Interprocessor Interrupt Enable. KSP - Kernel Stack Pointer. PRBR - Processor Base Register. SCBB - System Control Block Base. SSP - Supervisor Stack Pointer. TOY - Time Of Year. USP - User Stack Pointer.

Table 11-2: Qualifiers for Examine and Deposit (Continued)

Qualifier	Meaning
	The following symbolic addresses can be used for the Examine command only: ASN - Address Space Number. ASTEN - AST Enable. ASTSR - AST Summary Register. CRDB - Console Receive Data Buffer. PCBB - Privileged Context Block Base. PRSN - Processor 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. TBIASN - Translation Buffer Invalidate by ASN 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.

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

Table 11-2: Qualifiers for Examine and Deposit (Continued)

Qualifier	Meaning
	for quadword). For other address spaces, the address is the last addressed referenced, plus 1.
	'-' - 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.
	'*' - The location last referenced in an examine or deposit.
	'@' - The location addressed by the last location referenced in an examine or deposit.

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:

- o BEL - Illegal characters are ignored and are echoed as BEL.
- o ?NXM - Non-existent memory.
- o ?TNV - Translation Not Valid.

Revision History:

Revision 1.0, 22 December 1985

1. Initial review version.

CHAPTER 12
I/O ARCHITECTURE

12.1 TO BE SUPPLIED

APPENDIX A

INSTRUCTION SET SUMMARY

This appendix summarizes the instruction mnemonics and their opcode and function code fields in hex. There are three listings:

- o Functional group listing - Groups related instructions together.
- o Mnemonic listing - Lists the instructions sorted by mnemonic.
- o Opcode listing - Lists the instructions sorted by opcode and function code.

A.1 ENCODING HINTS

The instruction encoding was worked out 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	02
VLDQ	ra,rb,vc	30	03
VSTL	ra,rb,vc	30	06
VSTQ	ra,rb,vc	30	07
VGATHL	ra,vb,vc	31	02
VGATHQ	ra,vb,vc	31	03
VSCATL	ra,vb,vc	31	06
VSCATQ	ra,vb,vc	31	07
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
COPRD	ra	34	-
COPWR	ra	35	-
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
FOB	ra	2A	-

EPIRES0		2D	-
EPIRES1		2E	-
EPIRES2		2F	-
ADD	ra,rb,rc	01	00
ADDV	ra,rb,rc	01	01
SUB	ra,rb,rc	01	08
SUBV	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
DIV	ra,rb,rc	04	00
DIVV	ra,rb,rc	04	01
REM	ra,rb,rc	04	04
MULL	ra,rb,rc	04	02
MULV	ra,rb,rc	04	03
MULH	ra,rb,rc	04	06
UMULH	ra,rb,rc	04	0A
CVTFL	ra,rc	05	04
CVTFLZ	ra,rc	05	00
CVTLF	ra,rc	05	05
CVTLFZ	ra,rc	05	01
CVTFG	ra,rc	06	00
CVTLG	ra,rc	06	01
CVTGL	ra,rc	07	04
CVTGLZ	ra,rc	07	00
CVTGF	ra,rc	07	05
CVTGFZ	ra,rc	07	01
CVTGFU	ra,rc	07	0D
CVTGFUZ	ra,rc	07	09

ADDG	ra,rb,rc	08	04
ADDGZ	ra,rb,rc	08	00
ADDGU	ra,rb,rc	08	0C
ADDGUZ	ra,rb,rc	08	08
SUBG	ra,rb,rc	08	05
SUBGZ	ra,rb,rc	08	01
SUBGU	ra,rb,rc	08	0D
SUBGUZ	ra,rb,rc	08	09
ADDF	ra,rb,rc	09	04
ADDFZ	ra,rb,rc	09	00
ADDFU	ra,rb,rc	09	0C
ADDFUZ	ra,rb,rc	09	08
SUBF	ra,rb,rc	09	05
SUBFZ	ra,rb,rc	09	01
SUBFU	ra,rb,rc	09	0D
SUBFUZ	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
DIVGZ	ra,rb,rc	0C	00
DIVGU	ra,rb,rc	0C	0C
DIVGUZ	ra,rb,rc	0C	08
MULG	ra,rb,rc	0C	05
MULGZ	ra,rb,rc	0C	01
MULGU	ra,rb,rc	0C	0D
MULGUZ	ra,rb,rc	0C	09
DIVF	ra,rb,rc	0D	04
DIVFZ	ra,rb,rc	0D	00
DIVFU	ra,rb,rc	0D	0C
DIVFUZ	ra,rb,rc	0D	08
MULF	ra,rb,rc	0D	05
MULFZ	ra,rb,rc	0D	01
MULFU	ra,rb,rc	0D	0D
MULFUZ	ra,rb,rc	0D	09
LDA	d(rb),ra	0E	-
VMERGE	ra,vb,vc	10	00

VMERGE	va,vb,vc	10	10
IOTA	ra,vc	10	01
VADD	ra,vb,vc	11	00
VADDV	ra,vb,vc	11	01
VSUB	ra,vb,vc	11	02
VSUBV	ra,vb,vc	11	03
VADD	va,vb,vc	11	10
VADDV	va,vb,vc	11	11
VSUB	va,vb,vc	11	12
VSUBV	va,vb,vc	11	13
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
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
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
VDIV	ra,vb,vc	14	00
VDIVV	ra,vb,vc	14	01
VREM	ra,vb,vc	14	04
VMULL	ra,vb,vc	14	02
VMULV	ra,vb,vc	14	03
VDIV	va,vb,vc	14	10
VDIVV	va,vb,vc	14	11
VREM	va,vb,vc	14	14
VMULL	va,vb,vc	14	12
VMULV	va,vb,vc	14	13
VCVTFL	va,vc	15	14

VCVTFLZ	va,vc	15	10
VCVTLF	va,vc	15	15
VCVTLFZ	va,vc	15	11
VCVTFG	va,vc	16	10
VCVTLG	va,vc	16	11
VCVTGL	va,vc	17	14
VCVTGLZ	va,vc	17	10
VCVTGF	va,vc	17	15
VCVTGFZ	va,vc	17	11
VCVTGFU	va,vc	17	1D
VCVTGFUZ	va,vc	17	19
VADDG	ra,vb,vc	18	04
VADDGZ	ra,vb,vc	18	00
VADDGU	ra,vb,vc	18	0C
VADDGUZ	ra,vb,vc	18	08
VSUBG	ra,vb,vc	18	05
VSUBGZ	ra,vb,vc	18	01
VSUBGU	ra,vb,vc	18	0D
VSUBGUZ	ra,vb,vc	18	09
VADDG	va,vb,vc	18	14
VADDGZ	va,vb,vc	18	10
VADDGU	va,vb,vc	18	1C
VADDGUZ	va,vb,vc	18	18
VSUBG	va,vb,vc	18	15
VSUBGZ	va,vb,vc	18	11
VSUBGU	va,vb,vc	18	1D
VSUBGUZ	va,vb,vc	18	19
VADDF	ra,vb,vc	19	04
VADDFZ	ra,vb,vc	19	00
VADDFU	ra,vb,vc	19	0C
VADDFUZ	ra,vb,vc	19	08
VSUBF	ra,vb,vc	19	05
VSUBFZ	ra,vb,vc	19	01
VSUBFU	ra,vb,vc	19	0D
VSUBFUZ	ra,vb,vc	19	09
VADDF	va,vb,vc	19	14
VADDFZ	va,vb,vc	19	10
VADDFU	va,vb,vc	19	1C
VADDFUZ	va,vb,vc	19	18
VSUBF	va,vb,vc	19	15
VSUBFZ	va,vb,vc	19	11
VSUBFU	va,vb,vc	19	1D
VSUBFUZ	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
VDIVGZ	ra,vb,vc	1C	00
VDIVGU	ra,vb,vc	1C	0C
VDIVGUZ	ra,vb,vc	1C	08
VMULG	ra,vb,vc	1C	05
VMULGZ	ra,vb,vc	1C	01
VMULGU	ra,vb,vc	1C	0D
VMULGUZ	ra,vb,vc	1C	09
VDIVG	va,vb,vc	1C	14
VDIVGZ	va,vb,vc	1C	10
VDIVGU	va,vb,vc	1C	1C
VDIVGUZ	va,vb,vc	1C	18
VMULG	va,vb,vc	1C	15
VMULGZ	va,vb,vc	1C	11
VMULGU	va,vb,vc	1C	1D
VMULGUZ	va,vb,vc	1C	19
VDIVF	ra,vb,vc	1D	04
VDIVFZ	ra,vb,vc	1D	00
VDIVFU	ra,vb,vc	1D	0C
VDIVFUZ	ra,vb,vc	1D	08
VMULF	ra,vb,vc	1D	05
VMULFZ	ra,vb,vc	1D	01
VMULFU	ra,vb,vc	1D	0D
VMULFUZ	ra,vb,vc	1D	09
VDIVF	va,vb,vc	1D	14
VDIVFZ	va,vb,vc	1D	10
VDIVFU	va,vb,vc	1D	1C
VDIVFUZ	va,vb,vc	1D	18
VMULF	va,vb,vc	1D	15
VMULFZ	va,vb,vc	1D	11
VMULFU	va,vb,vc	1D	1D
VMULFUZ	va,vb,vc	1D	19

HALT		00	00
DRAIN		00	30
REI		00	02
BPT		00	03
BUGCHK		00	04
IFLUSH		00	31
MOVPS		00	32
PROBER		00	0A
PROBEW		00	0B
SWASTEN		00	05
SWIPL		00	06
SWPCTX		00	07
RMAQI		00	38
RMAQIP		00	39
TBFLUSH		00	08
MFPR	ESP	00	C1
MTPR	ESP	00	81
MFPR	SSP	00	C2
MTPR	SSP	00	82
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
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	TBIASN	00	90
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	PRSN	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
reserved		0F	00
reserved		1E	00
reserved		1F	00
reserved		2B	00
reserved		2C	00

A.3 MNEMONIC LISTING

Mnemonic		Opcode (hex)	Function Code (hex)
ADD	ra,rb,rc	01	00
ADDF	ra,rb,rc	09	04
ADDFU	ra,rb,rc	09	0C
ADDFUZ	ra,rb,rc	09	08
ADDFZ	ra,rb,rc	09	00
ADDG	ra,rb,rc	08	04
ADDGU	ra,rb,rc	08	0C
ADDGUZ	ra,rb,rc	08	08
ADDGZ	ra,rb,rc	08	00
ADDV	ra,rb,rc	01	01
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	-
BPT		00	03
BUGCHK		00	04
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
CMPUGE	ra,rb,rc	02	1C
CMPUGT	ra,rb,rc	02	1A
CMPULE	ra,rb,rc	02	1B
CMPULT	ra,rb,rc	02	1D
COPRD	ra	34	-
COPWR	ra	35	-
CVTFG	ra,rc	06	00

CVTFL	ra,rc	05	04
CVTFLZ	ra,rc	05	00
CVTGF	ra,rc	07	05
CVTGFU	ra,rc	07	0D
CVTGFUZ	ra,rc	07	09
CVTGFZ	ra,rc	07	01
CVTGL	ra,rc	07	04
CVTGLZ	ra,rc	07	00
CVTLF	ra,rc	05	05
CVTLFZ	ra,rc	05	01
CVTLG	ra,rc	06	01
DIV	ra,rb,rc	04	00
DIVF	ra,rb,rc	0D	04
DIVFU	ra,rb,rc	0D	0C
DIVFUZ	ra,rb,rc	0D	08
DIVFZ	ra,rb,rc	0D	00
DIVG	ra,rb,rc	0C	04
DIVGU	ra,rb,rc	0C	0C
DIVGUZ	ra,rb,rc	0C	08
DIVGZ	ra,rb,rc	0C	00
DIVV	ra,rb,rc	04	01
DRAIN		00	30
EPIRES0		2D	-
EPIRES1		2E	-
EPIRES2		2F	-
EPIRES3		36	-
EPIRES4		37	-
EQV	ra,rb,rc	03	0A
FOB	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	0E	-
LDB	d(rb),ra	38	-
LDL	d(rb),ra	3A	-
LDQ	d(rb),ra	3B	-
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	ESP	00	C1
MFPR	ICIE	00	CB
MFPR	IPIE	00	D2
MFPR	PCBB	00	C5
MFPR	PRBR	00	D3
MFPR	PRSN	00	D6
MFPR	PTBR	00	C4
MFPR	SCBB	00	C6
MFPR	SID	00	D5

MFPR	SISR	00	CA
MFPR	SSP	00	C2
MFPR	TBCHK	00	CE
MFPR	TOY	00	CC
MFPR	USP	00	C3
MFPR	WHAMI	00	D4
MOVPS		00	32
MTPR	ASTRR	00	87
MTPR	CRCS	00	97
MTPR	CTCS	00	99
MTPR	CTDB	00	9A
MTPR	ESP	00	81
MTPR	ICIE	00	8B
MTPR	IPIE	00	92
MTPR	IPIR	00	91
MTPR	PRBR	00	93
MTPR	SCBB	00	86
MTPR	SIRR	00	8A
MTPR	SSP	00	82
MTPR	TBIASN	00	90
MTPR	TBIS	00	8F
MTPR	TOY	00	8C
MTPR	USP	00	83
MULF	ra,rb,rc	0D	05
MULFU	ra,rb,rc	0D	0D
MULFUZ	ra,rb,rc	0D	09
MULFZ	ra,rb,rc	0D	01
MULG	ra,rb,rc	0C	05
MULGU	ra,rb,rc	0C	0D
MULGUZ	ra,rb,rc	0C	09
MULGZ	ra,rb,rc	0C	01
MULH	ra,rb,rc	04	06
MULL	ra,rb,rc	04	02
MULV	ra,rb,rc	04	03
OR	ra,rb,rc	03	01
ORNOT	ra,rb,rc	03	09
PROBER		00	0A
PROBEW		00	0B
RDVC	rc	32	01
RDVL	rc	32	00
RDVMH	rc	32	03
RDVML	rc	32	02
REI		00	02
REM	ra,rb,rc	04	04
RMAQI		00	38
RMAQIP		00	39
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	-
STW	ra,d(rb)	3D	-

SUB	ra,rb,rc	01	08
SUBF	ra,rb,rc	09	05
SUBFU	ra,rb,rc	09	0D
SUBFUZ	ra,rb,rc	09	09
SUBFZ	ra,rb,rc	09	01
SUBG	ra,rb,rc	08	05
SUBGU	ra,rb,rc	08	0D
SUBGUZ	ra,rb,rc	08	09
SUBGZ	ra,rb,rc	08	01
SUBV	ra,rb,rc	01	09
SWASTEN		00	05
SWIPL		00	06
SWPCTX		00	07
TBFLUSH		00	08
UMULH	ra,rb,rc	04	0A
VADD	ra,vb,vc	11	00
VADD	va,vb,vc	11	10
VADDF	ra,vb,vc	19	04
VADDF	va,vb,vc	19	14
VADDFU	ra,vb,vc	19	0C
VADDFU	va,vb,vc	19	1C
VADDFUZ	ra,vb,vc	19	08
VADDFUZ	va,vb,vc	19	18
VADDFZ	ra,vb,vc	19	00
VADDFZ	va,vb,vc	19	10
VADDG	ra,vb,vc	18	04
VADDG	va,vb,vc	18	14
VADDGU	ra,vb,vc	18	0C
VADDGU	va,vb,vc	18	1C
VADDGUZ	ra,vb,vc	18	08
VADDGUZ	va,vb,vc	18	18
VADDGZ	ra,vb,vc	18	00
VADDGZ	va,vb,vc	18	10
VADDV	ra,vb,vc	11	01
VADDV	va,vb,vc	11	11
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
VCVTFG	va,vc	16	10
VCVTFL	va,vc	15	14
VCVTFLZ	va,vc	15	10
VCVTGF	va,vc	17	15
VCVTGFU	va,vc	17	1D
VCVTGFUZ	va,vc	17	19
VCVTGFZ	va,vc	17	11
VCVTGL	va,vc	17	14
VCVTGLZ	va,vc	17	10
VCVTLF	va,vc	15	15
VCVTLFZ	va,vc	15	11
VCVTLG	va,vc	16	11
VDIV	ra,vb,vc	14	00
VDIV	va,vb,vc	14	10
VDIVF	ra,vb,vc	1D	04
VDIVF	va,vb,vc	1D	14
VDIVFU	ra,vb,vc	1D	0C
VDIVFU	va,vb,vc	1D	1C
VDIVFUZ	ra,vb,vc	1D	08
VDIVFUZ	va,vb,vc	1D	18
VDIVFZ	ra,vb,vc	1D	00
VDIVFZ	va,vb,vc	1D	10
VDIVG	ra,vb,vc	1C	04
VDIVG	va,vb,vc	1C	14
VDIVGU	ra,vb,vc	1C	0C
VDIVGU	va,vb,vc	1C	1C
VDIVGUZ	ra,vb,vc	1C	08
VDIVGUZ	va,vb,vc	1C	18
VDIVGZ	ra,vb,vc	1C	00
VDIVGZ	va,vb,vc	1C	10
VDIVV	ra,vb,vc	14	01
VDIVV	va,vb,vc	14	11
VEQV	ra,vb,vc	13	0A

VEQV	va,vb,vc	13	1A
VGATHL	ra,vb,vc	31	02
VGATHQ	ra,vb,vc	31	03
VLDL	ra,rb,vc	30	02
VLDQ	ra,rb,vc	30	03
VMERGE	ra,vb,vc	10	00
VMERGE	va,vb,vc	10	10
VMULF	ra,vb,vc	1D	05
VMULF	va,vb,vc	1D	15
VMULFU	ra,vb,vc	1D	0D
VMULFU	va,vb,vc	1D	1D
VMULFUZ	ra,vb,vc	1D	09
VMULFUZ	va,vb,vc	1D	19
VMULFZ	ra,vb,vc	1D	01
VMULFZ	va,vb,vc	1D	11
VMULG	ra,vb,vc	1C	05
VMULG	va,vb,vc	1C	15
VMULGU	ra,vb,vc	1C	0D
VMULGU	va,vb,vc	1C	1D
VMULGUZ	ra,vb,vc	1C	09
VMULGUZ	va,vb,vc	1C	19
VMULGZ	ra,vb,vc	1C	01
VMULGZ	va,vb,vc	1C	11
VMULL	ra,vb,vc	14	02
VMULL	va,vb,vc	14	12
VMULV	ra,vb,vc	14	03
VMULV	va,vb,vc	14	13
VOR	ra,vb,vc	13	01
VOR	va,vb,vc	13	11
VORNOT	ra,vb,vc	13	09
VORNOT	va,vb,vc	13	19
VREM	ra,vb,vc	14	04
VREM	va,vb,vc	14	14
VSCATL	ra,vb,vc	31	06
VSCATQ	ra,vb,vc	31	07
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	06
VSTQ	ra,rb,vc	30	07
VSUB	ra,vb,vc	11	02
VSUB	va,vb,vc	11	12
VSUBF	ra,vb,vc	19	05
VSUBF	va,vb,vc	19	15
VSUBFU	ra,vb,vc	19	0D
VSUBFU	va,vb,vc	19	1D
VSUBFUZ	ra,vb,vc	19	09
VSUBFUZ	va,vb,vc	19	19
VSUBFZ	ra,vb,vc	19	01
VSUBFZ	va,vb,vc	19	11
VSUBG	ra,vb,vc	18	05
VSUBG	va,vb,vc	18	15
VSUBGU	ra,vb,vc	18	0D

VSUBGU	va,vb,vc	18	1D
VSUBGUZ	ra,vb,vc	18	09
VSUBGUZ	va,vb,vc	18	19
VSUBGZ	ra,vb,vc	18	01
VSUBGZ	va,vb,vc	18	11
VSUBV	ra,vb,vc	11	03
VSUBV	va,vb,vc	11	13
VXOR	ra,vb,vc	13	02
VXOR	va,vb,vc	13	12
WRVC	ra	33	01
WRVL	ra	33	00
WRVMH	ra	33	03
WRVML	ra	33	02
XOR	ra,rb,rc	03	02
reserved		0F	00
reserved		1E	00
reserved		1F	00
reserved		2B	00
reserved		2C	00

A.4 OPCODE LISTING

Mnemonic		Opcode (hex)	Function Code (hex)
HALT		00	00
REI		00	02
BPT		00	03
BUGCHK		00	04
SWASTEN		00	05
SWIPL		00	06
SWPCTX		00	07
TBFLUSH		00	08
PROBER		00	0A
PROBEW		00	0B
DRAIN		00	30
IFLUSH		00	31
MOVPS		00	32
RMAQI		00	38
RMAQIP		00	39
MTPR	ESP	00	81
MTPR	SSP	00	82
MTPR	USP	00	83
MTPR	SCBB	00	86
MTPR	ASTRR	00	87
MTPR	SIRR	00	8A
MTPR	ICIE	00	8B
MTPR	TOY	00	8C
MTPR	TBIS	00	8F
MTPR	TBIASN	00	90
MTPR	IPIR	00	91
MTPR	IPIE	00	92
MTPR	PRBR	00	93
MTPR	CRCS	00	97
MTPR	CTCS	00	99
MTPR	CTDB	00	9A
MFPR	ESP	00	C1
MFPR	SSP	00	C2
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	PRSN	00	D6
MFPR	CRCS	00	D7
MFPR	CRDB	00	D8
MFPR	CTCS	00	D9
ADD	ra,rb,rc	01	00
ADDV	ra,rb,rc	01	01
SUB	ra,rb,rc	01	08
SUBV	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
DIV	ra,rb,rc	04	00
DIVV	ra,rb,rc	04	01
MULL	ra,rb,rc	04	02
MULV	ra,rb,rc	04	03
REM	ra,rb,rc	04	04
MULH	ra,rb,rc	04	06
UMULH	ra,rb,rc	04	0A
CVTFLZ	ra,rc	05	00
CVTLFZ	ra,rc	05	01
CVTFL	ra,rc	05	04
CVTLF	ra,rc	05	05
CVTFG	ra,rc	06	00
CVTLG	ra,rc	06	01
CVTGLZ	ra,rc	07	00
CVTGFZ	ra,rc	07	01
CVTGL	ra,rc	07	04
CVTGF	ra,rc	07	05
CVTGFUZ	ra,rc	07	09
CVTGFU	ra,rc	07	0D
ADDGZ	ra,rb,rc	08	00
SUBGZ	ra,rb,rc	08	01
ADDG	ra,rb,rc	08	04
SUBG	ra,rb,rc	08	05
ADDGUZ	ra,rb,rc	08	08
SUBGUZ	ra,rb,rc	08	09

ADDGU	ra,rb,rc	08	0C
SUBGU	ra,rb,rc	08	0D
ADDFZ	ra,rb,rc	09	00
SUBFZ	ra,rb,rc	09	01
ADDF	ra,rb,rc	09	04
SUBF	ra,rb,rc	09	05
ADDFUZ	ra,rb,rc	09	08
SUBFUZ	ra,rb,rc	09	09
ADDFU	ra,rb,rc	09	0C
SUBFU	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
DIVGZ	ra,rb,rc	0C	00
MULGZ	ra,rb,rc	0C	01
DIVG	ra,rb,rc	0C	04
MULG	ra,rb,rc	0C	05
DIVGUZ	ra,rb,rc	0C	08
MULGUZ	ra,rb,rc	0C	09
DIVGU	ra,rb,rc	0C	0C
MULGU	ra,rb,rc	0C	0D
DIVFZ	ra,rb,rc	0D	00
MULFZ	ra,rb,rc	0D	01
DIVF	ra,rb,rc	0D	04
MULF	ra,rb,rc	0D	05
DIVFUZ	ra,rb,rc	0D	08
MULFUZ	ra,rb,rc	0D	09
DIVFU	ra,rb,rc	0D	0C
MULFU	ra,rb,rc	0D	0D
LDA	d(rb),ra	0E	-
reserved		0F	00
VMERGE	ra,vb,vc	10	00
IOTA	ra,vc	10	01
VMERGE	va,vb,vc	10	10
VADD	ra,vb,vc	11	00
VADDV	ra,vb,vc	11	01
VSUB	ra,vb,vc	11	02
VSUBV	ra,vb,vc	11	03
VADD	va,vb,vc	11	10
VADDV	va,vb,vc	11	11
VSUB	va,vb,vc	11	12
VSUBV	va,vb,vc	11	13
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
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
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
VDIV	ra,vb,vc	14	00
VDIVV	ra,vb,vc	14	01
VMULL	ra,vb,vc	14	02
VMULV	ra,vb,vc	14	03
VREM	ra,vb,vc	14	04
VDIV	va,vb,vc	14	10
VDIVV	va,vb,vc	14	11
VMULL	va,vb,vc	14	12
VMULV	va,vb,vc	14	13
VREM	va,vb,vc	14	14
VCVTFLZ	va,vc	15	10
VCVTLFZ	va,vc	15	11
VCVTFL	va,vc	15	14
VCVTLF	va,vc	15	15
VCVTFG	va,vc	16	10
VCVTLG	va,vc	16	11
VCVTGLZ	va,vc	17	10
VCVTGFZ	va,vc	17	11
VCVTGL	va,vc	17	14
VCVTGF	va,vc	17	15
VCVTGFUZ	va,vc	17	19
VCVTGFU	va,vc	17	1D
VADDGZ	ra,vb,vc	18	00
VSUBGZ	ra,vb,vc	18	01
VADDG	ra,vb,vc	18	04
VSUBG	ra,vb,vc	18	05
VADDGUZ	ra,vb,vc	18	08
VSUBGUZ	ra,vb,vc	18	09
VADDGU	ra,vb,vc	18	0C

VSUBGU	ra,vb,vc	18	0D
VADDGZ	va,vb,vc	18	10
VSUBGZ	va,vb,vc	18	11
VADDG	va,vb,vc	18	14
VSUBG	va,vb,vc	18	15
VADDGUZ	va,vb,vc	18	18
VSUBGUZ	va,vb,vc	18	19
VADDGU	va,vb,vc	18	1C
VSUBGU	va,vb,vc	18	1D
VADDFZ	ra,vb,vc	19	00
VSUBFZ	ra,vb,vc	19	01
VADDF	ra,vb,vc	19	04
VSUBF	ra,vb,vc	19	05
VADDFUZ	ra,vb,vc	19	08
VSUBFUZ	ra,vb,vc	19	09
VADDFU	ra,vb,vc	19	0C
VSUBFU	ra,vb,vc	19	0D
VADDFZ	va,vb,vc	19	10
VSUBFZ	va,vb,vc	19	11
VADDF	va,vb,vc	19	14
VSUBF	va,vb,vc	19	15
VADDFUZ	va,vb,vc	19	18
VSUBFUZ	va,vb,vc	19	19
VADDFU	va,vb,vc	19	1C
VSUBFU	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
VDIVGZ	ra,vb,vc	1C	00
VMULGZ	ra,vb,vc	1C	01
VDIVG	ra,vb,vc	1C	04
VMULG	ra,vb,vc	1C	05
VDIVGUZ	ra,vb,vc	1C	08

VMULGUZ	ra,vb,vc	1C	09
VDIVGU	ra,vb,vc	1C	0C
VMULGU	ra,vb,vc	1C	0D
VDIVGZ	va,vb,vc	1C	10
VMULGZ	va,vb,vc	1C	11
VDIVG	va,vb,vc	1C	14
VMULG	va,vb,vc	1C	15
VDIVGUZ	va,vb,vc	1C	18
VMULGUZ	va,vb,vc	1C	19
VDIVGU	va,vb,vc	1C	1C
VMULGU	va,vb,vc	1C	1D
VDIVFZ	ra,vb,vc	1D	00
VMULFZ	ra,vb,vc	1D	01
VDIVF	ra,vb,vc	1D	04
VMULF	ra,vb,vc	1D	05
VDIVFUZ	ra,vb,vc	1D	08
VMULFUZ	ra,vb,vc	1D	09
VDIVFU	ra,vb,vc	1D	0C
VMULFU	ra,vb,vc	1D	0D
VDIVFZ	va,vb,vc	1D	10
VMULFZ	va,vb,vc	1D	11
VDIVF	va,vb,vc	1D	14
VMULF	va,vb,vc	1D	15
VDIVFUZ	va,vb,vc	1D	18
VMULFUZ	va,vb,vc	1D	19
VDIVFU	va,vb,vc	1D	1C
VMULFU	va,vb,vc	1D	1D
reserved		1E	00
reserved		1F	00
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
FOB	ra	2A	-
reserved		2B	00
reserved		2C	00
EPIRES0		2D	-
EPIRES1		2E	-
EPIRES2		2F	-
VLDL	ra,rb,vc	30	02
VLDQ	ra,rb,vc	30	03
VSTL	ra,rb,vc	30	06
VSTQ	ra,rb,vc	30	07
VGATHL	ra,vb,vc	31	02
VGATHQ	ra,vb,vc	31	03
VSCATL	ra,vb,vc	31	06
VSCATQ	ra,vb,vc	31	07
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
COPRD	ra	34	-
COPWR	ra	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	-

APPENDIX B

64-BIT ARCHITECTURE

B.1 GOALS AND NON-GOALS

At some point in the future the proposed 32-bit PRISM architecture will run out of virtual address bits. When this event occurs it is highly desirable to upgrade the PRISM architecture to a larger virtual address and migrate software with as little effort as possible.

If all software were written correctly and in a higher level language, then the source programs could simply be recompiled to take advantage of the larger virtual address space. It is doubtful, however, that this level of transportability will be achieved since a large amount of VAX software which is written in BLISS will be transported to PRISM architecture machines with little or no change (i.e., most BLISS software will not be rewritten to alleviate address size dependencies).

This appendix describes a possible 64-bit extension of the PRISM architecture. It does not claim or imply that this is an optimal solution, or for that matter, the one that will actually be implemented. It assumes that the 32-bit architecture specified in this document will be implemented first, and later, a 64-bit architecture with a compatible 32-bit mode will be implemented. This would allow software to be migrated to the extended architecture without extensive rewrite.

The 32-bit PRISM architecture has 32-bit registers. There is a defined set of 32-bit integer operations, 32-bit single precision floating operations and 64-bit double precision floating operations on even/odd register pairs. Virtual addresses are 32-bits long.

The proposed 64-bit architecture has 64-bit registers. There is a defined set of 64-bit integer operations, 32-bit single precision floating operations and 64-bit double precision floating operations. Virtual addresses are 64-bits long.

In addition, the 64-bit architecture has a 32-bit mode which is enabled by a bit in the PS. When running with 32-bit mode enabled, integer operations are executed compatibly with the 32-bit architecture and virtual addresses are constrained to 32-bits. Double precision floating operations are executed using even/odd register pairs.

Goals of this proposal are:

1. To design an architectural solution to the quandry surrounding the cost effectiveness of a 32-bit architecture versus the long-term desirability of a 64-bit architecture.

2. To be able to run 32-bit software on a 64-bit architecture WITHOUT recompilation or relinking.

Architectural constraints are:

1. A 32-bit program when run on a 64-bit machine must get identical answers. This means that if a computation overflows on the 32-bit machine it must also overflow on the 64-bit machine.

2. It must be possible to write a program that may be compiled and run on either the 32- or 64-bit environment without any source changes.

Non-goals are:

1. For a program compiled for a 64-bit architecture to be able to run on a 32-bit machine without recompilation.

The architectural modifications are such that new instructions are not required. The definition of an operation depends on whether the program is running in 32-bit or 64-bit mode.

B.2 DATA TYPES

The 64-bit architecture supports the following data types:

1. Byte - zero extended loads and stores only.

2. Word - zero extended loads and stores only.

3. Longword - zero extended loads and stores only.

4. Quadword - complete set of arithmetic, logical, and compare operations. This is the primary integer data type. All operations provided for longwords in the 32-bit architecture are provided on quadwords in the 64-bit architecture.

5. F_floating - same operations as the 32-bit architecture except that converts to and from quadword are provided instead of longword.

6. G_floating - same operations as the 32-bit architecture except that converts to and from quadword are provided instead of longword.

B.3 REGISTERS

B.3.1 Scalar Registers

There are 64 scalar registers, each 64-bits wide. R1 is the stack pointer. R0 always reads as zero and writes are ignored.

B.3.2 Vector Registers

The vector registers are identical to those in the 32-bit architecture. There are 16 vector registers, each containing 64 elements. Each element is 64-bits wide. The Vector Length register is 6-bits wide. The Vector Mask register is 64-bits wide. The Vector Count register is 7-bits wide.

B.3.3 Program Counter

The PC is 64-bits wide. Bits <1:0> and high order bits corresponding to reserved virtual address bits are RAZ/IGN (see Section B.6.2).

B.4 INSTRUCTION FORMATS

All instructions are 32-bits long. The instruction formats and encodings are identical to those used in the 32-bit architecture.

B.5 INSTRUCTION SET

The definition of an operation depends on whether the program is running in 32-bit or 64-bit mode. In 32-bit mode all integer operations are zero extended from bit 32 through 63. This is required so that addresses are the same in 32-bit mode as they are in 64-bit mode. The operating system must allocate space for 32-bit mode programs from the first 4 Gbytes of the virtual address space. Effective address calculations for loads and stores are zero extended from bit 32 through 63 also. And branches are constrained to not go outside the 32-bit range.

The following sections describe instruction operation in 32- and 64-bit modes. Table B-1 describes the instruction notation.

Table B-1: Instruction Notation

Notation	Meaning
L_x	When used on the left hand side of an assignment statement, bits x<31:0> receive the result and bits x<63:32> are cleared. When used as a source operand, only bits x<31:0> participate in the operation.
L_QRn	When used on the left hand side of an assignment statement, bits <31:0> of each of the even-odd register pair QRn receive the low and high parts of the result and bits <63:32> of each of the register pair are cleared. When used as a source operand, only bits <31:0> of each of the even-odd register pair QRn participate in the operation.
I	This designator is used to denote integer data type in convert instruction mnemonics. In 32-bit mode, I denotes longword, and in 64-bit mode I denotes quadword.

B.5.1 MEMORY LOAD/STORE INSTRUCTIONS

```
Instr          32-bit Mode                       64-bit Mode
----           -----------                       -----------

LDA     L_Ra <- Rbv + SEXT(disp)          Ra <- Rbv + SEXT(disp)

LDB     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        L_Ra <- ZEXT((va)<7:0>)           Ra  <- ZEXT((va)<7:0>)

LDW     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        L_Ra <- ZEXT((va)<15:0>)          Ra  <- ZEXT((va)<15:0>)

LDL     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        L_Ra <- (va)<31:0>                Ra <- ZEXT((va)<31:0>)

LDQ     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        L_QRa <- (va)<63:0>               Ra <- (va)<63:0>

STB     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        (va) <- Rav<7:0>                  (va) <- Rav<7:0>

STW     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        (va) <- Rav<15:0>                 (va) <- Rav<15:0>

STL     L_va <- Rbv + SEXT(disp)          va <- Rbv + SEXT(disp)
        (va) <- Rav<31:0>                 (va) <- Rav<31:0>

STQ     va <- Rbv + SEXT(disp)            va <- Rbv + SEXT(disp)
        (va) <- L_QRav                    (va) <- Rav

RMAQI   L_va <- R4                        va <- R4
        L_QR4 <- (va){interlocked}        R5 <- (va){interlocked}
        (va){interlocked} <-              (va){interlocked} <-
          {L_QR4 AND L_QR6} + L_QR8         {R5 AND R6} + R7

VLDL    L_va <- Rbv                       va <- Rbv
        FOR i <- 0 TO VL-1                FOR i <- 0 TO VL-1
          BEGIN                             BEGIN
          Vc[i] <- (va)<31:0>               Vc[i] <- (va)<31:0>
          L_va <- va + Rav                  va <- va + Rav
          END                               END

VLDQ    L_va <- Rbv                       va <- Rbv
        FOR i <- 0 TO VL-1                FOR i <- 0 TO VL-1
          BEGIN                             BEGIN
          Vc[i] <- (va)<63:0>               Vc[i] <- (va)<63:0>
          L_va <- va + Rav                  va <- va + Rav
          END                               END
```

```
Instr         32-bit Mode                        64-bit Mode
----          -----------                        -----------

VGATHL  FOR i <- 0 TO VL-1                 FOR i <- 0 TO VL-1
          BEGIN                              BEGIN
          L_va <- Rav + Vb[i]                va <- Rav + Vb[i]
          Vc[i] <- (va)<31:0>                Vc[i] <- (va)<31:0>
          END                                END

VGATHQ  FOR i <- 0 TO VL-1                 FOR i <- 0 TO VL-1
          BEGIN                              BEGIN
          L_va <- Rav + Vb[i]                va <- Rav + Vb[i]
          Vc[i] <- (VA)<63:0>                Vc[i] <- (va)<63:0>
          END                                END

VSCATL  FOR i <- 0 TO VL-1                 FOR i <- 0 TO VL-1
          BEGIN                              BEGIN
          L_va <- Rav + Vb[i]                va <- Rav + Vb[i]
          (va) <- Vc[i]<31:0>                (va) <- Vc[i]<31:0>
          END                                END

VSCATQ  FOR i <- 0 TO VL-1                 FOR i <- 0 TO VL-1
          BEGIN                              BEGIN
          L_va <- Rav + Vb[i]                va <- Rav + Vb[i]
          (va) <- Vc[i]                      (va) <- Vc[i]
          END                                END
```

B.5.2 INTEGER ARITHMETIC INSTRUCTIONS

```
Instr        32-bit Mode                      64-bit Mode
-----        -----------                      -----------

ADD
SUB
MUL
DIV
REM      L_Rc <- L_Rbv op L_Rav            Rc <- Rbv op Rav

MULH     L_Rc <-                           NOT IMPLEMENTED
           {L_Rbv * L_Rav}<63:32>

UMULH    L_Rc <-                           NOT IMPLEMENTED
           {L_Rbv *U L_Rav}<63:32>

CMP      IF L_Rav op L_Rbv THEN            IF Rav op Rbv THEN
           L_Rc <- 1                         Rc <- 1
         ELSE                              ELSE
           L_Rc <- 0                         Rc <- 0

VADD
VSUB
VDIV
VMUL
VREM     FOR i <- 0 TO VL-1                FOR i <- 0 TO VL-1
           BEGIN                             BEGIN
                    {Vector op Vector}
           Vc[i] <- L_Va[i] op L_Vb[i]       Vc[i] <- Va[i] op Vb[i]
                    {Scalar op Vector}
           Vc[i] <- L_Rav op L_Vb[i]         Vc[i] <- Rav op Vb[i]
           END                               END

VCMP     VM <- 0                           VM <- 0
         FOR i <- 0 TO VL-1                FOR i <- 0 TO VL-1
           BEGIN                             BEGIN
                     {Vector op Vector}
           IF L_Va[i] op L_Vb[i] THEN        IF Va[i] op Vb[i] THEN
             VM<i> <- 1                        VM<i> <- 1
                     {Scalar op Vector}
           IF L_Rav op L_Vb[i] THEN          IF Rav OP Vb[i] THEN
             VM<i> <- 1                        VM<i> <- 1
           END                               END
```

B.5.3 LOGICAL AND SHIFT INSTRUCTIONS

```
Instr          32-bit Mode                      64-bit Mode
-----          -----------                      -----------

AND
BIC
OR
ORNOT
XOR
EQV     L_Rc <- L_Rbv op L_Rav          Rc <- Rbv op Rav

SLL
SRL
SRA     L_Rc <- op(L_Rbv,Rav<4:0>)      Rc <- op(Rbv,Rav<5:0>)

ROT     L_Rc <- op(L_Rbv,Rav<4:0>)      Rc <- op(Rbv,Rav<5:0>)

VAND
VOR
VXOR
VBIC
VORNOT
VEQV    FOR i <- 0 TO VL-1              FOR i <- 0 TO VL-1
          BEGIN                           BEGIN
                   {Vector op Vector}
          Vc[i] <- L_Va[i] op L_Vb[i]     Vc[i] <- Va[i] op Vb[i]
                   {Scalar op Vector}
          Vc[i] <- L_Rav op L_Vb[i]       Vc[i] <- Rav op Vb[i]
          END                             END

VMERGE  FOR i <- 0 TO VL-1              FOR i <- 0 TO VL-1
          BEGIN                           BEGIN
                   {Vector op Vector}
          IF VM<i> EQ 0 THEN              IF VM<i> EQ 0 THEN
            Vc[i] <- Va[i]                  Vc[i] <- Va[i]
          ELSE                            ELSE
            Vc[i] <- Vb[i]                  Vc[i] <- Vb[i]
                   {Scalar op Vector}
          IF VM<i> EQ 0 THEN              IF VM<i> EQ 0 THEN
            Vc[i] <- L_QRav                 Vc[i] <- Rav
          ELSE                            ELSE
            Vc[i] <- Vb[i]                  Vc[i] <- Vb[i]
          END                             END

VSLL
VSRL    FOR i <- 0 TO VL-1              FOR i <- 0 TO VL-1
          BEGIN                           BEGIN
                   {vector op vector}
          Vc[i] <-                        Vc[i] <-
            op(L_Vb[i],Va[i]<4:0>)          op(Vb[i],Va[i]<5:0>)
                   {vector op scalar}
          Vc[i] <-                        Vc[i] <-
            op(L_Vb[i],Rav<4:0>)            op(Vb[i],Rav<5:0>)
          END                                 END
```

B.5.4 FLOATING POINT INSTRUCTIONS

```
Instr          32-bit Mode                        64-bit Mode
-----          -----------                        -----------

ADDF
SUBF
DIVF
MULF    L_Rc <- L_Rbv op L_Rav           L_Rc <- L_Rbv op L_Rav

ADDG
SUBG
MULG
DIVG    L_QRc <- L_QRbv op L_QRav        Rc <- Rbv op Rav

CMPF    IF L_Rav op L_Rbv THEN           IF L_Rav op L_Rbv THEN
          Rc <- 1                          Rc <- 1
        ELSE                             ELSE
          Rc <- 0                          Rc <- 0

CMPG    IF L_QRav op L_QRbv THEN         IF Rav op Rbv THEN
          Rc <- 1                          Rc <- 1
        ELSE                             ELSE
          Rc <- 0                          Rc <- 0

CVTFG   L_QRc <- cvt(L_Rav)              Rc <- cvt(L_Rav)

CVTGF   L_Rc <- cvt(L_QRav)              L_Rc <- cvt(Rav)

CVTFI   L_Rc <- cvt(L_Rav)               Rc <- cvt(L_Rav)

CVTGI   L_Rc <- cvt(L_QRav)              Rc <- cvt(Rav)

CVTIF   L_Rc <- cvt(L_Rav)               L_Rc <- cvt(Rav)

CVTIG   L_QRc <- cvt(L_Rav)              Rc <- cvt(Rav)

VADDF
VSUBF
VDIVF
VMULF   FOR i <- 0 TO VL-1               FOR i <- 0 TO VL-1
          BEGIN                            BEGIN
                 {Vector op Vector}
          Vc[i] <- L_Va[i] op L_Vb[i]      Vc[i] <- L_Va[i] op L_Vb[i]
                 {Scalar op Vector}
          Vc[i] <- L_Rav op L_Vb[i]        Vc[i] <- L_Rav op L_Vb[i]
          END                              END
```

```
Instr            32-bit Mode                           64-bit Mode
----             -----------                           -----------

VADDG
VSUBG
VDIVG
VMULG   FOR i <- 0 TO VL-1                     FOR i <- 0 TO VL-1
          BEGIN                                  BEGIN
                   {Vector op Vector}
          Vc[i] <- Va[i] op Vb[i]                Vc[i] <- Va[i] op Vb[i]
                   {Scalar op Vector}
          Vc[i] <- L_QRav op Vb[i]               Vc[i] <- Rav op Vb[i]
          END                                    END

VCMPF   VM <- 0                                VM <- 0
        FOR i <- 0 TO VL-1                     FOR i <- 0 TO VL-1
          BEGIN                                  BEGIN
                   {Vector cmp Vector}
          IF L_Va[i] op L_Vb[i] THEN             IF L_Va[i] op L_Vb[i] THEN
            VM<i> <- 1                             VM<i> <- 1
          ELSE                                   ELSE
            VM<i> <- 0                             VM<i> <- 0
                   {Scalar cmp Vector}
          IF L_Rav op L_Vb[i] THEN               IF L_Rav op L_Vb[i] THEN
            VM<i> <- 1                             VM<i> <- 1
          ELSE                                   ELSE
            VM<i> <- 0                             VM<i> <- 0
          END                                    END

VCMPG   VM <- 0                                VM <- 0
        FOR i <- 0 TO VL-1                     FOR i <- 0 TO VL-1
          BEGIN                                  BEGIN
                   {Vector cmp Vector}
          IF Va[i] op Vb[i] THEN                 IF Va[i] op Vb[i] THEN
            VM<i> <- 1                             VM<i> <- 1
          ELSE                                   ELSE
            VM<i> <- 0                             VM<i> <- 0
                   {Scalar cmp Vector}
          IF L_QRav op Vb[i] THEN                IF Rav op Vb[i] THEN
            VM<i> <- 1                             VM<i> <- 1
          ELSE                                   ELSE
            VM<i> <- 0                             VM<i> <- 0
          END                                    END
```

```
Instr           32-bit Mode                          64-bit Mode
----            -----------                          -----------

VCVTFG  FOR i <- 0 TO VL-1                   FOR i <- 0 TO VL-1
          BEGIN                                BEGIN
          Vc[i] <- cvt(L_Va[i])                Vc[i] <- cvt(L_Va[i])
          END                                  END

VCVTGF  FOR i <- 0 TO VL-1                   FOR i <- 0 TO VL-1
          BEGIN                                BEGIN
          Vc[i] <- cvt(Va[i])                  Vc[i] <- cvt(Va[i])
          END                                  END

VCVTFI  FOR i <- 0 TO VL-1                   FOR i <- 0 TO VL-1
          BEGIN                                BEGIN
          Vc[i] <- cvt(L_Va[i])                Vc[i] <- cvt(L_Va[i])
          END                                  END

VCVTGI  FOR i <- 0 TO VL-1                   FOR i <- 0 TO VL-1
          BEGIN                                BEGIN
          Vc[i] <- cvt(Va[i])                  Vc[i] <- cvt(Va[i])
          END                                  END

VCVTIF
VCVTIG  FOR i <- 0 TO VL-1                   FOR i <- 0 TO VL-1
          BEGIN                                BEGIN
          Vc[i] <- cvt(L_Va[i])                Vc[i] <- cvt(Va[i])
          END                                  END
```

B.5.5 CONTROL INSTRUCTIONS

```
Instr           32-bit Mode                         64-bit Mode
-----           -----------                         -----------

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

FOB     IF Rav<0> EQ 1 THEN                 IF Rav<0> EQ 1 THEN
          {FOB exception}                     {FOB exception}

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

B.5.6 MISCELLANEOUS INSTRUCTIONS

```
Instr           32-bit Mode                         64-bit Mode
-----           -----------                         -----------

BPT     {push current L_PC and              {push current PC and
          PS on kernel stack}                 PS on kernel stack}
        {Change Mode to Kernel}             {Change Mode to Kernel}
        {dispatch through SCB vector}       {dispatch through SCB vector}

BUGCHK  {push current L_PC and              {push current PC and
          PS on kernel stack}                 PS on kernel stack}
        {Change Mode to Kernel}             {Change Mode to Kernel}
        {dispatch through SCB vector}       {dispatch through SCB vector}

DRAIN   {Stall instruction issuing until all prior instructions completed}

IFLUSH  {Invalidate instruction prefetch and instruction cache}

IOTA    j <- 0                              j <- 0
        tmp <- 0                            tmp <- 0
        FOR i <- 0 TO VL-1                  FOR i <- 0 TO VL-1
          BEGIN                               BEGIN
          IF VM<i> EQ 1 THEN                  IF VM<i> EQ 1 THEN
            BEGIN                               BEGIN
            Vc[j] <- tmp                        Vc[j] <- tmp
            j <- j + 1                          j <- j + 1
            END                                 END
          L_tmp <- tmp + Rav                  tmp <- tmp + Rav
          END                                 END
        VC <- j                             VC <- j

MOVPS   L_R4 <- PS                          R4 <- ZEXT(PS)

PROBE   L_R4 contains the base address      R4 contains the base address
        L_R5 contains the signed offset     R5 contains the signed offset
        R6 contains the access mode         R6 contains the access mode
        R7<0> <- {success}                  R7<0> <- {success}
        R7<63:1> <- 0                       R7<63:1> <- 0

RDVC    L_Rc <- ZEXT(VC)                    Rc <- ZEXT(VC)

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

RDVL    L_Rc <- ZEXT(VL)                    Rc <- ZEXT(VL)

WRVL    VL <- Rav<5:0>                      VL <- Rav<5:0>

RDVMH   L_Rc <- VM<63:32>                   Rc <- VM

RDVML   L_Rc <- VM<31:0>                    Rc <- VM

WRVMH   VM<63:32> <- L_Rav                  VM <- Rav
```

```
WRVML   VM<31:0> <- L_Rav             VM <- Rav
```

```
Instr           32-bit Mode                        64-bit Mode
-----           -----------                        -----------

REI     tmpl <- (SP)<31:0>                 tmpl <- (SP)<31:0>
        IF tmpl<31> EQ 0 THEN              IF tmpl<31> EQ 0 THEN
          {return to 32-bit mode}            {return to 32-bit mode}
        ELSE                               ELSE
          {illegal operation}                {return to 64-bit mode}

SWASTEN tmp <- R4<0>                       tmp <- R4<0>
        L_R4 <- ZEXT(ASTEN<PS<CM>>)        R4 <- ZEXT(ASTEN<PS<CM>>)
        ASTEN<PS<CM>> <- tmp               ASTEN<PS<CM>> <- tmp
```

B.5.7 PRIVILEGED INSTRUCTIONS

Instr	32-bit Mode	64-bit Mode
HALT	{halt processor or enter restart sequence}	
MFPR	IPR specific results are returned in L_R4, L_R5, L_R6	IPR specific results are returned in R4, R5, R6
MTPR	L_R4 and L_R5 contain IPR specific source operands	R4 and R5 contain IPR specific source operands

RMAQIP	l_va <- L_QR4 AND {NOT 7}
	L_QR4 <- (addr){interlocked}
	(va){interlocked} <-
	{L_QR4 AND L_QR6} + L_QR8
		va <- R4 AND {NOT 7}
		R5 <- (addr){interlocked}
		(va){interlocked} <-
		{R5 AND R6} + R7

SWPCTX	L_QR4 contains the physical address of the HWPCB.	R4 contains the physical address of the HWPCB.

SWIPL	tmp <- R4<2:0>
	L_R4 <- ZEXT(PS<IPL>)
	PS<IPL> <- tmp
		tmp <- R4<2:0>
		R4 <- ZEXT(PS<IPL>)
		PS<IPL> <- tmp

TBFLUSH	{Invalidate all TB entries}	

B.6 MEMORY MANAGEMENT

B.6.1 Virtual Address Space

A virtual address is a 64-bit unsigned integer specifying a byte location within the virtual address space. The page size ranges from 8 Kbytes to 64 Kbytes.

B.6.2 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. The virtual address consists of three Segment Number fields, and a Byte Within Page field, as shown in Figure B-1

```
 6
 3                                                                       0
+------+------------+------------+------------+------------------+
| Rsvd | Seg1_Number | Seg2_Number | Seg3_Number | Byte Within Page |
+------+------------+------------+------------+------------------+
```

Figure B-1: Virtual Address Format

The byte within page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8 KBytes, 16 Kbytes, 32 KBytes, and 64 KBytes. All three segment number fields are the same size for a given implementation. The segment number field is a function of the page size; all page table entries at any given level fit in exactly one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows the virtual and physical address size also grows (as shown in Table B-2).

Table B-2: Virtual Address Options

Page Size (Bytes)	Byte Offset (bits)	Segment Size (bits)	Virtual Address (bits)	Physical Address (bits)
8 K	13	10	43	45
16 K	14	11	47	46
32 K	15	12	51	47
64 K	16	13	55	48

B.6.3 Physical Address Space

Physical addresses are at most 48 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.

B.6.4 Address Translation

Address translation is performed by accessing entries in a three-level page table structure. The Page Table Base Register (PTBR) contains the physical page frame number of the highest level (Segment 1) page table. Bits <Seg1_Number> 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 <Seg2_Number> of the virtual address are used to index into the second level page table to obtain the physical page frame number of the base of the third level (Segment 3) page table. Bits <Seg3_Number> of the virtual address are used to index the third level page table to obtain the physical Page Frame Number (PFN) of the page being referenced. The PFN is concatenated with virtual address bits <Byte_Within_Page> to obtain the physical address of the location being accessed.

The processor uses a 64-bit Page Table Entry that is identical to the one used in the 32-bit architecture. The algorithm to generate a physical address from a virtual address is similar to the one used in the 32-bit architecture with the addition of one more level of mapping.

B.7 PROCESSOR STATE

```
 3 3
 1 0                                                  8 7   5 4 3 2 1 0
+-+---------------------------------------------------+------+-+-+-+----+
|V|                                                   |      |V|V|V|    |
|A|                       MBZ                         | IPL  |E|E|M| CM |
|X|                                                   |      |N|F|M|    |
+-+---------------------------------------------------+------+-+-+-+----+
```

Figure B-2: Processor Status

Bits <30:0> of the PS are identical to the PS in the 32-bit architecture. Bit <31> is the Virtual Address eXtension (VAX) bit. When set, the processor is in 64-bit mode. When clear the processor is in 32 bit mode.

```
 3
 1                                                          2 1 0
+-----------------------------------------------------------+----+
|                                                           | I  |
|            Instruction Virtual Address <31:0>             | G  |
|                                                           | N  |
+-----------------------------------------------------------+----+
|                                                                |
|            Instruction Virtual Address <63:32>                 |
|                                                                |
+----------------------------------------------------------------+
```

Figure B-3: Program Counter

B.8 EXCEPTION STACK FRAMES

In 32-bit mode, the exception stack frames are identical to those in the 32-bit architecture. The exception stack frames for 64-bit mode are shown in the subsequent sections. A processor always enters 64-bit mode when an exception occurs.

B.8.1 Arithmetic Traps

```
 3
 1                                                             0
+---------------------------------------------------------------+
|                                                               |
|                      Exception Summary                        | :SP
|                                                               |
+---------------------------------------------------------------+
|                       Vector Register                         |
|                       Write  Mask for                         |
|                     Registers  V0 - V15                       |
+---------------------------------------------------------------+
|                       Scalar Register                         |
|                       Write  Mask for                         |
|                     Registers  R0 - R31                       |
+---------------------------------------------------------------+
|                       Scalar Register                         |
|                       Write  Mask for                         |
|                     Registers R32 - R64                       |
+---------------------------------------------------------------+
|                                                               |
|                    Processor Status (PS)                      |
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                            Zero                               |
|                                                               |
+---------------------------------------------------------------+
|                           Virtual                             |
|                  Address <31:0> of Next                       |
|                         Instruction                           |
+---------------------------------------------------------------+
|                           Virtual                             |
|                  Address <63:32> of Next                      |
|                         Instruction                           |
+---------------------------------------------------------------+
```

Figure B-4: Arithmetic Trap Exception Frame

B.8.2 Scalar Alignment Fault

```
   3
   1                                                          0
  +-----------------------------------------------------------+
  |                         Virtual                           |
  |                   Address <31:0> of                       |  :SP
  |                        Reference                          |
  +-----------------------------------------------------------+
  |                         Virtual                           |
  |                   Address <63:32> of                      |
  |                        Reference                          |
  +-----------------------------------------------------------+
  |                                                           |
  |                  Faulting Instruction                     |
  |                                                           |
  +-----------------------------------------------------------+
  |                                                           |
  |                          Zero                             |
  |                                                           |
  +-----------------------------------------------------------+
  |                                                           |
  |                Processor Status (PS)                      |
  |                                                           |
  +-----------------------------------------------------------+
  |                                                           |
  |                          Zero                             |
  |                                                           |
  +-----------------------------------------------------------+
  |                          Virtual                          |
  |              Address <31:0> of Faulting                   |
  |                       Instruction                         |
  +-----------------------------------------------------------+
  |                          Virtual                          |
  |              Address <63:32> of Faulting                  |
  |                       Instruction                         |
  +-----------------------------------------------------------+
```

Figure B-5: Scalar Alignment Fault Exception Frame

B.8.3 Vector Alignment Abort

```
  3
  1                                                              0
 +---------------------------------------------------------------+
 |                           Virtual                             |
 |                     Address <31:0> of                         | :SP
 |                          Reference                            |
 +---------------------------------------------------------------+
 |                           Virtual                             |
 |                     Address <63:32> of                        |
 |                          Reference                            |
 +---------------------------------------------------------------+
 |                                                               |
 |                   Processor Status (PS)                       |
 |                                                               |
 +---------------------------------------------------------------+
 |                                                               |
 |                            Zero                               |
 |                                                               |
 +---------------------------------------------------------------+
 |                           Virtual                             |
 |                 Address <31:0> of Next                        |
 |                         Instruction                           |
 +---------------------------------------------------------------+
 |                           Virtual                             |
 |                 Address <63:32> of Next                       |
 |                         Instruction                           |
 +---------------------------------------------------------------+
```

Figure B-6: Vector Alignment Abort Exception Frame

B.8.4 BPT, BUGCHK, Vector Enable, And Privileged Instruction Faults

```
 3
 1                                                            0
+--------------------------------------------------------------+
|                                                              |
|                  Processor Status (PS)                       | :SP
|                                                              |
+--------------------------------------------------------------+
|                                                              |
|                          Zero                                |
|                                                              |
+--------------------------------------------------------------+
|                         Virtual                              |
|                Address <31:0> of Faulting                    |
|                       Instruction                            |
+--------------------------------------------------------------+
|                         Virtual                              |
|                Address <63:32> of Faulting                   |
|                       Instruction                            |
+--------------------------------------------------------------+
```

Figure B-7: BPT, BUGCHK, and Privileged Instruction Fault Exception Frame

B.8.5 FOB, Illegal Operand, And Reserved Opcode Faults

```
 3
 1                                                              0
+----------------------------------------------------------------+
|                                                                |
|                      Faulting  Instruction                     | :SP
|                                                                |
+----------------------------------------------------------------+
|                                                                |
|                              Zero                              |
|                                                                |
+----------------------------------------------------------------+
|                                                                |
|                     Processor Status (PS)                      |
|                                                                |
+----------------------------------------------------------------+
|                                                                |
|                              Zero                              |
|                                                                |
+----------------------------------------------------------------+
|                             Virtual                            |
|                  Address <31:0> of Faulting                    |
|                           Instruction                          |
+----------------------------------------------------------------+
|                             Virtual                            |
|                  Address <63:32> of Faulting                   |
|                           Instruction                          |
+----------------------------------------------------------------+
```

Figure B-8: FOB, Illegal Operand, and Reserved Opcode Fault Exception Frame

B.8.6 Memory Management Faults

```
 3
 1                                                              1 0
+-------------------------------------------------------------+-+-+
|                     Bits <31:0> of Related                      |
|                      Virtual Address in                         | :SP
|                             Page                                |
+-----------------------------------------------------------------+
|                     Bits <63:32> of Related                     |
|                      Virtual Address in                         |
|                             Page                                |
+-------------------------------------------------------------+-+-+
|                                                               |R|
|                             Zero                              |/|
|                                                               |W|
+-------------------------------------------------------------+-+-+
|                                                                 |
|                             Zero                                |
|                                                                 |
+-----------------------------------------------------------------+
|                                                                 |
|                     Processor Status (PS)                       |
|                                                                 |
+-----------------------------------------------------------------+
|                                                                 |
|                             Zero                                |
|                                                                 |
+-----------------------------------------------------------------+
|                            Virtual                              |
|                   Address <31:0> of Faulting                    |
|                          Instruction                            |
+-----------------------------------------------------------------+
|                            Virtual                              |
|                   Address <63:32> of Faulting                   |
|                          Instruction                            |
+-----------------------------------------------------------------+
```

Figure B-9: Memory Management Fault Exception Frame

B.8.7 Machine Check

```
 3
 1                                                            0
+--------------------------------------------------------------+
|                           Number                             |
|                             of                               | :SP
|                        Bytes Pushed                          |
+--------------------------------------------------------------+
|                                                              |
|                            Zero                              |
|                                                              |
+--------------------------------------------------------------+
                              .
                              .
                              .
                       An even number of
                        implementation
                           specific
                          longwords
                              .
                              .
                              .
+--------------------------------------------------------------+
|                                                              |
|                   Processor Status (PS)                      |
|                                                              |
+--------------------------------------------------------------+
|                                                              |
|                            Zero                              |
|                                                              |
+--------------------------------------------------------------+
|                          Virtual                             |
|                 Address <31:0> of Next                       |
|                        Instruction                           |
+--------------------------------------------------------------+
|                          Virtual                             |
|                 Address <63:32> of Next                      |
|                        Instruction                           |
+--------------------------------------------------------------+
```

Figure B-10: Machine Check Abort Exception Frame

B.8.8 Stack Alignment Abort

```
 3
 1                                                            0
+--------------------------------------------------------------+
|                                                              |
|                   Processor Status (PS)                      | :SP
|                                                              |
+--------------------------------------------------------------+
|                                                              |
|                           Zero                               |
|                                                              |
+--------------------------------------------------------------+
|                         Virtual                              |
|                  Address <31:0> of Next                      |
|                       Instruction                            |
+--------------------------------------------------------------+
|                         Virtual                              |
|                  Address <63:32> of Next                     |
|                       Instruction                            |
+--------------------------------------------------------------+
```

Figure B-11: Stack Alignment Abort Exception Frame

B.8.9 Vector Exceptions

```
 3 3   2 2 2        2 2 1 1 1         1 1
 1 0   8 7 6        1 0 9 8 7         2 1         6 5         0
+-+-----+-+-----------+-+-+-+-----------+-----------+-----------+
|V|  E  |L|           |S|O|D|           |           |    SRC    |
|F|  T  |V|   Zero    |T|P|T|    ELT    |    CNT    |     or    |
|S|  Y  |F|           |R|R|Y|           |           |    DST    |
+-+-----+-+-----------+-+-+-+-----------+-----------+-----------+
|                                                               |
|                             Zero                              |
|                                                               |
+---------------------------------------------------------------+
|                   Bits <31:0> of Related                      |
|                     Virtual  Address in                       |
|                            Page                               |
+---------------------------------------------------------------+
|                   Bits <63:32> of Related                     |
|                     Virtual  Address in                       |
|                            Page                               |
+---------------------------------------------------------------+
|                           Vector                              |
|                            Base                               |
|                       Address <31:0>                          |
+---------------------------------------------------------------+
|                           Vector                              |
|                            Base                               |
|                       Address <63:32>                         |
+---------------------------------------------------------------+
|                        Stride <31:0>                          |
|                             or                                |
|                 Index Vector Register Number                  |
+---------------------------------------------------------------+
|                        Stride <63:32>                         |
|                             or                                |
|                            Zero                               |
+---------------------------------------------------------------+
```

Figure B-12: Vector Exception Information Frame

B.8.10 SCB Vectors

```
 3
 1                                                             2 1 0
+-------------------------------------------------------------+---+
|                         Virtual                             | S |
|                    Address <31:0> of                        | B |
|                     Service Routine                         | Z |
+-------------------------------------------------------------+---+
|                         Virtual                                 |
|                    Address  <63:32> of                          |
|                     Service Routine                             |
+-----------------------------------------------------------------+
```

Figure B-13: System Control Block Vector

B.9 64-BIT MODE INTERNAL PROCESSOR REGISTERS

Table B-3: Internal Processor Register (IPR) Summary

Register Name	Mnemonic	Access	R4	R5	R6
Address Space Number	ASN	R	number		
AST Enable	ASTEN	R	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		
Stack Pointer Registers					
Executive Stack Pointer	ESP	R/W	address		
Supervisor Stack Pointer	SSP	R/W	address		
User Stack Pointer	USP	R/W	address		
Interval Clock Int. Enable	ICIE	R/W	enable		
Interprocessor Int. Enable	IPIE	R/W	enable		
Interprocessor Int. Request	IPIR	W	number		
Privileged Context Block Base	PCBB	R	address		
Processor Base Register	PRBR	R/W	value		
Processor Serial Number	PRSN	R	serial		
Page Table Base Register	PTBR	R	frame		
System Control Block Base	SCBB	R/W	address		
System Identification	SID	R	ident		
Software Int. Request Register	SIRR	W	level		
Software Int. Summary Register	SISR	R	mask		
Trans. Buffer Check	TBCHK	R	number	address	status
Trans. Buffer Invalidate ASN	TBIASN	W	number		
Trans. Buffer Invalidate Single	TBIS	W	number	address	
Time Of Year	TOY	R/W	time		
Who-Am-I	WHAMI	R	number		

INDEX